UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Fault analysis of edge router Linux system message log files with machine learning

Trabajo Fin de Máster

Máster Universitario en Gestión de la Información

Autor: Péter László Lógó

Tutor: Prof. José Hernández-Orallo

2021

Abstract

In our everyday life, many things are done electronically, e-mails, buying different tickets, navigation, and so on. These devices must always work correctly. Linux-based routers are no exception. The devices create a log file about everything that happens to them, which in many cases is in an unreadable state for humans. When such a device fails, professional workers spend many hours searching for individual errors and often fail to detect them.

The main goal of this research is to create a System log files analyzer for predictive maintenance because there is currently no quick and effective program for this. Therefore, several publicly available programs were overviewed that are capable of individual log files, especially system log files, and analyzing them. It was investigated how they are structured and what types of logs are analyzed in what way.

Syslog files were examined with different labels in this dissertation, which were converted using text mining methods. Then they were analyzed with several models, including decision trees, random forests, the XGBoost model, and neural networks, and they predicted labels for each log file. To make this more successful, text-mining methods were applied, and each Syslog was sliced into sequences. These transformations gave much more transparent and accurate results. After teaching and testing the models, the results were obtained, which were evaluated with different indicators, such as accuracy and a confusion matrix.

It was also part of this thesis the writing of a program for a Raspberry Pi that can give the professional workers guidance on what kind of failures happened in the system, thus reducing their time spent on maintenance and more efficient debugging. The program was written in Python and tested on two different types of Raspberry Pi.

In conclusion, several types of errors can occur in a log file, and machine learning methods can significantly help the work of professionals by guiding analysis. With this AI technology, professionals know more precisely what error they need to look for in which sequence; they do not have to look through all the rows. This allows them to spend more time on improvements and upgrades, and not have to deal with maintenance.

Keywords: text mining, Syslog, TF-IDF, analysis, Python

Table of Contents

Acknowledgments

Throughout the writing of the dissertation, I have received a great deal of support and assistance.

First, I would like to thank my supervisor, Prof. Gábor Szűcs, whose expertise was invaluable in formulating the research questions and methodology. Your feedback pushed me to sharpen my thinking and brought my work to a higher level.

I would also like to thank the professor of the Universitat Politècnica de València, José Hernández-Orallo, for his acceptance and support, who helped me find a solution to my problems with his feedback and thoughts.

I would like to acknowledge my colleagues from my internship at Flex for their wonderful collaboration. I would particularly like to single out my supervisors at Flex, Gábor Magda, Milán Szücs, and Tamás Horváth. Thank you for the many wonderful conversations in the meetings, your support, and all of the given opportunities.

In addition, I would like to thank my parents, family for their wise counsel and their extreme support. You are always there for me. Köszönöm!

1 Introduction

Nowadays, the automation of services or production is becoming more and more common. During or after this automation, some products may fail and break down. In this thesis, the general goal is to find the answer to whether there is already a solution to automatically monitor Linux-based edge routers after a failure. In the real life the log files monitoring is essential to help the maintenance processes.

According to my best knowledge, it does not exist any program that could analyze the log files containing the system message. There are some programs which could only interpret the text generated by themself. Therefore, in this study, these system log files using machine learning and neural network are analyzed.

Raspberry Pis are easy-to-carry, small-sized computers. Since failures can occur in any country, it is enough for I.T. professionals to bring this small device with them and connect the appropriate elements to it for analysis; therefore, a program was developed for a Linux-based Raspberry Pi. Most of the times the system log files contain very sensitive data; therefore the companies try to avoid the online solutions, it can be risky to run the scripts throught servers or upload the files into the cloud. These are the reasons why the program had to be written for this device.

This dissertation has eight chapters followed by references and lists of tables, figures, and equations. *The entire work is based on the principle that the definition and theorems are taken from the cited references with the proper citation as they are.* At first, a comprehensive literature overview is conducted. Section two contains the theoretical background in brief. Then the programming steps and the coding itself are explained. The dissertation is closed by the results and the conclusions. The Cross-Industry Standard Process for Data Mining (*CRISP-DM*) methodology is applied throughout the project, which means the first step is business understanding. Initially, it has to understand how the system works log files, which system can write in the files, and why. The next important step is preparing the data, which means the system logs have been cleaned and reduce noise. For these steps, the Term Frequency-Inverse Document Frequency (*TF-IDF*) and other methods were used. The various transformations and tokenization are followed by modeling. After the evaluation, the last step was deployment. The thesis is closed by the sections of the conclusions and the references.

2 Methodology and Tools

In this chapter, a brief theoretical overview is presented to understand the data mining and the connected tools that were used during the analysis better.

2.1 Data mining

2.1.1 A brief overview of the fundamentals of data mining

Data mining is the process of finding patterns, correlations, and anomalies within datasets to extract insight and knowledge, and use it for explaining the data or making predictions. Using many of the data mining techniques, companies can use this information, for example, to increase revenues, cut costs, improve customer relationships, reduce risks, and predict the result of a game, and more [1].

Nowadays, data mining is widespread in business, science, engineering, medicine, and more several other applications. The bank sectors use data mining of credit card transactions, stock market movements, or the national government to use several data science tools for national security. These application areas are just the tip of the iceberg for data mining applications. Big Data is now commonplace, with data collection becoming relatively cheap and the proliferation of devices capable of collecting data. The most active techniques explored today come from Machine Learning, such as Deep Learning. Deep Learning can capture dependencies and complex patterns far beyond other techniques; it reignites some of the biggest challenges in the world of data mining, data science, and artificial intelligence [2].

2.1.2 About text mining

One of the earliest examples of text summarization, text mining, and classification was library catalogs. The earliest library catalog is attributed to Thomas Hyde for the Bodleian Library at the University of Oxford in the 17th century. In 1876, Melvin Dewey introduced the index card to form a library card catalog [3].

Nowadays, more and more information is in an unstructured and semi-structured format, like open-ended survey answers, news, call-center notes, and even books or web forms. Text mining is the process of analyzing textual collections of material to understand key concepts and topics and explore hidden contexts and trends without knowing the words or phrases used by the authors to express the concepts.

The text mining process usually includes the following steps:

- *Identify the text you want to mine*: Identify the text and prepare it for mining.
- *Mine the text and extract the structured data*. To the source text apply the text mining methods.
- *Build concept and category models*. Define key concepts, categories. Determine the best categories and concepts for scoring.
- *Analysis of structured data*. Use traditional data mining techniques such as clustering, classification, and predictive modeling to explore the relationships between concepts. Combine extracted concepts with other structured data to predict future behavior based on the concepts.

2.1.3 CRISP-DM and text mining methods

The Cross-Industry Standard Process for Data Mining (*CRISP-DM*) stands for a cross-industry process for data mining, data science. The methodology provides a structured approach to planning and doing data mining projects; unfortunately, this model belongs to an ideal world, an idealized sequence of events. Therefore, in practice, many of the tasks can be performed differently, in a separate order. The CRISP-DM methodology will often be necessary to backtrack to previous assignments and repeat specific actions. For example, Business and datasets understanding are one of the most critical tasks. If the data miners do not understand what the company wants, there will be many unnecessary costs [6].

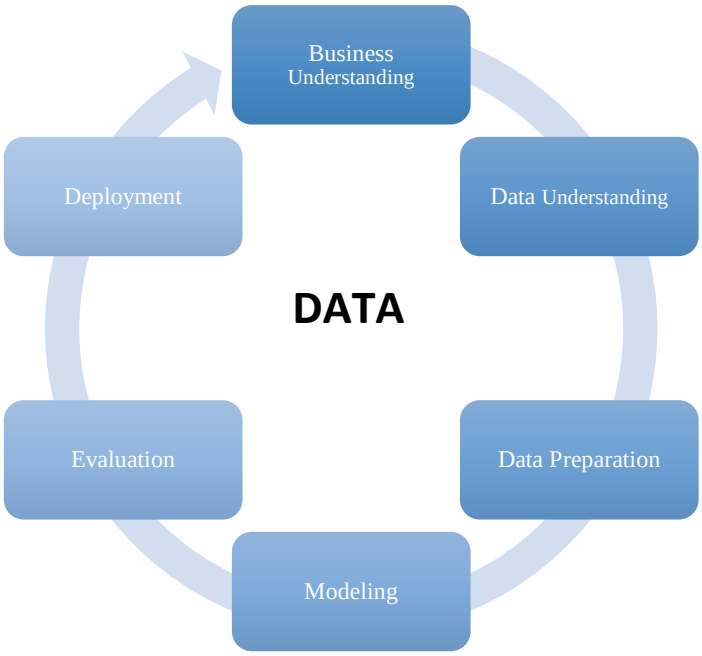

Figure 1: CRISP-DM reference model. The boxes represent the six phases of the life cycle model

The CRISP-DM life cycle model consists of six phases (*Figure 1*). The sequence of the stages is not strict; therefore, most data mining projects move many times back and forth between evaluation and modeling as necessary. The model is flexible, and it can be customized easily. The six phases are business understanding (determining the purpose of the study), data understanding (data exploration and understanding), data preparation, modeling, evaluation, and deployment [7].

In [3], the authors presented a model for text mining based on the CRISP-DM method. As they wrote, "Within the six phases, CRISP-DM methodology provides comprehensive coverage of all of the activities involved in carrying out data mining projects. Because the primary distinction between data mining and text mining is simply the type of data involved in the knowledge discovery process, we adopt CRISP-DM as a foundation upon which to derive the text mining methodology followed in this book."

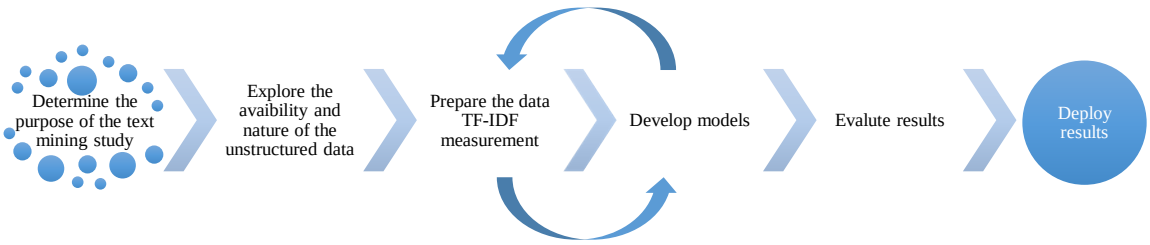

Figure 2: Text mining process flow. These are the general steps

Figure 2 shows the flow of the text mining process. The first phase (*Determine the purpose of the text mining study*) works like any other project's first step, so the text mining study starts with determining the purpose of the study. This requires a thorough understanding of the business case and what the study aims to accomplish. In order to achieve this understanding and define the aims precisely, we must assess the nature of the problem that initiated the study.

The second phase (*Explore the availability and the quality of the data*) include a few tasks, like

- Identification of the textual data sources (digitized or paper-based; internal or external to the organization)
- Assessment of the accessibility and usability of the data
- Collection of an initial set of data
- Exploration of the richness of the data (does it have the information content needed for the text mining study)
- Assessment of the quantity and quality of the data: Once the exploration is concluded with positive outcomes, the next phase is to collect and integrate large amounts of data from various sources used in the study.

The third and fourth phases (*Prepare data and Develop models*) present the most significant differences between data mining and text mining. Within the context of knowledge discovery, the primary purpose of text mining is to process unstructured, textual data and structured and semi-structured data to extract novel, meaningful, and actionable knowledge/information for better decision making.

In the fifth phase (*Evaluate the results*), if the models are developed and assessed for accuracy and quality from a data analysis perspective, we must verify and validate the proper execution of all of the activities.

At the end, in the sixth phase (*Deploy the results*), if the models and the modeling process successfully pass the assessment process; they can be deployed [3].

2.2 Term Frequency-Inverse Document Frequency weight

After reviewing the data mining and CRISP-DM methodology, we will review the TF-IDF measure because it is used for normalization later on.

The Term Frequency-Inverse Document Frequency (*TF-IDF*) weight is often used in text mining and information retrieval. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in a document but is offset by the frequency of the word in the corpus. Search engines often use variations of the TF-IDF weighting scheme as a central tool in scoring and ranking a document's relevance given a user query. TF-IDF can be successfully used for stop-words filtering in various subject fields, including classification and text summarization [6].

Typically, the TF-IDF weight is composed of two terms. The first computes the normalized Term Frequency (*TF*), which means the number of times a word appears in a document, divided by the total number of words in that document. The second term is the Inverse Document Frequency (*IDF*), computed as the logarithm of the number of the documents in the corpus divided by the number of documents where the specific term appears.

Term Frequency, which measures how frequently a term occurs in a document. Since every document is different in length, it is possible that a term would appear much more time in long documents than shorter ones. Thus, the document length as a way of normalization often divides the term frequency (*Equation 1*):

$$TF(t) = \frac{Number\ of\ times\ term\ t\ appears\ in\ a\ document}{Total\ number\ of\ term\ in\ the\ document}$$

Equation 1: The equation for the Term Frequency

Inverse Document Frequency, which measures how important a term is. While computing term frequency, all terms are considered equally important. However, it is known that certain terms, such as "*that*", "*is*", "*of*", and "*the*", may appear many times but have little importance. Thus, we need to weigh down the frequent terms while scaling up the rare ones by computing the following [7] (*Equation 2*):

$$IDF(t) = \log \frac{Total\ number\ of\ documents}{Number\ of\ document\ with\ term\ t\ in\ it}$$

Equation 2: The equatin for the Inverse Document Frequency

As a result of the previous equitations one can get TF-IDF what is the multiplications of Equation 1 and Equation 2 (*Equation 3*).

$$TFIDF(t) = TF(t) * IDF(t)$$

Equation 3: The equation for Term Frequency-Inverse Document Frequency

2.3 Regular expressions

Regular expressions were very useful during data preparation. It was easier to search and replace some specific words. In this section, we explain how the regex works.

A regular expression (*regex*) is a sequence of characters that define a search pattern. Usually, such patterns are used by string searching algorithms to find and replace or find operations on strings or input validation. It is a technique developed in theoretical computer science and formal language theory. Regular expressions are widely used in different areas, for example in search engines, or text editors. With this process, the user can check the e-mail address is correct or not, find and replace the date format with something else, or validate the bank account number.

A regular expression specifies a set of strings that each individual string in the set should match it; the functions in this module let you check if a particular string matches a given regular expression (or whether a given regular expression corresponds to a given string that applies to the same thing).

Regular expressions can be concatenated to create new regular expressions. "*A*" and "*B*" are both regular expressions, in this case, "*AB*" is also a regular expression. Usually, if a string "*a*" matches with "*A*" and another string "*b*" matches with "*B*", the string "*ab*" will match with "*AB*". This holds unless "*A*" or "*B*" contains low precedence operations, boundary conditions between "*A*" and "*B*"; or have numbered group references. Thus, complex expressions can easily be constructed from simpler primitive expressions like those described here [8].

In this part, some regular expressions were written with some examples.

If the regular expression is inside a bracket "*[a-zA-z]*", this will match any character from "*a*" to "*z*" or "*A*" to "*Z* ", but it is also possible to set the opposite, that means "*[^abc]*" will match any character except "*a*", "*b*", "*c*".

Some of the special sequences beginning with "\" represent predefined sets of characters that are often useful, such as the set of digits, the set of letters, or the set of anything that is not whitespace. "*d*" matches any decimal digit; this is equivalent to the class "*[0-9]*", and also it has negation "*D*" it is equivalent to the class "*[^0-9]*". The user can find the whitespace characters with regular expressions, like space, newline, tabulator with the "*s*" expression. The "*w*" matches any alphanumerical character; it is equivalent to the class "*[a-zA-Z0-9_]*".

These sequences can be included inside a character class. For example, "*[\s,!]*" is a character class that will match any whitespace character, or "," or "*!*".

Dot "." is a metacharacter. It matches anything except a newline character. The first metacharacter for repeating things that we will look at is "*". The star character does not match the literal character "*"; instead, it specifies that the previous character can be matched zero or more times instead of exactly once.

For example (*Figure 3*), in this expression, "*da*ta*" will match "*dta*", it means zero "*a*" character, "*data*" (one "*a"* character), "*daaaata"* (four "*a*" character), and so on.

Find the date in a text could look like this formula: "*[0-9]{4}[^0-9]*[0-9]{2}[^0-9]*[0-9]{2}*"

This means "*[0-9]{4}*" four number which is the year, next "*[^0-9]**" zero or more non-numeric character (it is the separate character); the next step is "*[0-9]{2}*" it means two numbers, which is the month, and again the separate characters "*[^0-9]**" and in the end again "*[0-9]{2}*" the last two digits which are the days. Therefore, this regular expression's source text could be this "*2019. 10.31*" or "*2019-10-31*" or "*20191031*" as well [9].

```
star = re.compile("da*ta")
text = "dta, data, daaata"
print(star.findall(text))

date = re.compile("[0-9]{4}[^0-9]*[0-9]{2}[^0-9]*[0-9]{2}")
text = "2019.10.31 or 2019-10-31 or 20191031"
print(date.findall(text))
```

```
['dta', 'data', 'daaata']
['2019.10.31', '2019-10-31', '20191031']
```

Figure 3: Regular expression example

2.4 Syslog files

In this chapter, the Syslog files were presented. What a Syslog is and how it is built, and what it can be used for. This section helps understand the Syslog files (*Figure 4*).

In computer science, Syslogs are the Message Logging Standard by which almost any device or application can send data about events, status, diagnostics, and more [24]. The default programs to read these files are the notebook programs.

The log is the file extension for an automatically produced file that contains a record of events from specific software and operating systems. While they can include several things, log files are often used to show all events associated with the system or application that created them. For example, a backup program might keep log files showing exactly what happened or did not happen during a backup. The point of a log file is to keep track of what is happening behind the scenes, and if something should happen within a complex system, you have access to a detailed list of events that took place before the malfunction. Whatever the application, server, or Operating System (*O.S.*) thinks needs to be recorded [25].

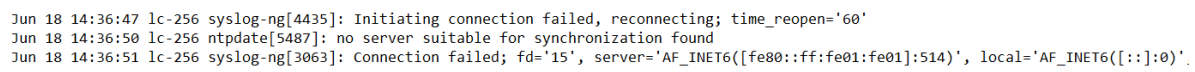

Figure 4: A System log file, Syslog

The Syslog Protocol (*RFC 5424*) describes [26] that a log file uses three layers:

- The "*Syslog transport*" layer puts messages on the wire and takes them off the wire.
- The "*Syslog application*" layer handles the generation, interpretation, routing, and storage of Syslog messages.
- The "*Syslog content*" is the management information contained in a Syslog message.

Syslog message size limits are dictated by the "*Syslog transport*" mapping in use, and there is no upper limit per se. Each transport mapping defines the minimum maximum required message length support.

In a Syslog file, the date is an important part. It could be four different formats.

1. format : "*1985-04-12T23:20:50.52Z"*

This example represents 20 minutes and 50.52 seconds after the 23rd hour of 12 April 1985 in UTC.

2. format: "*1985-04-12T19:20:50.52-04:00*"

The second format represents the same time as in the first example but is expressed in U.S. Eastern Standard Time.

3. format: "*2003-10-11T22:14:15.003Z*"

This date represents 11 October 2003 at 10:14:15pm, 3 milliseconds into the next second. The timestamp is in UTC.

4. format: "*2003-08-24T05:14:15.000003-07:00*"

The last date format represents 24 August 2003 at 05:14:15am, 3 microseconds into the next second. The timestamp indicates that its local time is -7 hours from UTC [26].

2.5 System log analyzer programs

At the beginning of this research several programs were overviewed to see whether they were capable of analyzing various log files. In this section these programs are explained, including my testing experiences.

Since a program can create many log files, it is worth organizing and analyzing them somehow; there are different programs for these; a couple of them were presented in this section.

Most system log file analysis programs cannot be used in this case. The log file management means how many files there are, at what intervals they were created, and whether they contain any anomaly. Often these programs are used to extract the message from many other unreadable words and numbers. Some programs can categorize log files somehow, but you cannot attach other existing data to this program; only those files can handle that are created on that computer.

Some analyzer program works only in the Ubuntu environment, like "*logpai*". Others are available from browsers, like "*LOGalyze*" (*Figure 5*).

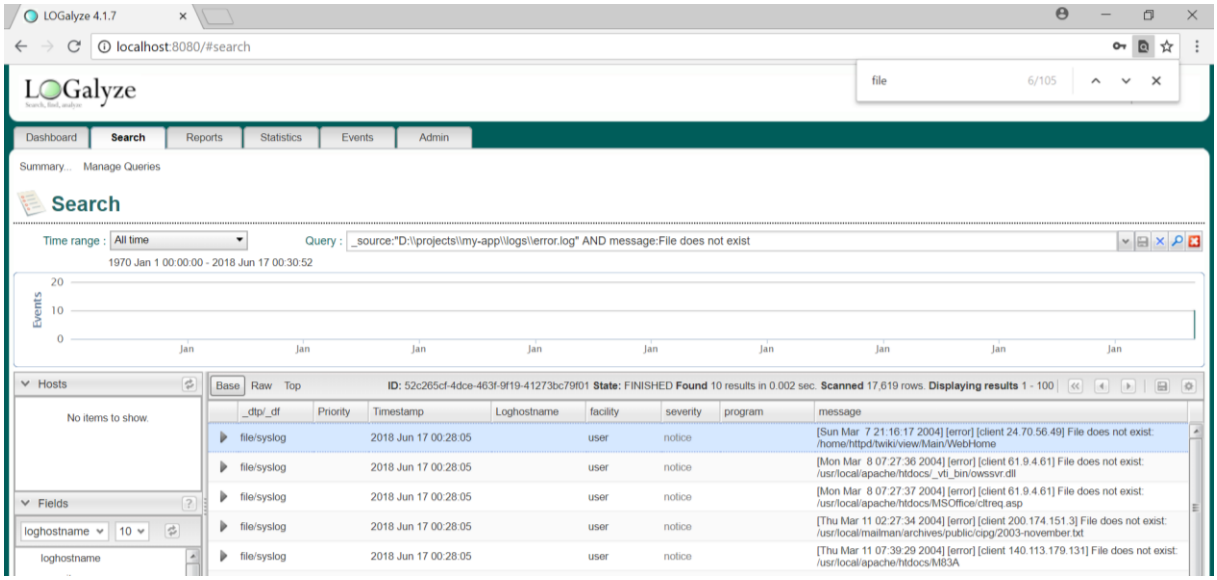

Figure 5: Screenshot of the LOGanalyze program, which is a log analyzer program

For a log analysis program, it is essential to read and understand the data, the text. The program then split the log files into different aspects, such as time, log type, message, and facility. Based on these, the log analyzer programs can summarize and create charts based on the processed data. For example, when an anomaly is detected, the program looks when the log files were created more often.

2.6 Raspberry Pi

In this thesis, in part Raspberry Pi was used, so in this chapter it was intruduced.

The Raspberry Pi is a credit-card-sized single-board computer that plugs into a computer monitor or television and uses a standard keyboard and mouse. A capable device enables people to explore computing and learn how to program in a language like Python. It can do everything people would expect a desktop computer to do, from browsing the internet and playing high-definition videos to making spreadsheets, word-processing, and even playing video games [27].

The first Raspberry Pi product was released in 2012, and the latest, the Raspberry Pi 4 Model B, was introduced in June 2019. It contains Gigabit Ethernet, along with onboard wireless networking and Bluetooth. It can handle the 4K output and run two monitors at once [28].

Figure 6: Raspberry Pi 4 Model B[1]. General view of the single computer board

As we can see above (*Figure 6*), the graphical interface of the Raspberry Pi device is the same as the standard desktop computer interfaces.

[1] *Source: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/ (2021. April)*

2.7 Tools for implementation

Python was used in this research because Python is a general-purpose, high-level programming language, which is very useful for data science, and deep learning tasks. It has many downloadable packages, modules, or machine learning, and many people use it in data mining and text mining.

For creating tables and working with data frames, the "*pandas*" package was used, and for the regular expressions, the "*re*" module. The "*os*" and "*glob*" modules were used to manage folders and files. Natural Language Toolkit ("*nltk*") was used to tokenize the words. This module is a very useful module for tokenization, stemming, text processing, and work with human language. For the calculations, evaluation "*math*", Scikit-learn ("*sklearn*"), and "*NumPy*" were used. Sklearn is also useful for separating training and test datasets, encoding labels, and create different models, such as Random Forest or Decision Tree. The charts and figures were created with the help of the "*matplotlib*" module. The "*pickle*" package was used to save and load the models.

"*Keras*" and "*TensorFlow*" were used for creating the neural network. It is also an open-source neural network library, which was written in Python, as well. Keras contains a significant number of implementations of commonly used neural network building blocks like layers, activation functions, and so on. It also supports convolutional and Recurrent Neural Networks. Keras module is also useful to help the text preprocessing.

TensorFlow is a helpful tool for neural networks; it is open-source software. This module is fast and flexible so that it can be used for many different problems. TensorFlow is a symbolic math library, and for example, at Google, the engineers use this module for research and productivity. It is prevalent, and most of the time, it performs very well [29].

2.8 Evaluation with metrics

For the models, the accuracy of the given model was calculated as well as the value of the area under the receiver operating characteristic curve. The formula for accuracy is (*Equation 4*):

$$accuracy = \frac{correctly\ predicted\ class}{total\ testing\ class} * 100$$

Equation 4: The formula for the models' accuracy

On the other hand, the other formula is (*Equation 5*):

$$accuracy = \frac{True\ positive\ +\ True\ negative}{True\ positive + False\ positive + True\ negative + False\ negative}$$

Equation 5: The formula for the models' accuracy rearranged

[30].

For the formula for the Area Under the receiver-operating characteristic Curve (*AUC*) score, we have to calculate the True Positive Rate (*TPR*), and the False Positive Rate (*FPR*), which are the following equations (*Equation 6*):

$$TPR = \frac{True\ positives}{All\ positives}; \qquad FPR = \frac{False\ positives}{All\ negatives}$$

Equation 6: The formula for the models' AUC score

The value of AUC is the area under the line on the diagram where the x-axis is the False Positive Rate, and the y-axis is the True Positive Rate [31].

The AUC is equal to the probability that the classifier will rank a randomly chosen positive example higher than a randomly chosen negative example. There are some important basic terms for this; the True Positive Rate is also known as Recall or Sensitivity, and the False Positive Rate or Specificity. They both have values in the range [0, 1] close intervals, and they are computed at the threshold values. After a graph is drawn, the AUC is the area under the curve between them. Other good evaluation numbers are Precision, Recall, and F1 score [32] (*Equation 7*).

$$Precision = \frac{True\ positives}{True\ positives + False\ positives}$$

$$Recall = \frac{True\ positives}{True\ positives + False\ negatives}$$

$$F1\ score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Equation 7: The formulas for Precision, Recall, and F1 scores

The Confusion Matrix is a matrix; it describes the performance of the model. Actual values can be Positive or Negative. For example, in a binary classification, the first label is positive, and the second is negative. When we predicted the positive and negative value correctly, then it is the True Positive (*TP*), which means that we predicted the first label, and the first label happened to the system. The True Negative (*TN*) means that we predicted the second and the second label happened. However, suppose our predicted positive value is a true negative, and the predicted negative is actually positive. In that case, they are False Positive (*FP*), where we predicted the first label, but the system had the second label and False Negative (*FN*), where we predicted the second, but it actually had the first (*Table 1*).

		Predicted label	
		Positive	**Negative**
True label	**Positive**	TP	FN
	Negative	FP	TN

Table 1: Confusion matrix

Cross-validation was used during this research. Cross-validation helps the models to give much more accurate results and avoid overfitting; the "*KFold()*"function from the "*sklearn*" module was used to create fifteen folds.

Cross-validation is a resampling procedure used to evaluate machine-learning models on a limited data sample. The procedure has a single parameter called '*k*', which refers to the number of groups to which a particular data sample should be divided [33].

To understand the cross-validation procedure, an example was written:

We have a dataset with six observations.

$$dataset = [x_1, x_2, x_3, x_4, x_5, x_6]$$

The first step is to select a k value to determine the number of folds used to split the data; let us say, in this case, $k = 3$. This means; first, we shuffle the data and split it into three different groups. Since we have six observations, each group will have the same number of observations. The result is the following folds:

- $Fold\_1 = [x_1, x_2]$
- $Fold\_2 = [x_3, x_4]$
- $Fold\_3 = [x_5, x_6]$

Three models are trained and evaluated, with each fold given a chance to be the held-out test set. The table below (*Table 2*) illustrates these splits; the fold with red text is the test set, and the others are the train set.

	Fold_1	***Fold_2***	***Fold_3***
Model_1	$[x_1, x_2]$	$[x_3, x_4]$	$[x_5, x_6]$
Model_2	$[x_1, x_2]$	$[x_3, x_4]$	$[x_5, x_6]$
Model_3	$[x_1, x_2]$	$[x_3, x_4]$	$[x_5, x_6]$

Table 2: Cross-validation example

3 Models and deep learning

In this chapter, first, decision Trees were written, and after, the XGBoost model, which is a gradient boosted decision trees.

3.1 Machine learning

Machine learning is a deep learning area that focused on constructing algorithms that predict based on data. A machine learning task aims to learn a function "*f: Data (X) → Possible predictions (Y)"* that maps the input domain of the data onto the output domain of possible predictions. The function "*f*" is selected from a specific function class, which is different for each family of learning algorithms. Elements of "*X*" and "*Y*" are application-specific representations of data objects and predictions, respectively [18].

Machine learning algorithms can be classified mainly into three different categories by the type of datasets used as experience. These categories are reinforcement learning, supervised learning, and unsupervised learning. Other learning algorithms combine two categories, like semi-supervised learning, which uses both unlabeled and labeled data. Reinforcement learnings do not experience a fixed dataset but a feedback loop between the system and its experiences. Supervised learnings use labeled datasets, where part of the dataset represents the data point, and the other part is the corresponding true prediction for the first part. This training set of input-output pairs is used to find a deterministic function that maps any input to output, predicting future input-output observations while minimizing errors as much as possible. Unsupervised learning algorithms use unlabeled datasets to train the model. The point of unsupervised learning is to derive structure from unlabeled data by investigating the similarity between pairs of objects and is usually associated with density estimation or data clustering [19]. A possible machine learning method can minimize the mean squared error (*MSE*) on the training set *X*:

$$MSE = \frac{1}{n}\sum_{i}(\hat{y} - y)^2$$

Equation 8: Mean Squared error

In this equation (*Equation 8*), the datasets *X* has *n* instances *i* (the events). $\hat{y}$ is the prediction of the model.

A good, successful machine-learning algorithm should perform well on unseen input samples of the same type as the training and validation datasets. Therefore, the algorithm's performance should be evaluated on its ability to minimize the training error and its ability to reduce the difference between the training error and validation error.

3.2 Decision Trees

Decision Tree Learning is a non-parametric supervised learning method. It can be used for both classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. The deeper the tree, the more complex the decision rules and fitter the model.

The Decision tree method is useful in data science because it is simple to understand and easy to visualize. It requires little data preparation because, for example, this module does not support missing values; other models often require data normalization or need to create dummy variables and blank values to be removed. The cost of predicting data is logarithmic in the number of data points used to train the tree. The Decision tree model can handle both categorical and numerical data, unlike other models, which are usually specialized in analyzing datasets with only one variable type. This method uses a white-box model. If a given situation is observable in a model, the explanation for the condition is easily explained by Boolean logic. By contrast, in a black-box model (for example, in an artificial neural network), results may be more challenging to interpret. Decision trees perform well even if their assumptions are somewhat violated by the correct model from which the data were generated. With this model, it is possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.

Unfortunately, Decision trees have some disadvantages. Decision-tree learners can create over-complex trees that do not generalize the data well. If this happens it is called overfitting. Mechanisms such as pruning, setting the minimum number of samples required at a leaf node, or setting the maximum depth of the tree are necessary to avoid this problem. Decision trees can be unstable because small variations in the data might result in a completely different tree. This problem is reduced by using decision trees within an ensemble. Practical decision-tree learning algorithms are based on heuristic algorithms, like the greedy algorithm, where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. It can be reduced by training multiple trees in an ensemble learner, where the models are sample the features and samples randomly with replacement.

Decision tree learners create biased trees if some classes dominate, therefore recommended to balance the dataset before fitting with the decision tree. Some concepts are hard to learn because decision trees do not express them easily, like the multiplexer problems [12].

Below we can see a short example to understand the decision trees better.

The following example shows how a decision tree learning algorithm can obtaine a decision tree that is designed to decide whether we can joyfully play tennis or not (*Figure 7*).

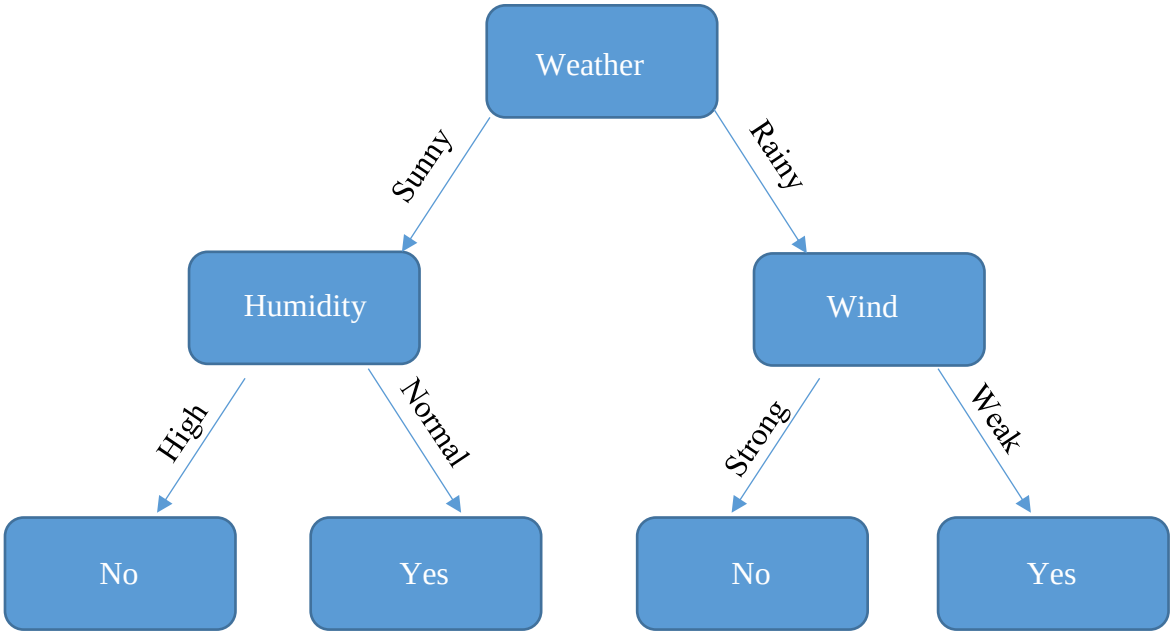

Figure 7: A decision tree for classifying whether to play tennis or not. Each box represents a condition on an attribute whose value takes you down to the left of right. Box at the leaves represent the final decision (the class)

3.3 XGBoost

The other model used in this research is XGBoost; it stands for eXtreme Gradient Boosting [13].

XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, portable, and flexible. It implements machine learning algorithms under the Gradient Boosting framework. This model provides a parallel tree boosting that solves many data science problems quickly and accurately. The same code runs on a significant distributed environment and can solve problems beyond billions of examples [14].

XGBoost is an algorithm that has recently been dominating applied machine learning for structured or tabular data. It is an implementation of gradient boosted decision trees designed for speed and performance. This model supports many interfaces, like C++, Python, R, Julia,

Command Line Interface, and Java. The implementation of the model supports the features of the scikit-learn (Python) and R , with new additions like regularization. The model support three types of gradient boosting. The first is the Gradient Boosting algorithm; it is also called the Gradient Boosting machine, including the learning rate. The second is the Stochastic Gradient Boosting with sub-sampling at the row, column, and column per split levels. The third, and the last one, is the Regularized Gradient Boosting with both Lasso Regression and Ridge Regression regularization. The XGBoost library provides a system for use in a range of computing environments, for example, the parallelization of tree construction using all of your CPU cores during training. Distributed computing is useful for training huge models utilizing a cluster of machines. Out-of-core computing helps for enormous datasets that do not fit into memory. Cache optimization of data structures and algorithm to make the best use of hardware. The implementation of the XGBoost algorithm was engineered for the efficiency of memory resources and compute time. A design goal was to make the best use of available resources to train the model. The figure below (*Figure 8*) shows that XGBoost was usually faster than the other benchmarked implementations from R, Python Spark, and H2O.[13].

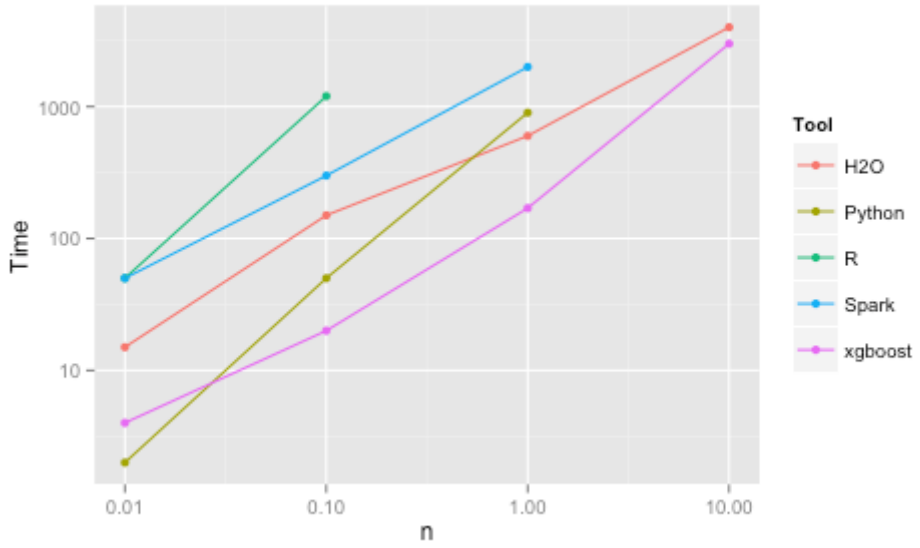

Figure 8: Benchmark Performance of XGBoost [2] and other programing laguages

Representing input data using sparsity in this way has implications on how to calculate the splits. The XGBoost model's default method of handling missing data when learning decision tree splits is to find the best 'missing direction' in addition to the standard threshold decision rule for numerical values. The one-hot encoded categorical variable where the zeros are

[2] *Source: http://datascience.la/benchmarking-random-forest-implementations/ (2020. September)*

encoded as missing values is equivalent to testing one versus all splits for each category. To select the missing direction is the direction that maximizes the gain from equality. When enumerating through all possible split values, we can also test the effect on our gain function of sending all missing examples down the left or right branch and select the best option. This step makes split selection slightly more complicated, since we do not know the gradient statistics of the missing values for any given feature we are working on, although we do know the sum of all the gradient statistics for the current node [15]. The XGBoost algorithm can handle this by performing two reads over the input data, and the second reverse. For the first left-to-right scan, the left path gradient statistic is the scan value maintained by the scan, the corresponding direction gradient statistic for this node is the total gradient statistic minus the scan values. Therefore, the correct direction implicitly contains all the missing values. When scanning from right to left, the reverse direction is correct and the left direction contains all missing values. The algorithm then selects the best split from the forward or reverse scan [15].

3.3.1 XGBoost Trees for Regression

In *Figure 9*, we can see how effective a drug is depending on the dosage. The second and the third observations are relatively large positive values for Drug Effectiveness, which means that the drug was helpful. The first and the fourth are relatively large negative values, so that means the drug did more harm than good. The red line is the initial prediction.

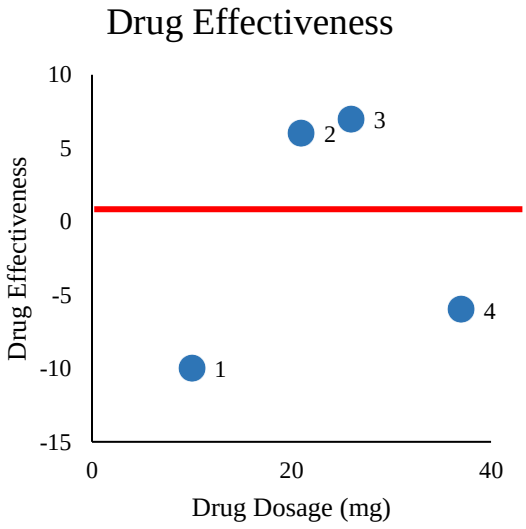

Figure 9: XGBoost Regression example. This figure shows the relation between the drug dosage and the drug effectiveness

The first step in fitting XGBoost to the training data is to create an initial prediction. This value can be anything, but by default, it is 0.5, regardless of whether you are using the model

for Regression or Classification. The predictions, 0.5, corresponds to this thick, red, horizontal line, and the residuals, the differences between the observed and predicted values, show us how good the initial prediction is. After the XGBoost fits a unique regression tree to the residuals. There are many ways to build trees like this; below the most common way was presented to build them for regression.

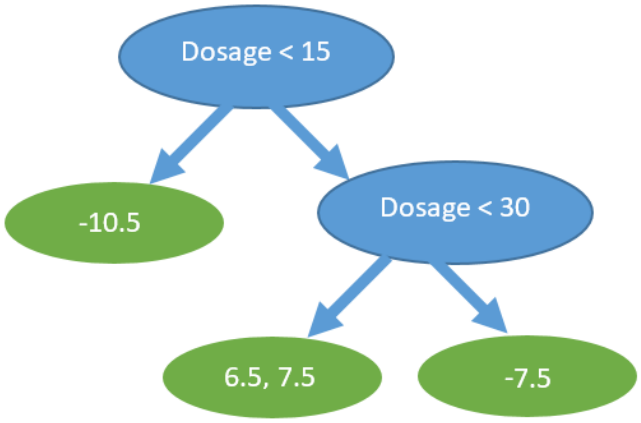

Figure 10: XGboost Regression example. The figure shows the splits based on the drug dosage

Each tree starts as a single leaf. All of the residuals go to the leaf after calculating a quality score or similarity score for these points, where the lambda is a regularization parameter (*Equation 9*).

$$Similarity\ Score = \frac{(\sum Residuals)^2}{Number\ of\ Residuals + \lambda}$$

Equation 9: Similarity Score for Regression

[16] so, in this case, if the lambda is zero, which is the default number, then the similarity score is:

$$Similarity\ Score = \frac{(-10.5 + 6.5 + 7.5 - 7.5)^2}{4 + 0} = 4$$

The question is whether we can do a better job clustering similar residuals if we split them into two groups. Therefore first focus on the two observations with the lowest dosages. Their average dosage is 15, so we split the points into two groups, based on whether or not the dosage less than 15. The first point on the far left is the only one that is smaller, so its residual goes to the leaf on the left, the others go to the leaf on the right. Now the model calculates the similarity score again for the left and the right as well, which are:

$$Similarity\ Score\ on\ left = \frac{-10.5^2}{1+0} = 110.25$$

$$Similarity\ Score\ on\ right = \frac{(6.5 + 7.5 - 7.5)^2}{3+0} = 14.08$$

We can see that when the residuals in a node are very different, they cancel each other out, and the similarity score is relatively small (*4*). In contrast, when the points are similar, or there is just one of them, they do not cancel out, and the similarity score is relatively large (110.25). After the model needs to quantify how much better the leaves cluster similar residuals than the root by calculating the gain of splitting the residuals into two groups (*Equation 10*).

$$Gain = Left_{Similarity} + Right_{Similarity} - Root_{Similarity}$$

Equation 10: Gain score for regression

[17] In this case is the Gain is $110.25 + 14.08 - 4 = 120.33$. After the threshold's gain calculation, XGBoost can compare it to the gain calculated for the other thresholds. Therefore, it shifts the threshold over so that it is the average of the next to observations and builds a simple tree that divides the observations using the new threshold, dosage less than 22.5. Calculate the similarity score for the leaves and calculate the gain again. This gain is 4. Since this gain is less than the gain for dosage less than 15 (120.33), therefore, dosage < 15 is better at splitting the residuals into clusters of similar values. So again, it shifts the threshold over so that it is the average of the last two observations and builds again a simple tree that divides the observations using the new threshold, dosage less than 30, and calculates the similarity scores and the gain. At this time, the gain is 56.33, which is smaller than 120.33. Since the model cannot shift the threshold over any further to the right, it is done comparing different thresholds, using the threshold that gave the largest gain for the first branch in the tree. Since there is only one point in the leaf on the left, it cannot split it any further. Therefore, it can split the 3 points in the leaf on the right, and do the process from the beginning, calculate the average for the new first two residuals, and try the thresholds by calculate the similarity scores and the gains, and select the highest gain threshold, this time the gain is 140.17. As we can see on the chart, the tree depth of the levels has limit, and this means the model does not split the 6.5, 7.5 leaves any further, and it is done building this tree, but the default is to allow up to 6 levels. XGBoost prunes this tree based on its gain values. It starts by picking a number, for example, 130. XGBoost calls this number γ $(gamma)$. Than it starts to calculate the difference between the gain associated with the lowest branch in the tree and the value for gamma. If the difference between the gain

and gamma is negative, it will remove the branch, but if it is positive, we will not. In this case, $140.17 - 130 = 10.17$ is a positive number so that the branch will stay, and the model is done pruning. We can see the gain for the root is 120.33, and it is less than 130, the value for gamma so that the difference will be negative. However, because the model did not remove the first branch, it will not remove the root. In contrast, if we set $\gamma = 150$, than it would remove the branch, therefore, it will remove the root too, and all it would be left with is the original prediction, which is extreme pruning.

If we set the regularization parameter, $\lambda\ (lambda)\ to\ 1$, which means it is intended to reduce the prediction's sensitivity to individual observations, then the new similarity score for the root is 3.2, which is $\frac{8}{10}s$ of what we got when $\lambda = 0$. When it calculates the similarity score for the leaf on the left, it gets 55.12, which is half of before. We can see is that when $\lambda > 0$, the similarity scores are smaller, and the amount of decrease is inversely proportional to the number of residuals in the node. In other words, the leaf on the left had only 1 point, and it had the largest decrease in similarity score, 50%; in contrast, the root had all 4 points and the smallest decrease, 20%. Therefore the gain values will be smaller, so if we use the same $\gamma\ (gamma)$ values (130), we will prune the whole tree away. When the $\lambda > 0$, it is easier to prune leaves because the values for gain are smaller. Setting the $\gamma = 0$, does not turn off pruning because the gain can be a negative number, and negative – 0 = negative so that the model will remove the branch. On the other hand, by setting $\lambda = 1$, λ prevented overfitting the training data. Now the model will calculate the output values for the leaves (*Equation 11*); it is like the similarity score, except it does not square the sum of the residuals.

$$Output\ Value = \frac{\sum Residuals}{Number\ of\ Residuals + \lambda}$$

Equation 11: Output Value for Regression

[16] So the output value to the leaf on the left is $\frac{-10.5}{1+\lambda}$, if the $\lambda = 0$, the value is -10.5; if it is 1, then -5.25. In other words, when $\lambda > 0$, it will reduce the amount that this individual observation adds to the overall prediction and reduce the prediction's sensitivity to this individual observation. When $\lambda = 0$, then the output value for a leaf is simply the average of the residuals in that leaf. After the calculation of the output values, the model can create new predictions. Like an unextreme gradient boost, XGBoost creates new predictions by starting with the initial prediction and adding the output of the tree scaled by a learning rate. XGBoost calls the learning rate, $\varepsilon\ (eta)$, and the default value is 0.3 (*Equation 12*).

$$New\ prediction = original\ prediction + (Learning\ rate * Output\ value)$$

Equation 12: How the regression's predictions work

[17] Therefore the new predicted value for the first observation, with 10 dosages, is the original prediction, 0.5 + (0.3 * -10.5) = -2.65. For the second residual the new predicted values is 0.5 + (0.3 * 7) = 2.6, and so on. It builds another tree based on the new residuals and creates new predictions that give even smaller residuals, and it keeps building trees until the residuals are super small or it has reached the maximum number.

In summary, when building XGBoost Trees for Regression, the model calculates the similarity scores and the gain to determine how to split the data, and it prunes the tree by calculating the differences between gain values and a user-defined tree complexity parameter, $\gamma\ (gamma)$ (*Equation 13*).

$$Gain - \gamma = \begin{cases} > 0, than\ do\ not\ prune \\ < 0, than\ it\ will\ prune \end{cases}$$

Equation 13: Pruning for Regression

If the model prunes, it will subtract γ from the next gain value and work the way up the tree. Then it calculates the output values for the remaining leaves. When $\lambda\ (regularization\ parameter) > 0$, it results in more pruning by shrinking the similarity scores, and it results in smaller output values for the leaves.

3.3.2 XGBoost Trees for Classification

To present this, almost the same training set was used as before. The second and the third observation were effective, and the first and fourth was not. As we can see above, the first step in fitting XGBoost to the training data is always to create an initial prediction. As it was mentioned, this prediction can be anything, for example, the probability of observing an effective dosage in the training data, but by default, it is 0.5 (*Figure 11*).

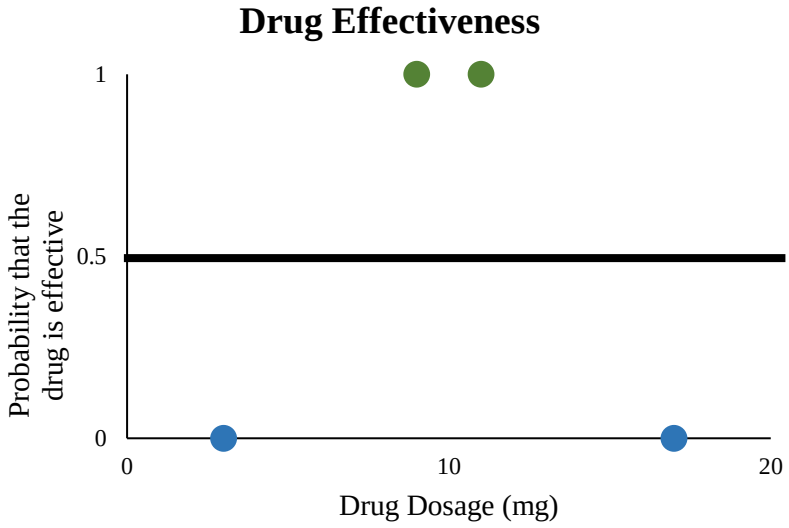

Figure 11: XGBoost Classification example. This figure shows the relation between the drug dosage and the probability that the drug is effective

Since the second and the third green dots represent effective dosages, the probability that the drug is effective is one. The others represent ineffective dosages, so the probability for them is zero. The residuals, the differences between the observed and predicted values, show us how good the initial prediction is. As it was explained above, a unique tree was fitted to the residuals; the XGBoost model for classification has a new formula for the similarity scores (*Equation 14*).

$$Similarity\ Scores\ for\ classification = \frac{(\sum Residual_i)^2}{\sum[Previous\ Probability_i * (1 - Previous\ Probability_i)] + \lambda}$$

Equation 14: Similarity Scores for Classification

The numerator for classification is the same as the numerator for regression. As with regression, the denominator contains λ $(lambda)$, the regularization parameter; however, the rest of the denominator is different. The sum for each observation of the predicted probability times one minus the previously predicted probability.

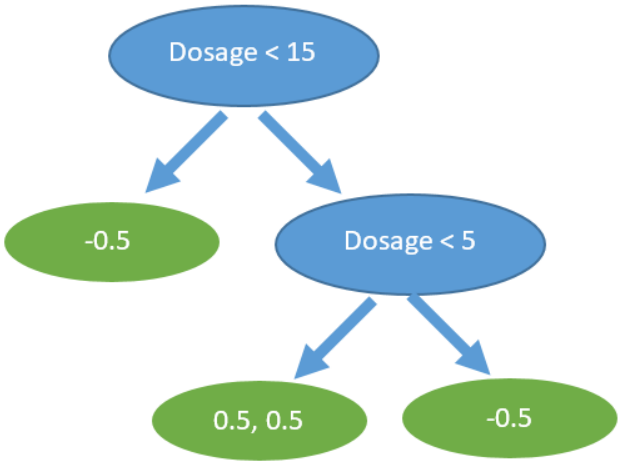

Figure 12: XGBoost Classification example. The figure shows the splits based on the drug dosage

Although this formula is different from what XGBoost uses for regression, it is very closely related, and I will show this later. As for the regression, each tree starts as a single leaf, and all of the residuals go to the leaf (*Figure 12*). The model calculates the similarity score. This time means it is zero because the nominator will be 0:

$$Similarity\ score = \frac{(-0.5 + 0.5 + 0.5 - 0.5)^2}{\sum[Previous\ Probability_i * (1 - Previous\ Probability_i)] + \lambda} = 0$$

So again, it needs to decide if it can do a better job clustering similar residuals if it split them into two groups. For example, I will start with the threshold, where the dosage is less than 15 (this number is the average value between the last two observations). The similarity score for the leaf on the left is $\frac{(-0.5)^2}{0.5*(1-0.5)+\lambda} = 1, when\ \lambda = 0.$ For the right, it is $\frac{(-0.5+0.5+0.5)^2}{(0.5*(1-0.5))+(0.5*(1-0.5))+(0.5*(1-0.5))+\lambda} = 0.33, when\ \lambda = 0.$ Now it can calculate the gain, just as it did when I used XGBoost for regression. Therefore, when it split the observations based on the threshold dosage < 15, the gain is 0.33 + 1 – 0 = 1.33, and because no other threshold gives a larger gain value, this will be the first branch in the tree. The model will split the rest of the residuals into two leaves and start again from the beginning. As we can see above, the model stopped growing this tree because the number of levels has limit, and this number is two; however, XGBoost also has a threshold for the minimum number of residuals in each leaf. This number is determined by calculating cover. The cover is defined as the denominator of the similarity score minus λ. In other words, during classification, the cover is equal to (Equation 15):

$$Cover = \sum [Previous\ Probability_i * (1 - \ Previous\ Probability_i)]$$

Equation 15: Cover for Classification

In contrast, during the regression, the cover is equal to the number of residuals. By default, the minimum value for cover is one. When we use XGBoost for regression and use the default minimum value for cover, the cover does not affect how the tree grows because we can have as few as one residual per leaf. In contrast, classification is more complicated because cover depends on the previously predicted probability of each residual in a leaf. For example, the cover value for the -0.5 leaf is 0.5 * (1 – 0.5) = 0.25, and since the default value for the minimum cover is one, the model would not allow this leaf. Likewise, the cover for the 0.5, 0.5 leaf is 0.5 * (1 – 0.5) + 0.5 * (1-0.5) = 0.5, so by default the model would not allow this leaf either, and so on. XGBoost requires trees to be larger than just the root, so we have to set the cover value to zero. That means in the Python setting the min_child_weight parameter equal to zero. The trees are pruned in the same way as in the first case; lambda and gamma work the same way. For classification, the output value for a leaf is (*Equation 16*):

$$Output\ Value = \frac{(\sum Residual_i)}{\sum [Previous\ Probability_i * (1 - Previous\ Probability_i)] + \lambda}$$

Equation 16: Output Value for Classification

However, just like with an unextreme gradient boost for classification, the model needs to convert the initial probability to the value of the logarithm of the odds (Equation 17).

$$odds = \frac{probability}{1 - probability} \rightarrow \log(odds) = \log(\frac{probability}{1 - probability})$$

Equation 17: Odds for Classification

In this case, when the initial prediction is 0.5, $\log\left(\frac{0.5}{1-0.5}\right) = 0$. As mentioned before, like gradient boost for classification, we add the log(odds) of the initial prediction to the output of the tree, scaled by a learning rate, ε (eta). The default value is 0.3, so same in regression. Therefore, the new predicted value for the first observation with two dosage is 0 + (0.3 * -2) = -0.6. To convert a log(odds) value into a probability, the model plugs into the logistic function, which is (*Equation 18*):

$$Probability = \frac{e^{\log(odds)}}{1 + e^{\log(odds)}}$$

Equation 18: Probability for Classification

In this case, the new predicted probability is $\frac{e^{-0.6}}{1+e^{-0.6}} = 0.35$, for the second point, the new value is 0.65. After this, the model will calculate the whole process to the next leaves and build a new tree with the new residuals. After the first tree, when the model builds the tree, calculating the similarity scores is more interesting because the previous probabilities are no longer the same for all observations. The model will keep building trees until the residuals are very small or reached the maximum number of trees.

In summary, when we use XGBoost for classification, it calculates similarity scores and gains to determine how to split the data. It prunes the tree by calculating the difference between gain values and a user-defined gamma parameter like with the regression. Then it calculates the output values for the leaves. We have to be aware that the minimum number of residuals in a leaf is related to a metric called “cover”, which is the denominator of the similarity score minus lambda.

3.4 Deep learning

After the decision tree, and XGBoost models, in this chapter, neural networks is written because neural networks were used during the thesis.

3.4.1 Artificial Neural network

A neural network is a network of simple components and hierarchically organized parallel structures that deal with the realities of the world in the same way as the biological nervous system. An artificial neural network is an information processing system composed of the structure and function of the physiological human brain neural network and some theoretical abstraction, simplification, and simulation of several basic characteristics. The goal of artificial neural networks is not to model the human brain. Artificial neural network research is fundamentally based on and driven by mathematical and engineering disciplines rather than biological brain function.

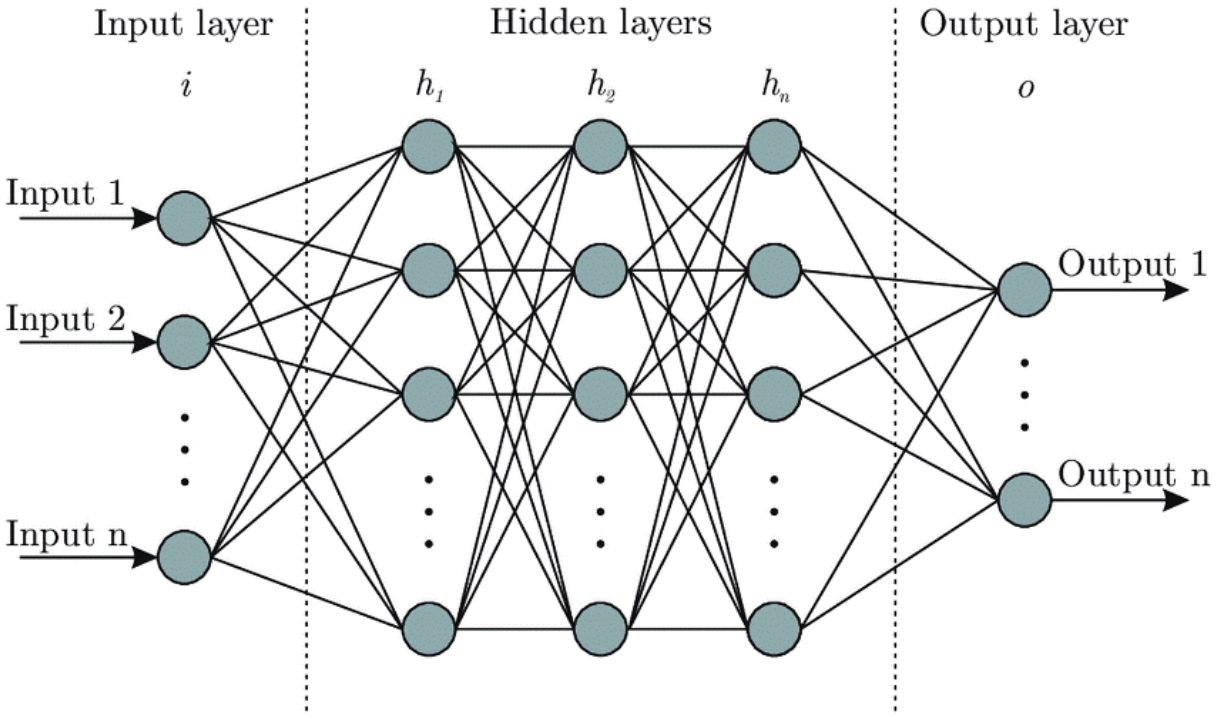

Figure 13: Layers of the Artificial Neural Network[3]. The first layer is the input layer, the last is the output layer, between them are the hidden layers. Every layer could have more nodes on it.

The structural model of a neuron can be represented as a node. Each input weight can be adjusted during the learning process. Multiple such points are connected into a directed graph

[3] *Source: https://www.researchgate.net/figure/Artificial-neural-network-architecture-ANN-i-h-1-h-2-h-n-o_fig1_321259051 (2020. September)*

to form an adaptive neural network. The information processing of artificial neurons can be divided into three parts. In the first part, the inner product of the input signal and the strength of the neuron connection is completed, and then the result is passed through the activation function and then judged on the basis of the threshold function [34]. If it is higher than the threshold, the nerve is activated; otherwise they will be suppressed. In this way, artificial neurons are very similar to biological neurons. In general, the neural network model is determined by three factors: network topology, neuronal characteristics, and learning or training rules. The topology of neural networks can be divided into hierarchical network models and interconnected network models (*Figure 13*). The former divides neurons into several layers, and the layers are connected one after the other. The layer input is only associated with the layer output. The latter allows any two neurons to be connected. Some networks are a mixture of the two.

The default choice for an activation function in artificial neural networks is the max function (*Equation 19*):

$$g(z) = max\{0, z\}.$$

Equation 19: The Neural networks' max funcion

A unit that employs this function is called a rectified linear unit or, in shorter version ReLU. ReLUs are quickly optimized since the derivative is either zero or a positive constant value through the domain. Therefore, the gradient direction is far more useful for learning than it would be with activation functions with non-vanishing and higher-order derivatives. Unfortunately, ReLU has some drawbacks; one of them is that they cannot learn via gradient-based methods on examples for which the activation is zero.

The other activation function is the sigmoid, and this function is used to represent a probability distribution over a binary variable. It defines as (*Equation 20*):

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Equation 20: The Neural networks' sigmoid function

The softmax function of "z", is a generalization of the sigmoid function representing a probability distribution over a discrete variable with n possible values. Softmax functions are often used as the output units of a classifier [20].

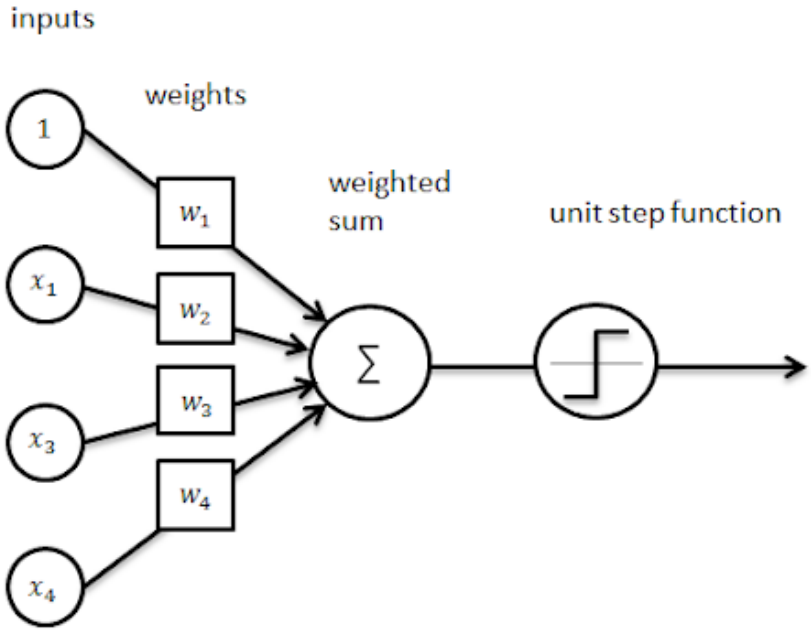

Figure 14: Single Perceptron[4]. After the multiplication and summarization the last step is the unit function.

Perceptron is an essential element in a neural network. We can see the figure above about a single perceptron (*Figure 14*).

A perceptron is taking in the different inputs, processes them, and then generates an output. Inputs have different weights because some inputs have more importance than others do. I would like to write an example for representing how it works. It is a simple question: Should I go out to grill chicken wings in the park? I have more inputs, which decides this question,"x_1" (Is the weather good? 1: means yes, 0: means no), and"x_2" (Is it a weekend? 1: yes, 0: no),"x_3" (Is there a crowd in the park? 0: yes, 1: no), and"x_4" (Do I have chicken wings? 1: yes, 0: no). There are four weights for these; their value is 1 for "w_1", 2 for "w_2", 1 for "w_3" and 2 for "w_4". When the weather is excellent, it is Saturday, nobody is in the park, and I have wings; the equation would be (*Equation 21*):

$$\sum_{i=1}^{4}(x_i * w_i) = x_1 * w_1 + x_2 * w_2 + x_3 * w_3 + x_4 * w_4$$

Equation 21: Simple equation for Neural networks

[4] *Source: https://towardsdatascience.com/introduction-to-artificial-neural-networks-ann-1aea15775ef9 (2020. October)*

In this case, $1 + 2 + 1 + 2 = 6$, so I will go to the park to grill. We need to define the threshold. In this example, if the limit is six, that means, if the total is less than six, then we will not go to the park, but if it is equal or greater than six, then we will. I gave the "x_2", "x_4" a higher weight, because on the weekdays I usually go to work, and if I do not have wings then I cannot grill, so "x_2", "x_4" are more important than "x_1", "x_3". The output was binary as well. A single perceptron cannot be used for tasks that are more complex because if the job is to recognize digits from a picture, the output could only have two values so that it would not work. A neuron is similar to a perceptron, but instead of zero or one for input, it can take fraction values as valid input. The threshold was defined before, and it looked like this (*Equation 22*).

$$output = \begin{cases} 0 \; if \; \sum_i w_i * x_i < threshold \\ 1 \; if \; \sum_i w_i * x_i \geq threshold \end{cases}$$

Equation 22: The Neural networks output equation

Now I will move the threshold to the other side of the inequality. We can use a threshold on the other side too, but because of bias = - threshold, the inequality will be this, but in reality, the outcomes of neural networks are not so discrete but continuous (*Equation 23*):

$$output = \begin{cases} 0 \; if \; \sum_i w_i * x_i + bias < 0 \\ 1 \; if \; \sum_i w_i * x_i + bias \geq 0 \end{cases}$$

Equation 23: The Neural network output equation with bias

Usually, in the real world, the problems are more complex, so we cannot solve them with only one perceptron. Typically, we create a complex neural network with more layers [21].

Learning in artificial neural networks is closely related to how humans learn in our regular lives; we perform an action, and we are either accepted or corrected by a coach to understand how to get better at a specific task. Neural networks require a leader in order to describe what should have been produced as a response to the input. Based on the difference between the actual value and the predicted value, an error value, also called the "*cost function*", is computed and sent back through the system. Cost Function is a value, one-half of the squared difference between actual and output value. For each layer of the network, the cost function is analyzed and used to adjust the threshold and weights for the following input. The network aims to minimize this cost. The lower the cost function, the closer the actual value to the predicted value. In this way, the error keeps becoming marginally lesser in each run as the network learns how to analyze values. The system feeds the resulting data back through the entire neural

network. The weighted synapses connecting input variables to the neuron are the only thing we can control. As long as there exists a disparity between the actual value and the predicted value, the system needs to adjust those weights. We have to repeat this process until we scrub the cost function down to as small as possible [21].

4 Data description and preparation

In this section, information about the data, the preparation and processing was written.

4.1 Data

Only the system log files were used in the analysis. There were 837 files on 128 individual cards, which number was increased later to 3203 different sliced log files thanks to a slicing method, which is explained in the "*Data preparation*" section.

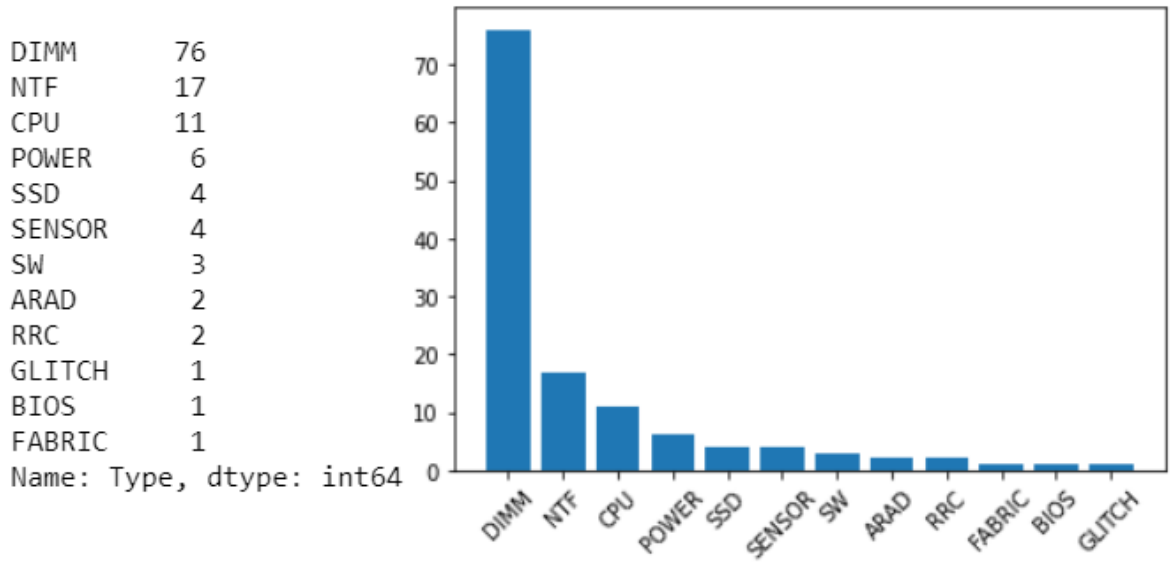

Figure 15: The card's label distribution. Some of the labels are raleted to the memory, some of them are power or sensor related

The Syslogs were of various lengths; the shortest had two lines, which means only 77 bytes, and the longest system log file had almost 14.000.000 lines, which means more than 2 gigabytes. Due to the imbalance in the lengths of the files, later it was important to pay attention to the method of analysis because they could lead to false results.

As we can see in the figure above (***Error! Reference source not found.***), there was no balance in the dataset at the beginning of the research. The dataset contained twelve labels, the most common label was the "*DIMM*" label; as shown in the figure, there were 17 cards that received the "*NTF*" label, and "*NTF*" corresponds to "*No Trouble Found*", which means the good log files. Some labels were similar to each other and could be grouped, but each was a unique label.

4.2 Tokenization and card level

The system log files are not the same length; it could have two rows, but they could also have more than 400.000 rows (*Figure 16*), so these files were normalized. In the beginning, many regular expressions were used to find and replaced parts of the text. First, a single log file was checked how it looks, for example, the date format. Some functions were written to replace the words: a few were simple, and one of them was a harder and more complex function.

```
Sep 22 05:59:19 lc-X syslog-ng[4408]: Internal error, duplicate configuration elements refer to the same pers
Sep 22 05:59:19 lc-X syslog-ng[4408]: Syslog connection established; fd='18', server='AF_INET6([fe80::ff:fe01
Sep 22 05:59:19 lc-X syslog-ng[4408]: Configuration reload request received, reloading configuration;
Sep 22 06:01:57 lc-X kernel: [    0.000000] Initializing cgroup subsys cpuset
Sep 22 06:01:57 lc-X kernel: [    0.000000] Initializing cgroup subsys cpu
Sep 22 06:01:57 lc-X kernel: [    0.000000] Initializing cgroup subsys cpuacct
Sep 22 06:01:57 lc-X kernel: [    0.000000] Linux version 3.14.25-00270-ga5c01ed (xxxxx@xxxxxx1111.mo.ca.am.(
Sep 22 06:01:57 lc-X kernel: [    0.000000] Command line: bzImage.efi earlyprintk=serial,ttyS0,115200n8 consc
Sep 22 06:01:57 lc-X kernel: [    0.000000] e820: BIOS-provided physical RAM map:
Sep 22 06:01:57 lc-X kernel: [    0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009ffff] usable
Sep 22 06:01:57 lc-X kernel: [    0.000000] BIOS-e820: [mem 0x0000000000100000-0x00000000784dffff] usable
Sep 22 06:01:57 lc-X kernel: [    0.000000] BIOS-e820: [mem 0x00000000784e0000-0x00000000796a2fff] reserved
Sep 22 06:01:57 lc-X kernel: [    0.000000] BIOS-e820: [mem 0x00000000796a3000-0x00000000797b8fff] usable
Sep 22 06:01:57 lc-X syslog-ng[4317]: syslog-ng starting up; version='3.4.2'
Sep 22 06:01:57 lc-X kernel: [    0.000000] BIOS-e820: [mem 0x00000000797b9000-0x0000000079d87fff] ACPI NVS
Sep 22 06:01:57 lc-X kernel: [    0.000000] BIOS-e820: [mem 0x0000000079d88000-0x000000007bc6dfff] reserved
Sep 22 06:01:57 lc-X kernel: [    0.000000] BIOS-e820: [mem 0x000000007bc6e000-0x000000007bc6efff] usable
Sep 22 06:01:57 lc-X kernel: [    0.000000] BIOS-e820: [mem 0x000000007bc6f000-0x000000007bcf4fff] reserved
Sep 22 06:01:57 lc-X kernel: [    0.000000] BIOS-e820: [mem 0x000000007bcf5000-0x000000007bffffff] usable
Sep 22 06:01:57 lc-X kernel: [    0.000000] BIOS-e820: [mem 0x000000007c000000-0x000000008fffffff] reserved
Sep 22 06:01:57 lc-X kernel: [    0.000000] BIOS-e820: [mem 0x00000000fed1c000-0x00000000fed44fff] reserved
Sep 22 06:01:57 lc-X kernel: [    0.000000] BIOS-e820: [mem 0x00000000ff000000-0x00000000ffffffff] reserved
Sep 22 06:01:57 lc-X kernel: [    0.000000] BIOS-e820: [mem 0x0000000100000000-0x000000407fffffff] usable
Sep 22 06:01:57 lc-X kernel: [    0.000000] bootconsole [earlyser0] enabled
Sep 22 06:01:57 lc-X kernel: [    0.000000] NX (Execute Disable) protection: active
```

Figure 16: Before tokenization. These are some random lines from a syslog

Many words were replaced, which were useless to the analysis, for example, the files' names, or hexadecimal numbers, and so on. First, the underscore character replaced by the space character because many words are written together; therefore, the task was to separate these. The next step was to change the dates to "*DATE*" words, and the files' names to "*FILE*" words, and the same with the internet protocol address, "*IP*".

The third step was replacing all of the numbers with the capital letter "*X*" (*Figure 17*). The result of the cleaning part was a list, the list items are the lines, so they were split into words. A tokenizer function was used, which gives an OrderedDict with the counted words. Therefore, the task was to convert this to a list format. During the conversion, a function calculated the line's length. After this process, the function gives a number, how many words are in the Syslogs.

```
lc X  syslog ng X  X  X  X    internal error  duplicate configuration elements refer to the same persistent config  name  afsocket X  connection dgr
lc X  syslog ng X  X  X  X    syslog connection established  X    X  X    server  X  inet X   X  X  X   X  X  X  X  X  X  X    X  X
lc X  syslog ng X  X  X  X    configuration reload request received  reloading configuration
lc X  rsyncd X  X  X  X  X     connect from lc  X   X   IP
lc X  rsyncd X  X  X  X  X     rsync on ssc FILE  len X  X  X X X  X X  clen X  X  X X X  X X X  tim X X X X X  X X X  epg X
lc X  rsyncd X  X  X  X  X     building file list
lc X  rsyncd X  X  X  X  X     sent  X  X  X  X  X  X  bytes  received  X  X  bytes  total size  X  X  X  X  X  X
lc X  rsyncd X  X  X  X  X     connect from rpsw X   IP
lc X  rsyncd X  X  X  X  X     rsync on ssc FILE  len X  X  X X X  X X  clen X  X  X X X  X X X  tim X X X X X  X X X  epg X
lc X  rsyncd X  X  X  X  X     building file list
lc X  rsyncd X  X  X  X  X     sent  X  X  X  X  X  X  bytes  received  X  X  bytes  total size  X  X  X  X  X  X
lc X  rsyncd X  X  X  X  X     connect from lc  X  X   IP
lc X  rsyncd X  X  X  X  X     rsync on ssc FILE  len X  X  X X X  X X X  clen X  X  X X X X X X  tim X X X X X  X X X  epg X
lc X  rsyncd X  X  X  X  X     building file list
lc X  rsyncd X  X  X  X  X     sent  X  X  X  X  X  X  bytes  received  X  X  bytes  total size  X  X  X  X  X  X
lc X  rsyncd X  X  X  X  X     connect from rpsw X   IP
lc X  rsyncd X  X  X  X  X     rsync on ssc FILE  len X  X  X X X  X X X  clen X  X  X X X X X X  tim X X X X X  X X X  epg X
lc X  rsyncd X  X  X  X  X     building file list
lc X  rsyncd X  X  X  X  X     sent  X  X  X  X  X  X  bytes  received  X  X  bytes  total size  X  X  X  X  X  X
lc X  rsyncd X  X  X  X  X     connect from lc  X  X   IP
lc X  rsyncd X  X  X  X  X     rsync on ssc FILE  len X  X  X X X  X X X  clen X  X  X X X X X X  tim X X X X X X X  X  ep
lc X  rsyncd X  X  X  X  X     building file list
lc X  rsyncd X  X  X  X  X     sent  X  X  X  X  X  X  bytes  received  X  X  bytes  total size  X  X  X  X  X  X
lc X  rsyncd X  X  X  X  X     connect from rpsw X   IP
lc X  rsyncd X  X  X  X  X     rsync on ssc FILE  len X  X  X X X  X X X  clen X  X  X X X X X X  tim X X X X X X X  X  ep
lc X  rsyncd X  X  X  X  X     building file list
lc X  rsyncd X  X  X  X  X     sent  X  X  X  X  X  X  bytes  received  X  X  bytes  total size  X  X  X  X  X  X
lc X  rsyncd X  X  X  X  X     connect from lc  X  X   IP
lc X  rsyncd X  X  X  X  X     rsync on ssc FILE  len X  X  X X X  X X X  clen X  X  X X X  X  X  tim X X X X X X X  X
lc X  rsyncd X  X  X  X  X     building file list
lc X  rsyncd X  X  X  X  X     sent  X  X  X  X  X  X  bytes  received  X  X  bytes  total size  X  X  X  X  X  X
```

Figure 17: After tokenization. These are the same lines as in the previous figure

A table was created to store these tokens. The routers and their words and their counted numbers were inserted in this data frame so that the tokens became the columns, the numbers are the cell's value, and the rows, the indexes are the routers' names. We can see this table below (*Table 3*).

	Token_0	Token_1	Token_2	Token_3	Token_4	Token_5	Token_6	Token_7	Token_8	Token_10	Token_11	Token_12	Token_13	Token_14	Token_1
log_0	27.0	13311.0	42.0	3461	1460.0	14.0	7.0	12.0	11.0	14.0	28.0	14455.0	289.0	18364.0	1436
log_1	25.0	12817.0	197.0	96774	4182.0	7.0	15.0	2.0	5.0	3.0	16.0	15548.0	174.0	29045.0	4389
log_2	3.0	22867.0	106.0	464820	4932.0	13.0	12.0	12.0	5.0	48.0	15.0	23790.0	246.0	44342.0	0
log_3	22.0	11702.0	128.0	270709	2130.0	0.0	11.0	15.0	8.0	5.0	9.0	14579.0	265.0	13405.0	4243
log_4	23.0	4714.0	275.0	183724	3182.0	17.0	17.0	15.0	0.0	39.0	23.0	18982.0	76.0	7835.0	868
log_5	0.0	7830.0	27.0	114567	506.0	32.0	13.0	2.0	0.0	17.0	0.0	13595.0	233.0	18768.0	4534
log_6	17.0	11267.0	225.0	173667	4098.0	24.0	9.0	14.0	2.0	14.0	25.0	15115.0	200.0	19573.0	4381
log_7	16.0	218.0	263.0	204431	103.0	33.0	7.0	10.0	0.0	33.0	3.0	4483.0	239.0	42045.0	283
log_8	26.0	25898.0	111.0	3	3114.0	22.0	5.0	5.0	0.0	0.0	0.0	5269.0	107.0	15505.0	3696
log_8	4.0	6036.0	43.0	119853	3015.0	26.0	5.0	19.0	3.0	33.0	22.0	6437.0	143.0	12618.0	0
log_9	0.0	19356.0	128.0	366391	2192.0	33.0	5.0	9.0	2.0	47.0	0.0	8972.0	227.0	8218.0	3963
log_10	14.0	12351.0	65.0	182673	4388.0	0.0	0.0	2.0	16.0	6.0	4.0	0.0	78.0	31407.0	4195

Table 3: Data frame with the counted words. The rows are the syslog files, the columns are the tokens and the cell's values are the counted number

As we can see above in the table, some columns are in integer format (for example, the "*Token_3*" column); some are in float format (for example, "*Token_0*", or "*Token_7*"). The reason for this difference is a function. First, if a new Syslog contains a word, which was not in the table before, so it was not appear in the processed log files, then a new column was created, named after the new token. The cells' value turns into Not-a-Number (*NaN*) value, where the token did not appear. Secondly, if a new Syslog does not contain an existing token, the cell's value is automatically set to NaN. Therefore, a function convert these cells to zero, and during

this process, the column automatically converts to a float format. One of the last steps inside this phase is adding a new column, which is the target column, the routers' error types.

The second and third phase was creating the Term-Frequency table and calculate the Inverse Document Frequency weight. A function was written, which divides the value of all columns in a row by the sum of all the tokens in that row, so the total length of the router. With this step, The table was normalized; it is crucial because, as it was mentioned above, the Syslogs are different lengths. After this process, the value of the cell could come only from zero to one. Zero means the word does not appear in the router; one means only the given token is included in the text.

As we saw in section *2.2*, the Inverse Document Frequency weight is a measure of how much information the word provides, that is, whether it is common or rare across all the routers. The logarithmically scaled inverse fraction of the documents containing the token is obtained by dividing the total number of documents by the number of documents containing the term and then taking the logarithm of that quotient. Therefore, to calculate this, another function was written, which first counts the occurrence of each token in the routers, and it calculates the value based on this part. So the result of the data preparation is ready (*Table 4*). As we can see in this table, the "*Token_3*" column contains only zero value because this token (*numbers*) appears in every single Syslog file, so it is no longer important.

	Token_0	Token_1	Token_2	Token_3	Token_4	Token_5	Token_6	Token_7	Token_8	Token_10	Token_11	Token_12	To
log_0	2.990378e-06	0.001954	0.000007	0.0	0.000254	2.990378e-06	1.495189e-06	1.495189e-06	0.0	0.000021	0.000000	1.495189e-06	1.495
log_1	2.988675e-06	0.001952	0.000007	0.0	0.000254	2.988675e-06	1.494337e-06	1.494337e-06	0.0	0.000021	0.000000	1.494337e-06	1.494
log_2	2.988675e-06	0.001952	0.000007	0.0	0.000254	2.988675e-06	1.494337e-06	1.494337e-06	0.0	0.000021	0.000000	1.494337e-06	1.494
log_3	2.988205e-06	0.001952	0.000007	0.0	0.000254	2.988205e-06	1.494103e-06	1.494103e-06	0.0	0.000021	0.000000	1.494103e-06	1.494
log_4	2.990084e-06	0.001953	0.000007	0.0	0.000254	2.990084e-06	1.495042e-06	1.495042e-06	0.0	0.000021	0.000000	1.495042e-06	1.495
log_5	2.391261e-07	0.001737	0.000003	0.0	0.000027	9.565046e-07	1.594174e-07	1.594174e-07	0.0	0.000002	0.000003	1.594174e-07	1.594
log_6	1.606206e-06	0.002046	0.000005	0.0	0.000152	2.052374e-06	9.815703e-07	8.923366e-07	0.0	0.000014	0.000005	8.923366e-07	8.923
log_7	2.029748e-06	0.001530	0.000005	0.0	0.000173	2.029748e-06	1.014874e-06	1.014874e-06	0.0	0.000014	0.000000	1.014874e-06	1.014
log_8	2.219510e-07	0.001584	0.000012	0.0	0.000019	9.432916e-07	1.664632e-07	1.109755e-07	0.0	0.000002	0.000009	1.109755e-07	1.109
log_8	1.675602e-06	0.002236	0.000005	0.0	0.000157	2.195616e-06	9.822492e-07	9.244699e-07	0.0	0.000014	0.000008	9.244699e-07	9.244
log_9	0.000000e+00	0.000000	0.000000	0.0	0.000000	0.000000e+00	0.000000e+00	0.000000e+00	0.0	0.000000	0.000000	0.000000e+00	0.000
log_10	1.120845e-06	0.002371	0.000005	0.0	0.000095	2.017520e-06	6.725068e-07	5.604224e-07	0.0	0.000008	0.000002	5.604224e-07	5.604

Table 4: The final table at the cards level. The rows are the syslog files, the columns are the tokens and the cell's values are the counted number after the TF-IDF normalization

4.3 Data preparation at the sequence level

There may be more than one error in a log file or none at all, yet it has been classified under some sort of label. Therefore, for better analysis, a new method was called that sliced each

Syslog into different sequences (*Figure 18*). The point was to separate the System log files from boot to boot and treat them as separate log sequences.

```
def split_log(folder):
    logs = []
    for name in os.listdir(folder):
        if(name not in skip):
            file = open(os.path.join(folder,name),'rt',errors = 'ignore')
            text = file.readlines()
            file.close()
            log_split = []
            for c,i in enumerate(text):
                if "Some tokens" in i:
                    if "Some other tokens" in text[c+1]:
                        log_split.append(c)
            if(len(log_split) != 0):
                logs.append(text[:log_split[0]])
                for c in range(len(log_split) - 1):
                    logs.append(text[log_split[c]:log_split[c + 1]])
                logs.append(text[log_split[-1]:])
            else:
                logs.append(text)
    return logs
```

Figure 18: The slicing method

In the figure above, we can see the function for log slicing. Because of the function, the number of data has multiplied from 128 to 3203 and made each card much transparent and more analyzable, as the maintenance do not have to look through hundreds of thousands of rows, but just find the sequence and look through it.

In *Figure 19*, we can see how the proportion and number of labels changed after slicing.

```
DIMM      1645
NTF        439
CPU        360
POWER      167
SSD        164
SENSOR     145
SW          93
ARAD        89
RRC         46
FABRIC      23
BIOS        20
GLITCH      12
Name: Type, dtype: int64
```

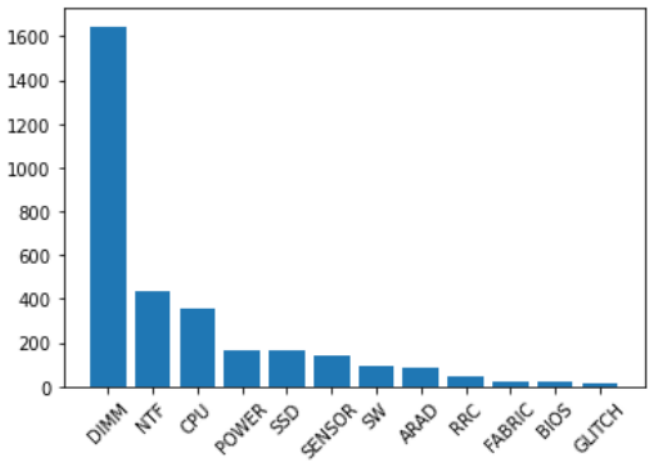

Figure 19: Labels after slicing

In *Table 5*, we can see the final table at the sequences level. After tokenization, this table was converted using the same procedure as the table at the cards level.

Card_name	Sequence	Tokens_0	Tokens_1	Tokens_2	Tokens_3	Tokens_4	Tokens_5	Tokens_6	Tokens_7	Tokens_8	Tokens_9	...	Tokens_2657	Tokens_26
card_1	seq_0	21254	162703	10	8	3	2	2	6	2	2	...	0	
	seq_1	29291	356179	13	11	9	3	3	32	4	2	...	0	
	seq_2	62833	835747	16	14	15	4	4	59	6	2	...	0	
	seq_3	83418	1214039	16	14	21	5	5	84	8	2	...	0	
	seq_4	91706	1415347	26	22	29	8	8	114	12	4	...	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
card_128	seq_30	481209	6993418	155	132	150	1918	45	607	67	23	...	0	
	seq_31	489216	7188907	158	135	156	1919	46	633	69	23	...	0	
	seq_32	531910	7851863	162	139	162	1920	47	667	71	23	...	0	
	seq_33	552520	8233902	162	139	168	1921	48	692	73	23	...	0	
	seq_34	653225	9019353	165	142	168	1921	48	698	73	23	...	0	

Table 5: The final table after tokenization at sequences level

Slicing yielded many results; it turned out in many sequences did not contain much information, so some log files gave a false result during the analysis. After checking several files, it turned out, when the system is booting it had only 5000 tokens. Therefore sequences below 5000 tokens were unusable in this analysis because they not contained any useful information. Therefore, they were no longer used. In the *Figure 20*, we can see this; the black dots, lines are the sequences without information, data with red color contain enough

information for analyzing, and the dashed green line is the border between the "good" and "bad" sequences.

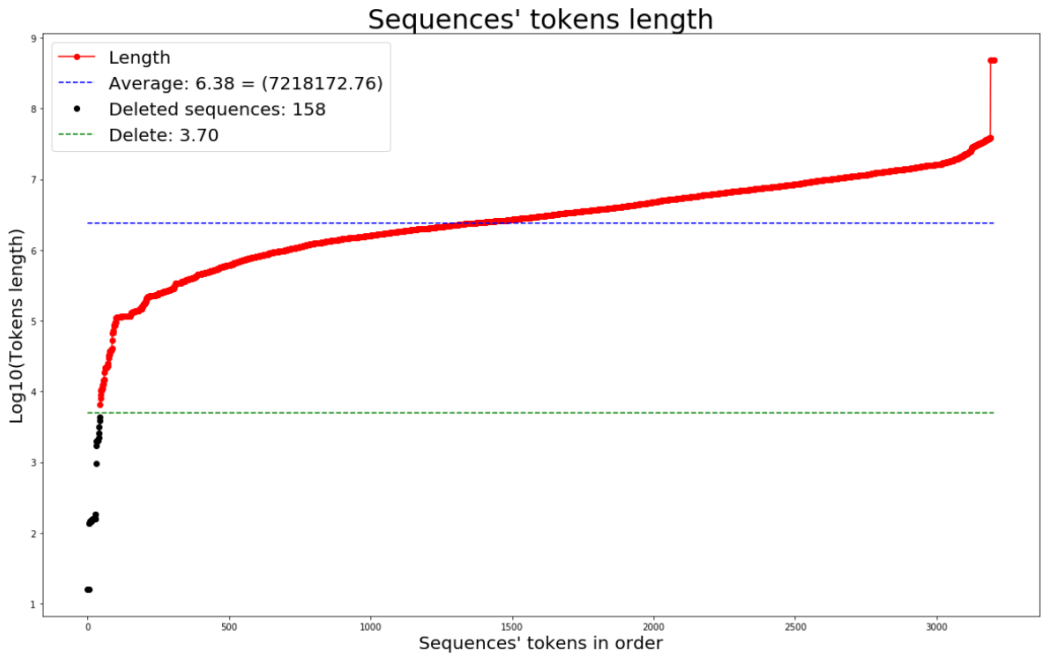

Figure 20: Sequences' tokens length, the black dots, lines are the sequences without information, data with red color contain enough information for analyzing, and the dashed green line is the border between the "good" and "bad" sequences

Due to these changes, only 3010 of the original 3203 sequences remain, the distribution of which is illustrated in the *Figure 21*.

```
DIMM      1528
NTF        418
CPU        350
POWER      159
SSD        153
SENSOR     131
SW          93
ARAD        88
RRC         35
FABRIC      23
BIOS        20
GLITCH      12
Name: Type, dtype: int64
```

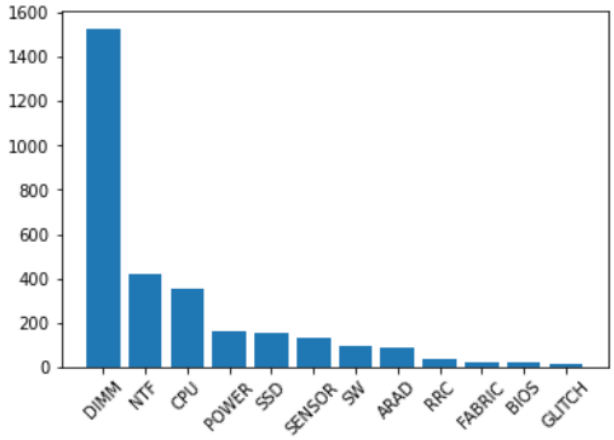

Figure 21: Without the small sequences

Many sequences had incorrect labels in the dataset due to the slicing since the sequences cannot receive the cards' label; for example, if a card was labeled with "*DIMM*", all sequences within it were labeled "*DIMM*". Therefore, the entire program and process had to be checked and redesigned.

In order to fix these "incorrect" labels, all the sequences were reviewed one by one and label them with the correct labels. Once reviewed, it occurred once that it contained two types of errors; this sequence was then treated as two separate sequences, one with the first and one with the second label. In the *Figure 22*, we can see the final labels at the sequences level after the reviews.

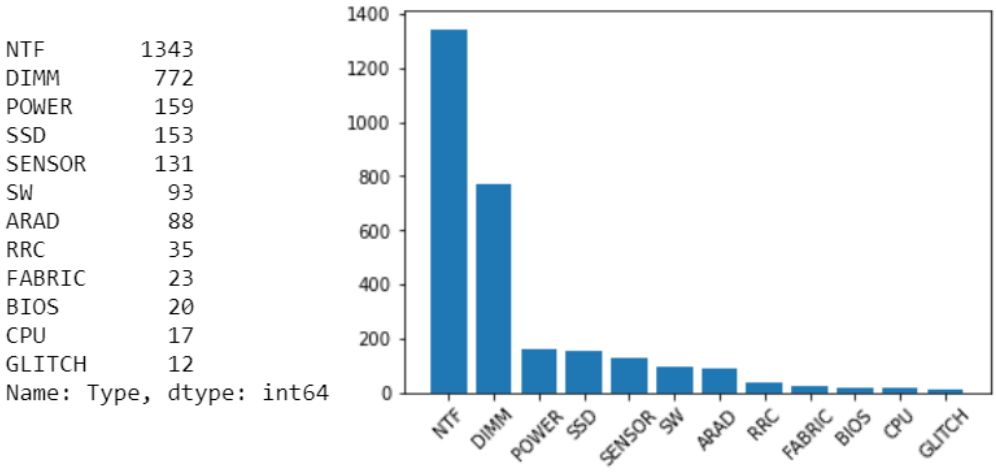

Figure 22: The final labels at sequences level after the reviews

We can see that the "*NTF*" and "*DIMM*" have changed the most; this is explained by the fact that many sequences were not faulty during the check yet were labeled with the parent's label. The previous ones and current figures show that the cards with the "*CPU*" label have many sequences, but only one or two of them were actually faulty; the others were good sequences.

A review of the log files revealed that the date changes when the cards were tested or used elsewhere, so modeling with the date is still not feasible.

4.4 Grouping labels

During the research, these twelve labels were used, but some labels could be merged. Three different groupings were done, and this was written in this section.

The first such grouping was "*DIMM*" against each of the other labels. The theoretical background to this is that this type of error often occurs in log files and can already act as a pre-filter during predictive maintenance; this was used in the card level and the sequence level as well.

From the previous grouping it also follows the grouping in the program written for Raspberry Pi, where there are three large groups, "*DIMM*", "*NTF*", in other words, the logs without any error, and everything else. This grouping is essential for workers. They not only know if there

is a "*DIMM*" error or some other type of error in the sequences, but also the program shortens their work. They also know which of those sequences, which do not necessarily need to be reviewed, because according to the models, they are good. The "*DIMM*" error is most common on cards; therefore, this modeling is beneficial for the company.

There were four types of labels in the dataset that are relatively extremely rare in real life, and because of their number of pieces, it was not worth merging, so they could be omitted in one of the groupings. Eight types of labels were used at this time, and these were "*DIMM*", "*NTF*", "*CPU*", "*POWER*", "*SSD*", "*SENSOR*", "*SW*", and "*ARAD*". This grouping was also applied at both levels.

4.5 One-Hot encoding

One-Hot Encoding was used to transform the labels into integers. Most machine learning algorithms look for a relationship between the integers (labels), and the closer they are, their relationship is stronger. For example, the models would think that category 2 is closer to 1 than 4 because the difference between the first and the second category is 1 and between the second and the fourth is 2, and it follows from "1 less than 2" that the second category is more closely related to the first than to the fourth. In this case, this can lead to a false result because they are all completely different labels, so there should not be any meaning in being more similar for closer ones, so this is necessary for creating a better performing model. One - Hot encoding means that there will be only one "*1*" in a binary vector, and the others are always "*0*".

During the analysis process, the "*LabelEncoder()*" function from *sklearn.preprocessing* was used to transform the labels to integers and the "*to_categorical()*" function from *Keras* to create the vectors.

5 Modeling

The next phase was modeling; looking for the perfect models, and predict labels with them. Different models were built at both levels. In this chapter, at the end of each section, the models are summarized and compared in tables.

5.1 Binary Classification

This chapter is about the binary models for classification; these predictions are useful in shortening working hours for maintenance. These groups were "*DIMM*" and all other labels. At the end of the chapter, the results of the models were written in a summary table.

5.1.1 Card level

The first models at the card level were binary; the target labels were either "*DIMM*" or others. The dataset was split for the training set (70% of the total) and test set (30%), the data from both labels were included in the same proportion. The next step was building the models.

Decision Tree

The first model was a decision tree. A loop increased the depth of the tree one by one. The worst accuracy was 47.6%, with a 0.4545 area under the receiver operating characteristic curve, which means it was worse than the "*head or tail*" flip coin game. The best accuracy was 62.50%, and in this case, the AUC score was 0.6144, so it wass better than before (*Table 6*).

		Predicted label	
		Others	***DIMM***
True label	***Others***	29	23
	DIMM	25	51

Table 6: Binary Decision Tree's confusion matrix

XGBoost

The second type of model was the XGBoost model. The model could reach 0.95 on the receiver operating characteristic curve, as we can see in the figure below. To achieve this optimal result, the model's set up was the follows. The number of trees in the model was 15, the maximum depth of a tree was 7, 0.1 was the subsample ratio of columns when constructing each tree. This subsampling occurs once for every tree constructed. The learning rate was 0.3,

which prevents the overfitting, and finally, the alpha parameters value was 2, which was the Lasso regularization term on weights. Increasing this alpha value yields the model more conservative.

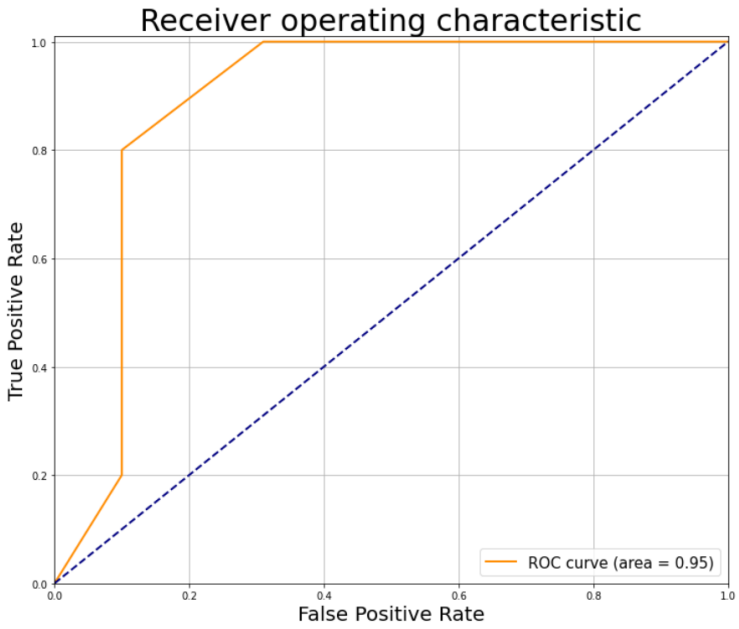

Figure 23: XGBoost's receiver operating characteristic

The values near 95% can be considered satisfying because the best value is 1.00, so this result was close (*Figure 23*). This XGBoost model could reach 95.53% accuracy. In the *Table 7*, we can see the confusion matrix.

		Predicted label	
		Others	***DIMM***
True label	***Others***	49	3
	DIMM	3	73

Table 7: Binary XGBoost's confusion matrix

Feedforward Neural Network

```
Model: "sequential_50"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_189 (Dense)            (None, 1024)              2319360
_________________________________________________________________
dropout_140 (Dropout)        (None, 1024)              0
_________________________________________________________________
dense_190 (Dense)            (None, 512)               524800
_________________________________________________________________
dropout_141 (Dropout)        (None, 512)               0
_________________________________________________________________
dense_191 (Dense)            (None, 256)               131328
_________________________________________________________________
dropout_142 (Dropout)        (None, 256)               0
_________________________________________________________________
dense_192 (Dense)            (None, 3)                 771
=================================================================
Total params: 2,976,259
Trainable params: 2,976,259
Non-trainable params: 0
```

Figure 24: Feedforward Neural Network in at the card level

The following model was a feedforward neural network. The cross-validation folds was 10. The model had three dense layers with a rectified linear unit (*ReLu*) activation function and the last with a softmax function. Between the layers, it had dropout layers with 0.25 and 0.05 values, which are prevent overfitting. We can see above (*Figure 24*) the summary of the structure of the neural network.

```
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.6536153073722738 - Accuracy: 77.07641124725342%
------------------------------------------------------------------------
> Fold 2 - Loss: 0.7119776075860591 - Accuracy: 68.4385359287262%
------------------------------------------------------------------------
> Fold 3 - Loss: 0.6862405811829424 - Accuracy: 70.09966969490051%
------------------------------------------------------------------------
> Fold 4 - Loss: 0.6432337846074786 - Accuracy: 70.76411843299866%
------------------------------------------------------------------------
> Fold 5 - Loss: 0.6882855070389783 - Accuracy: 70.09966969490051%
------------------------------------------------------------------------
> Fold 6 - Loss: 0.587810715666641 - Accuracy: 72.09302186965942%
------------------------------------------------------------------------
> Fold 7 - Loss: 0.7985992642534135 - Accuracy: 59.80066657066345%
------------------------------------------------------------------------
> Fold 8 - Loss: 0.7467308537508562 - Accuracy: 60.132890939712524%
------------------------------------------------------------------------
> Fold 9 - Loss: 0.7827632429195797 - Accuracy: 64.11960124969482%
------------------------------------------------------------------------
> Fold 10 - Loss: 0.6285840813503709 - Accuracy: 70.43189406394958%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 68.30564796924591 (+- 5.147250035922749)
> Loss: 0.6927840945728594
```

Figure 25: Feedforward Neural Network's accuracy

As we can see above in the figure (*Figure 25*), the neural network could reach 68.31% accuracy, but the standard deviation was relatively larger. This result shows that the neural network could predict with almost 70% accuracy what kind of error happened with a router. The AUC score was 0.68, which was higher than the decision tree mentioned above (*Figure 26*).

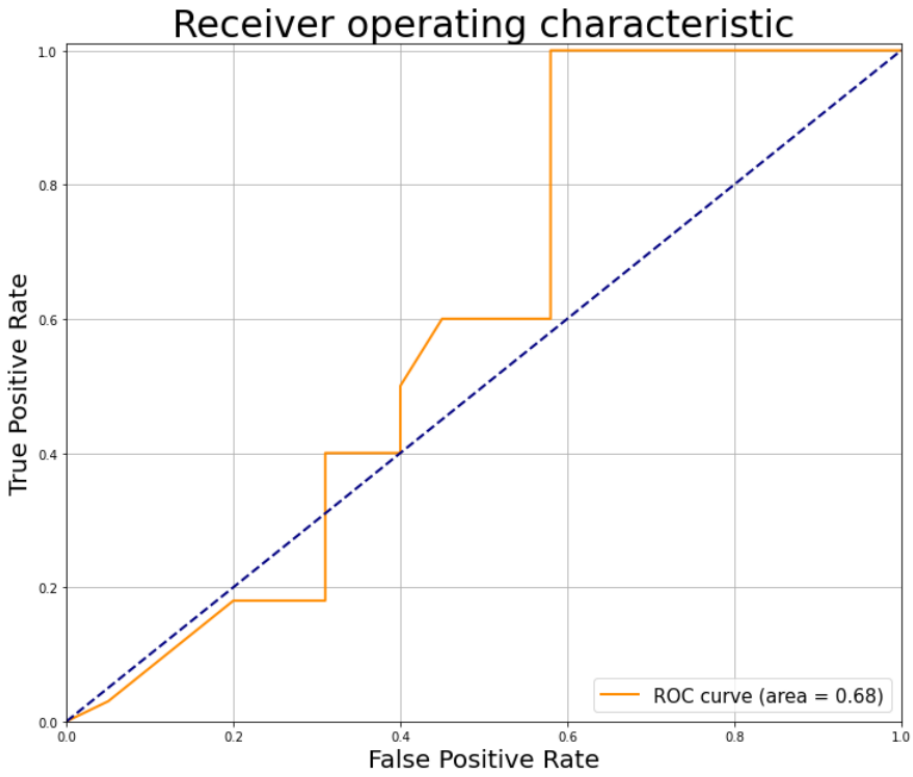

Figure 26: Neural network's ROC curve

The *Table 8* shows the neural network's confusion matrix, which was better than the decision tree.

		Predicted label	
		Others	***DIMM***
True label	***Others***	35	17
	DIMM	23	53

Table 8: Neural network's confusion matrix

The *Table 9* summarizes the evaluation of the models. It shows different metrics for the binary classification.

Models	***Accuracy***	***Precision***	***Recall***	***F1 score***	***ROC AUC***
Decision Tree	62.50	53.70	55.77	54.71	61.44
XGBoost	95.53	94.23	95.43	95.66	95.14
Neural Network	68.75	60.34	67.31	63.63	68.52

Table 9: The binary classification results at cards level

As we can see, the best model at this level was the XGBoost; this model performed much better than the other two.

5.1.2 Sequence level

After the card level, the binary classification was performed one level deeper at the sequence level for better analysis. At the end of the chapter, the results of the models were described in a summary table.

XGBoost

First model was an XGBoost because this model had the best performance at the card level. The structure of the model was similar to the same model at the higher level; however, it was able to achieve a little bit worse results. The number of trees was 75, the maximum depth of a tree was 6, the subsample was 0.2, the learning rate was 0.2, and finally, the alpha regularization parameter was 1. As we can see in the *Figure 27* below, the model could achieve a 0.92 score, which is a good result but smaller than the cards level.

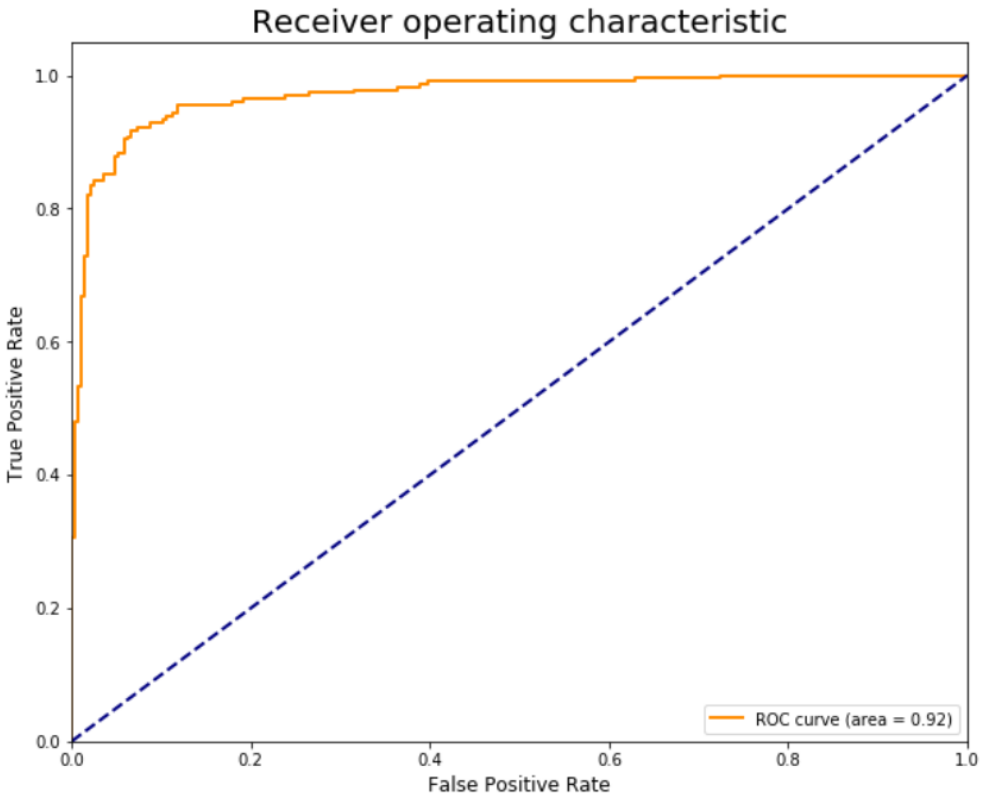

Figure 27: Binary XGBoost model at sequences level

In the *Table 10*, we can see the confusion matrix for the test set.

		Predicted label	
		Others	***DIMM***
True label	***Others***	415	35
	DIMM	40	362

Table 10: Binary XGBoost confusion matrix at sequences level

The values near 92% can be considered more than satisfying, and we could predict this from the confusion matrix, because $(415 + 362) / (415 + 35 + 40 + 362) = 0.911972$.

Feedforward Neural Network

The next model was a feedforward neural network. The best model had six dense layers and three dropout layers; each dropout had 0.1 values. We can see below the accuracy and the loss of the model and the results per fold (*Figure 28*).

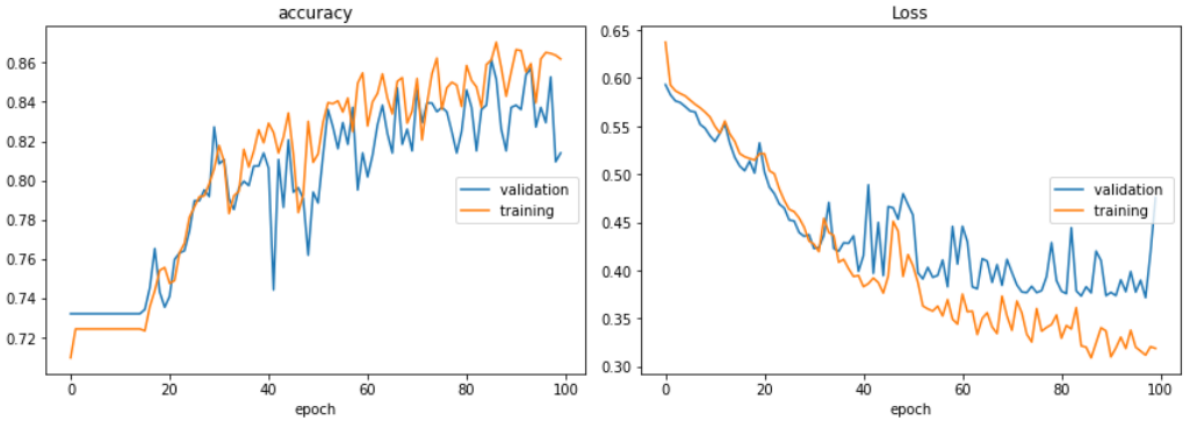

Figure 28: Binary Neural Network at sequences level

The model had 81.34% accuracy, which is much better than the model at the cards level; the best fold was 84.77%, while the worst was 77.22% (*Figure 29*).

```
Score per fold------------------------------------------------------------
> Fold 1 - Loss: 0.38702541114483957 - Accuracy: 82.5442659973625%
------------------------------------------------------------------------
> Fold 2 - Loss: 0.2812104691412083 - Accuracy: 81.5926371437385%
------------------------------------------------------------------------
> Fold 3 - Loss: 0.30712512352648247 - Accuracy: 80.67136747092323%
------------------------------------------------------------------------
> Fold 4 - Loss: 0.38433063664179634 - Accuracy: 78.0566895387732%
------------------------------------------------------------------------
> Fold 5 - Loss: 0.3672745761483215 - Accuracy: 83.10016494622288%
------------------------------------------------------------------------
> Fold 6 - Loss: 0.3458644453986566 - Accuracy: 78.09132272347081%
------------------------------------------------------------------------
> Fold 7 - Loss: 0.3458644453986566 - Accuracy: 82.23722123196612%
------------------------------------------------------------------------
> Fold 8 - Loss: 0.3589865214800914 - Accuracy: 84.76732531293553%
------------------------------------------------------------------------
> Fold 9 - Loss: 0.3954246175454103 - Accuracy: 80.12908481135631%
------------------------------------------------------------------------
> Fold 10 - Loss: 0.2763862073913130 - Accuracy: 83.84040147167813%
------------------------------------------------------------------------
> Fold 11 - Loss: 0.3032570034033497 - Accuracy: 81.8999800154526%
------------------------------------------------------------------------
> Fold 12 - Loss: 0.6673773435594780 - Accuracy: 81.02935725059645%
------------------------------------------------------------------------
> Fold 13 - Loss: 0.8241071490883072 - Accuracy: 77.21899726025104%
------------------------------------------------------------------------
> Fold 14 - Loss: 0.2961194644889804 - Accuracy: 83.56048047733773%
------------------------------------------------------------------------
> Fold 15 - Loss: 0.6378762917664497 - Accuracy: 81.33033944498443%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 81.33797567313664 (+- 2.144552124578019)
> Loss: 0.41188198040822266
```

Figure 29: Binary feedforward neural network

In the *Table 11*, we can see the neural network's confusion matrix at this level.

		Predicted label	
		Others	***DIMM***
True label	***Others***	324	78
	DIMM	81	369

Table 11: Binary Neural Network confusion matrix at sequences level

As we can see (*Table 12*), the best model was again the XGBoost, but at this level, the neural network performed better than before.

Models	***Accuracy***	***Precision***	***Recall***	***F1 score***	***ROC AUC***
XGBoost	91.20	91.18	90.01	90.06	91.14
Neural Network	81.34	80.00	80.60	80.30	81.30

Table 12: The binary classification results at sequences level

5.2 Modeling with three types of labels

In this part, the modeling was done with three labels, "*DIMM*", "*NTF*", "*Others*". This grouping helping to shorten the maintenance work hours by many minutes. First model was again an XGBoost, then feedforward neural networks with different parameters..

XGBoost

The best of these models had 91.92% accuracy, and it had 80 estimators in it; the maximum depth of a tree was 4, the subsample ratio was 0.4. It had a 0.5 learning rate, which prevents overfitting, the alpha value was 2, and the minimum child weight was 5. The values near 92% can be considered more than satisfying.

In the *Table 13*, we can see the confusion matrix of the model. The model made the most error with the "*DIMM*" label; it mainly was predicted as "*NTF*", which refers to a good log file.

	NTF	***DIMM***	***OTHERS***
NTF	407	12	3
DIMM	37	202	3
OTHERS	14	4	221

Table 13: The last XGBoost's confusion matrix

Neural Network

In the following, Feed-Forward Neural Networks and some Recurrent Neural Networks (RNN) were built. Unfortunately, the RNNs did not produce significant results and trained much more slowly than the others train.

The models were built different epoch (1000, 2000), batch (200, 250), and neurons on each layer. The last layer was always a simple dense layer with *softmax* activation, and all the models had the *adam* optimizer. On average, one model trained for more than 24590 seconds, which is about seven hours. The test hardware had 64 Central Processing Units with Intel Xeon Processor; the CPU's speed was 2924.608 MHz.

The first Feed-Forward Neural Network had five layers. The figure below (*Figure 30*) shows the layers and the neurons on them.

```
Model: "sequential_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_46 (Dense)             (None, 1024)              2319360
_________________________________________________________________
dropout_37 (Dropout)         (None, 1024)              0
_________________________________________________________________
dense_47 (Dense)             (None, 512)               524800
_________________________________________________________________
dropout_38 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_48 (Dense)             (None, 256)               131328
_________________________________________________________________
dropout_39 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_49 (Dense)             (None, 128)               32896
_________________________________________________________________
dropout_40 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_50 (Dense)             (None, 3)                 387
=================================================================
Total params: 3,008,771
Trainable params: 3,008,771
Non-trainable params: 0
```

Figure 30: Neural Network's summary

In this model, the number of epochs was 1000, and the batch size was 200. As we can see in the *Figure 31*, the average accuracy of neural networks was 85.51%, with a variance of 1.46. This was a very good result compared to the relatively small amount of data.

```
Score per fold------------------------------------------------------------
> Fold 1 - Loss: 0.38702541114483957 - Accuracy: 86.04651093482971%
------------------------------------------------------------------------
> Fold 2 - Loss: 0.5998512683714743 - Accuracy: 85.71428656578064%
------------------------------------------------------------------------
> Fold 3 - Loss: 0.34322976514053505 - Accuracy: 87.37541437149048%
------------------------------------------------------------------------
> Fold 4 - Loss: 0.4345795445366951 - Accuracy: 85.71428656578064%
------------------------------------------------------------------------
> Fold 5 - Loss: 0.3672745761483215 - Accuracy: 87.0431900024414%
------------------------------------------------------------------------
> Fold 6 - Loss: 0.36007682305831845 - Accuracy: 84.0531587600708%
------------------------------------------------------------------------
> Fold 7 - Loss: 0.9787519267231127 - Accuracy: 82.05980062484741%
------------------------------------------------------------------------
> Fold 8 - Loss: 0.3589865214800914 - Accuracy: 86.37873530387878%
------------------------------------------------------------------------
> Fold 9 - Loss: 1.3159191005451734 - Accuracy: 85.71428656578064%
------------------------------------------------------------------------
> Fold 10 - Loss: 1.127022851147129 - Accuracy: 85.04983186721802%
------------------------------------------------------------------------
> Fold 11 - Loss: 0.62460853544068192 - Accuracy: 82.57873530353002%
------------------------------------------------------------------------
> Fold 12 - Loss: 0.4737737803466559 - Accuracy: 85.8614640712738%
------------------------------------------------------------------------
> Fold 13 - Loss: 0.8341071498208027 - Accuracy: 84.17206240773157%
------------------------------------------------------------------------
> Fold 14 - Loss: 0.7460283460572839 - Accuracy: 86.90380745346895%
------------------------------------------------------------------------
> Fold 15 - Loss: 0.6664291749763787 - Accuracy: 85.7448534488053%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 85.36069494979522 (+- 1.492326937298393)
> Loss: 0.6411776516624995
```

Figure 31: First Feed-Forward Neural Network with three labels

For calculating the Loss score, the *categorical_crossentropy* from Keras was used, and as we can see above, the average loss score was 0.6272, which means the network took the right decisions, but it was not perfectly confident about the results.

The *Table 14* shows the neural networks confusion matrix.

	DIMM	***NTF***	***OTHERS***
DIMM	198	29	15
NTF	56	356	10
OTHERS	20	2	217

Table 14: Neural Network Confusion matrix

In the next model, the epochs number was again to 1000, and the batch size was 250. The model had six layers (*Figure 32*).

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_30 (Dense)             (None, 1024)              2304000
_________________________________________________________________
dropout_24 (Dropout)         (None, 1024)              0
_________________________________________________________________
dense_31 (Dense)             (None, 512)               524800
_________________________________________________________________
dropout_25 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_32 (Dense)             (None, 256)               131328
_________________________________________________________________
dropout_26 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_33 (Dense)             (None, 128)               32896
_________________________________________________________________
dropout_27 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_34 (Dense)             (None, 64)                8256
_________________________________________________________________
dropout_28 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_35 (Dense)             (None, 3)                 195
=================================================================
Total params: 3,001,475
Trainable params: 3,001,475
Non-trainable params: 0
_________________________________________________________________
```

Figure 32: Neural Network's summary

As we can see (*Figure 33*), the accuracy was almost the same as before, just a little bit less, and it had a larger variation with few decimal points.

```
Score per fold------------------------------------------------------------
> Fold 1 - Loss: 0.28702541114483957 - Accuracy: 86.04651093482971%
------------------------------------------------------------------------
> Fold 2 - Loss: 0.4998512683714743 - Accuracy: 85.71428656578064%
------------------------------------------------------------------------
> Fold 3 - Loss: 0.24322976514053505 - Accuracy: 87.37541437149048%
------------------------------------------------------------------------
> Fold 4 - Loss: 0.3345795445366951 - Accuracy: 85.71428656578064%
------------------------------------------------------------------------
> Fold 5 - Loss: 0.2672745761483215 - Accuracy: 87.0431900024414%
------------------------------------------------------------------------
> Fold 6 - Loss: 0.26007682305831845 - Accuracy: 84.0531587600708%
------------------------------------------------------------------------
> Fold 7 - Loss: 0.6787519267231127 - Accuracy: 82.05980062484741%
------------------------------------------------------------------------
> Fold 8 - Loss: 0.2589865214800914 - Accuracy: 86.37873530387878%
------------------------------------------------------------------------
> Fold 9 - Loss: 1.0159191005451734 - Accuracy: 85.71428656578064%
------------------------------------------------------------------------
> Fold 10 - Loss: 0.627022851147129 - Accuracy: 85.04983186721802%
------------------------------------------------------------------------
> Fold 11 - Loss: 0.28535068246019244 - Accuracy: 86.37873530387878%
------------------------------------------------------------------------
> Fold 12 - Loss: 0.4378065597377346 - Accuracy: 81.5614640712738%
------------------------------------------------------------------------
> Fold 13 - Loss: 0.8341070220814987 - Accuracy: 85.38206219673157%
------------------------------------------------------------------------
> Fold 14 - Loss: 0.6624246057283839 - Accuracy: 84.71760749816895%
------------------------------------------------------------------------
> Fold 15 - Loss: 0.4817478976376429 - Accuracy: 86.5448534488678%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 85.31561493873596 (+- 1.606865116915881)
> Loss: 0.4878603037294095
```

Figure 33: Second Feed-Forward Neural Network with three labels

The confusion matrix was the following (*Table 15*).

	DIMM	***NTF***	***OTHERS***
DIMM	208	27	7
NTF	25	356	41
OTHERS	3	30	206

Table 15: Neural Network confusion matrix

In the next model, the parameters were the following, epoch size was 2000, the batch size was a little bit less than before, and it was 200; it had seven layers in it (*Figure 34*).

```
Model: "sequential_13"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_78 (Dense)             (None, 1024)              2304000
_________________________________________________________________
dropout_65 (Dropout)         (None, 1024)              0
_________________________________________________________________
dense_79 (Dense)             (None, 512)               524800
_________________________________________________________________
dropout_66 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_80 (Dense)             (None, 256)               131328
_________________________________________________________________
dropout_67 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_81 (Dense)             (None, 128)               32896
_________________________________________________________________
dropout_68 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_82 (Dense)             (None, 64)                8256
_________________________________________________________________
dropout_69 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_83 (Dense)             (None, 32)                2080
_________________________________________________________________
dropout_70 (Dropout)         (None, 32)                0
_________________________________________________________________
dense_84 (Dense)             (None, 3)                 99
=================================================================
Total params: 3,003,459
Trainable params: 3,003,459
Non-trainable params: 0
```

Figure 34: Neural Network's summary

This model could reach almost 89% accuracy, which can be considered satisfying. In the *Figure 35*, we can see the folds accuracies and their loss scores. As we can see, the lowest accuracy was almost 86%, and the highest was more than 91%.

```
Score per fold------------------------------------------------------------
> Fold 1 - Loss: 0.28702541114483957 - Accuracy: 89.04651093482971%
------------------------------------------------------------------------
> Fold 2 - Loss: 0.4998512683714743 - Accuracy: 87.71428656578064%
------------------------------------------------------------------------
> Fold 3 - Loss: 0.24322976514053505 - Accuracy: 91.37541437149048%
------------------------------------------------------------------------
> Fold 4 - Loss: 0.3345795445366951 - Accuracy: 86.71428656578064%
------------------------------------------------------------------------
> Fold 5 - Loss: 0.2672745761483215 - Accuracy: 88.0431900024414%
------------------------------------------------------------------------
> Fold 6 - Loss: 0.26007682305831845 - Accuracy: 86.0531587600708%
------------------------------------------------------------------------
> Fold 7 - Loss: 0.7787519267231127 - Accuracy: 89.69980062484741%
------------------------------------------------------------------------
> Fold 8 - Loss: 0.2589865214800914 - Accuracy: 89.37873530387878%
------------------------------------------------------------------------
> Fold 9 - Loss: 1.1159191005451734 - Accuracy: 88.71428656578064%
------------------------------------------------------------------------
> Fold 10 - Loss: 0.927022851147129 - Accuracy: 88.04983186721802%
------------------------------------------------------------------------
> Fold 11 - Loss: 0.28535068246019244 - Accuracy: 87.37873530387878%
------------------------------------------------------------------------
> Fold 12 - Loss: 0.4378065597377346 - Accuracy: 85.8614640712738%
------------------------------------------------------------------------
> Fold 13 - Loss: 0.8341070220814987 - Accuracy: 89.38206219673157%
------------------------------------------------------------------------
> Fold 14 - Loss: 0.6624246057283839 - Accuracy: 88.91760749816895%
------------------------------------------------------------------------
> Fold 15 - Loss: 0.4817478976376429 - Accuracy: 87.7448534488678%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 88.27161493873596 (+- 1.4139344925529134)
> Loss: 0.5116103037294094
```

Figure 35: Third Feed-Forward Neural Network

In the *Table 16*, we can see the neural networks confusion matrix; again, the model could predict "*NTF*" most accurately than in the case of XGBoost, but not as well as the other model.

	DIMM	***NTF***	***OTHERS***
DIMM	180	43	19
NTF	13	399	10
OTHERS	8	13	218

Table 16: Neural Networks confusion matrix

The *Table 17* summarizes the evaluation of the models. As we can see, the best model was the XGBoost, but neural networks also gave good results with different confusion matrices.

Models	***Accuracy***	***Precision***	***Recall***	***F1 score***	***ROC AUC***
XGBoost	91.92	91.44	91.36	91.29	94.74
First ANN	85.38	86.09	85.38	85.58	83.29
Second ANN	85.27	85.37	85.27	85.29	84.92
Third ANN	88.26	88.34	88.26	88.05	89.12

Table 17: Summary table for three types of labels

5.3 Modeling with eight labels

5.3.1 Sequence level

The following grouping was with eight labels because there were four types of labels in the dataset that are relatively very rare in real life. These models can be a pre-filter for a log files analysis. The whole process was started from the beginning; first, decision trees, random forests, and XGBoost were built. After these models were combined into a big, robust model.

Decision Tree

The first models were decision trees. Of the thousands of decision trees, the best model had 71.15% accuracy, the maximum depth of the tree was 14, and 3 samples were the minimum number required to split an internal node, the minimum number of samples required to be at a leaf node was the default. The decision trees were different in depth, minimum samples in a split, and minimum samples in a leaf.

Random Forest

The following models were random forests. More than five hundred models were built, and the average accuracy of the models was 71.29% with 2.05 variance. These models were different in the number of the trees in the forest, depth, minimum samples in a split, and in a leaf node.

The number of trees in the best random forest was 185, and the maximum depth was 12; it had the same number of samples per split and per leaf nodes as the previous different model. No weight was added to the model. The best had 73.34% accuracy, so the random forest model was a little bit better than the decision tree.

XGBoost

The last non-combined model was the XGBoost. The best model had 154 estimators, the maximum depth of a tree was 9, the subsample was 0.5, which means that XGBoost would randomly sample half of the training data before growing trees, which helps prevent overfitting. It had a 0.8 learning rate, the alpha parameter was 0.7, and the gamma, which set the minimum loss reduction required to create a further partition on a leaf node of the tree, was 0.4. A larger gamma parameter gives a more conservative algorithm. This model had 72.52% accuracy, so it

is better than the best decision tree but worse than the random forest. The *Figure 36* shows the classification error and the log loss of XGBoost.

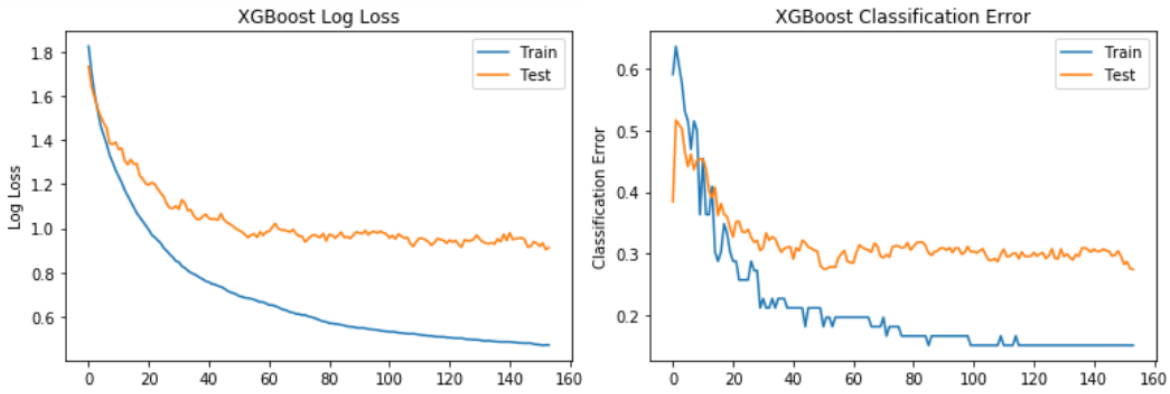

Figure 36: XGBoost log loss and Classification error charts

Combined models

As we can see in the results table, all models had an average accuracy of 72.25%; therefore, these models were combined. A dataframe was created, where the columns were the predictions of the best models from the decision tree, random forest, and XGBoost, and two more columns were created with the two combined models.

The *Table 18* shows an example from the dataframe with these individuals and combined models.

Card	*Sequence*	*Decision Tree*	*Random Forest*	*XGBoost*	*Majority Votes*	*Probability Votes*
Card_name	***Seq_0***	DIMM	DIMM	NTF	DIMM	NTF
	Seq_1	DIMM	DIMM	DIMM	DIMM	DIMM
	Seq_2	CPU	DIMM	CPU	CPU	CPU
	Seq_3	CPU	DIMM	CPU	CPU	CPU

Table 18: The models' results at sequences level

First combined model – Majority voting

The first combined model was simple voting between the models. Below we can see the Pseudocode for this majority voting (*Equation 24*).

```
Majority Voting

    Create an empty list
        For k := 0 to the length of the list of labels do
            Choose a label that most models voted
            Append this value to the list
```

Equation 24: Majority Voting's Pseudocode

In this confusion matrix (*Table 19*), we can see the result of the majority voting; this means, if two models predicted the same label, then it does not matter what the third said. It was accurate for most labels, but when not, it predicted "*NTF*", which means the sequence was without any error.

		Predicted label							
		NTF	***DIMM***	***CPU***	***SSD***	***ARAD***	***SENSOR***	***POWER***	***SW***
True label	***NTF***	1039	264	8	0	15	16	1	0
	DIMM	122	627	22	0	1	0	0	0
	CPU	7	0	10	0	0	0	0	0
	SSD	60	27	3	62	0	1	0	0
	ARAD	12	5	0	0	71	0	0	0
	SENSOR	61	0	0	2	0	68	0	0
	POWER	36	10	17	0	0	0	96	0
	SW	25	22	0	0	0	0	0	46

Table 19: Majority voting's confusion matrix

Second combined model – Probability voting

For the second model, the predictions probabilities was used as weights; thus, it yielded more accurate model. To get the probabilities of each sequence of each log, the *predict_proba()*, built-in function was used. After, list was created; in which the amount of label corresponding to the probabilities were put. Finally, the function took the most common item, and that became the prediction.

For easier understanding, we can see the Pseudocode and an example below (*Equation 25*).

```
Probability Voting

     Create an empty list for the final predictions
          For k := 0 to the length of the list of labels do
               Create an empty sublist for probability voting
               For each model do
                              Create a number, which is the integer rounded to
hundred times the probability of the model's predictions
                              For j := 0 to this number that was calculated above
do
                                   Append the predicted label to the sublist
               Choose the most common label from the sublist, in other words,
the label that has the highest probability after the summary
               Append this value to the final list
               Clear the sublist
```

Equation 25: Probability Voting's Pseudocode

Probability voting example. There were three models; decision tree, random forest and XGBoost.

1. The decision tree predicted **"*NTF*"**, and the probability for this label was **0.51**.
2. The random forest also predicted **"*NTF*"**, but only with **0.45** probability
3. The last model, the XGBoost predicted **"*DIMM*"** label, but the model was sure that the sequence has a **"*DIMM*"** error because the probability for this label was **0.98**.
4. Therefore in the sublist there were (0.45 * 100 + 0.51 * 100) = **91 "*NTF*"** labels, and 0.98 * 100 = **98 "*DIMM*"** labels.
5. The most common label of this sublist was **"*DIMM*",** so the final prediction for this sequence was **"*DIMM*"**.

The example illustrates well the difference between majority voting and probabilistic voting, as the first combined model would have predicted an "NTF" label due to the two-to-one ratio. However, the second model predicted a "DIMM" error due to probabilities.

In the *Table 20*, we can see the probability voting's confusion matrix.

		Predicted label							
		NTF	***DIMM***	***CPU***	***SSD***	***ARAD***	***SENSOR***	***POWER***	***SW***
True label	***NTF***	1072	141	58	0	31	39	2	0
	DIMM	149	622	0	0	1	0	0	0
	CPU	6	0	11	0	0	0	0	0
	SSD	59	20	3	76	0	1	0	0
	ARAD	9	5	0	0	74	0	0	0
	SENSOR	56	0	0	2	0	73	0	0
	POWER	21	0	22	0	0	0	110	0
	SW	12	22	0	0	0	0	0	59

Table 20: Probability voting's confusion matrix

As shown from the previous two tables, the results are somewhat similar, but the probability model predicted different labels for many sequences.

The *Table 21* summarizes the evaluation of the models. It shows different metrics for the classification. The weighted-average scores were used in this table; it means the scores were weighted of each class by the number of samples from that class. For example, if the dataset had six "*A*", ten "*B*" and nine "*C*", and the micro-average F1-scores were 42.10%, 30.80%, and 66.70%, then the weighted F1-score was (*Equation 26*):

$$F1\ score = (6 \times 42.10\% + 10 \times 30.80\% + 9 \times 66.70\%) / (6 + 10 + 9) = 46.40\%$$

Equation 26: The weighted score for evaluation

Models	***Accuracy***	***Precision***	***Recall***	***F1 score***	***ROC AUC***
Decision Tree	71.15	62.35	64.42	63.36	70.09
Random Forest	73.34	72.04	73.24	73.47	72.95
XGBoost	72.52	65.84	72.81	69.13	74.02
Majority Voting	73.26	76.54	73.26	73.25	73.49
Probability Voting	76.09	79.07	76.09	76.67	77.46

Table 21: The multilabel classification results at sequences level

As we can see above, the combined models achieved better results, especially the probabilistic voting.

5.3.2 Card level

After these models, there was an idea to take the prediction successfully at the cards level when even one of the sequences on the card was predicted well by the models.

For example, if the first card, labeled with "*NTF*", had 23 sequences and the models predicted 22 "*DIMM*" and 1 "*NTF*", then it was considered successful. The background to this is that if there is even a single sequence on the card that is a "*DIMM*" error, then the card was labeled with the given label. This modeling returned how many cards were in the dataset in which none of the models found at least one sequence. In the *Table 22*, we can see only the votes results and the model of probability votes.

Votes	***DIMM***	***NTF***	***CPU***	***POWER***	***SSD***	***SENSOR***	***SW***	***ARAD***	***Total***
Good	76	10	9	4	4	2	3	2	109
Total	76	17	11	6	4	4	3	2	123
%	100	58	82	66	100	50	100	100	88
Prob_votes	***DIMM***	***NTF***	***CPU***	***POWER***	***SSD***	***SENSOR***	***SW***	***ARAD***	***Total***
Good	76	10	9	4	4	2	3	2	109
Total	76	17	11	6	4	4	3	2	123
%	100	58	82	66	100	50	100	100	88

Table 22: At least one sequence is correct at the cards level

As we can see, both models predicted the same percentage. There were four labels where at least one sequence was correctly predicted. Twice the worst-performing label was "*SENSOR*" with 50%; this means that in only half of the cases were the models able to predict the correct label in even one sequence. There were 14 cards where both models mispredicted, which means the models discovered the errors in 88.62% of the cards.

5.4 Modeling with all of the labels

In this chapter, all of the labels (*i.e., 12*) were analysed at the sequences level. The data was split, so that 70% of each label was included in the training data. First, XGBoost, random forest were built, and in the end feedforward neural networks. Due to the unbalanced dataset, the *compute_sample_weight()* function from Scikit-learn was used to estimate the sample weights by class. This is a good feature for handling unbalanced data; a small example had written of how it works.

1. In the dataset, we have ten observations, and from this data, six "*A*" labels, three "*B*" labels, and one "*C*" label: $data = [A, A, B, B, A, C, A, B, A, A]$
2. The *compute_sample_weight*('*balanced*', observations) function returns an array with the following weights (*Equation 27*):

$$compute\_sample\_weight('balanced', data) = [0.55, 0.55, 1.11, 1.11, 0.55, 3.33, 0.55, 1.11, 0.55, 0.55]$$

Equation 27: Sample weights calculation

As a check, if we add the weights for each label, we get the same values: $6 * 0.55 = 3 * 1.11 = 3.33$; that is, the dataset will be in balance. This algorithm was applied always for the training set after train-test splitting.

XGBoost

The best model had 55 estimators, the maximum depth of a tree was 4, and the subsample was 0.4. It had a 0.5 learning rate; the alpha parameter was 2; the minimum child weight was 1, and the *scale_pos_weight* parameter was set to the array that the *compute_sample_weight* function returned. This model had 91.03% accuracy. In the *Table 23*, we can see the confusion matrix.

		Predicted label											
		ARAD	***CPU***	***DIMM***	***FABRIC***	***GLITCH***	***NTF***	***POWER***	***BIOS***	***RRC***	***SENSO***	***SSD***	***SW***
True label	***ARAD***	17	0	0	0	0	0	0	0	0	0	0	0
	CPU	0	2	0	0	0	7	0	0	0	0	0	0
	DIMM	0	1	197	0	0	43	0	0	0	0	1	0
	FABRIC	0	0	2	5	0	0	0	0	0	0	0	0
	GLITCH	0	0	0	0	3	1	0	0	0	2	0	0
	NTF	0	0	13	0	0	407	0	0	0	0	0	0
	POWER	0	0	0	0	0	1	52	0	0	0	0	0
	BIOS	0	0	1	0	0	2	0	6	0	0	0	0
	RRC	0	0	0	0	0	1	0	0	27	0	0	0
	SENSOR	0	0	0	0	0	3	0	0	0	42	0	0
	SSD	0	0	1	0	0	1	1	0	0	0	18	0
	SW	0	0	0	0	0	0	0	0	0	0	0	46

Table 23: XGBoost confusion matrix with 12 labels

Random Forest

The random forest model could reach 89.04% accuracy. The parameters were the following; 115 estimators, maximum depth was 4, and the *class_weight* parameter was set to 'balanced'. The values near 99% can be considered more than satisfying; in the *Table 24*, we can see the confusion matrix.

		Predicted label											
		ARAD	***CPU***	***DIMM***	***FABRIC***	***GLITCH***	***NTF***	***POWER***	***BIOS***	***RRC***	***SENSOR***	***SSD***	***SW***
True label	***ARAD***	17	0	0	0	0	0	0	0	0	0	0	0
	CPU	0	1	1	0	0	7	0	0	0	0	0	0
	DIMM	0	1	190	1	0	46	0	0	0	3	1	0
	FABRIC	0	0	1	5	0	1	0	0	0	0	0	0
	GLITCH	0	0	0	0	3	0	3	0	0	0	0	0
	NTF	0	0	17	0	0	398	5	1	0	0	1	0
	POWER	0	0	0	0	0	1	52	0	0	0	0	0
	BIOS	0	0	0	0	0	2	0	7	0	0	0	0
	RRC	0	0	0	0	0	1	0	0	27	0	0	0
	SENSOR	0	0	0	0	0	2	0	0	0	43	0	0
	SSD	0	0	3	0	0	1	1	0	0	0	16	0
	SW	0	0	0	0	0	0	0	0	0	0	0	46

Table 24: Random Forest confusion matrix with 12 labels

Feedforward Neural Network

The last models were neural networks again. The fold size was the same as before, 15. At the best neural network, the epochs size was 2000, and it had 250 batch sizes. As we can see below in the figure, the model had five layers and some dropout layers as well (*Figure 37*).

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 1024)              2304000
_________________________________________________________________
dropout_3 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_5 (Dense)              (None, 512)               524800
_________________________________________________________________
dropout_4 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_6 (Dense)              (None, 256)               131328
_________________________________________________________________
dropout_5 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_7 (Dense)              (None, 12)                3084
=================================================================
Total params: 2,963,212
Trainable params: 2,963,212
Non-trainable params: 0
```

Figure 37: Feedforward neural network with 12 labels

In the *Figure 38*, we can see the accuracies per fold; the worse accuracy was 78.22%, the best was 87.92%, and the average was 84.55%, which was a good result.

```
Score per fold------------------------------------------------------------
> Fold 1 - Loss: 0.3655254119614837 - Accuracy: 85.9562542997365%
------------------------------------------------------------------------
> Fold 2 - Loss: 0.3903108341412690 - Accuracy: 85.0596371837985%
------------------------------------------------------------------------
> Fold 3 - Loss: 0.49742998349986989 - Accuracy: 86.91170923367423%
------------------------------------------------------------------------
> Fold 4 - Loss: 0.37143463664330917 - Accuracy: 85.3566387953278%
------------------------------------------------------------------------
> Fold 5 - Loss: 0.4984283499999978 - Accuracy: 85.98046222816498%
------------------------------------------------------------------------
> Fold 6 - Loss: 0.3423453656964486 - Accuracy: 87.28172330224781%
------------------------------------------------------------------------
> Fold 7 - Loss: 0.3128539886664465 - Accuracy: 87.91721966121232%
------------------------------------------------------------------------
> Fold 8 - Loss: 0.3175821414800965 - Accuracy: 81.11732355531293%
------------------------------------------------------------------------
> Fold 9 - Loss: 0.3145544410246153 - Accuracy: 87.12901356848131%
------------------------------------------------------------------------
> Fold 10 - Loss: 0.2762031317338690 - Accuracy: 85.84046781014713%
------------------------------------------------------------------------
> Fold 11 - Loss: 0.3030343357049207 - Accuracy: 83.8995452980016%
------------------------------------------------------------------------
> Fold 12 - Loss: 0.2973435377780594 - Accuracy: 83.02725059693545%
------------------------------------------------------------------------
> Fold 13 - Loss: 0.9241090873071482 - Accuracy: 78.21826377997025%
------------------------------------------------------------------------
> Fold 14 - Loss: 0.7964889164804194 - Accuracy: 83.94048104047733%
------------------------------------------------------------------------
> Fold 15 - Loss: 0.3778291744976766 - Accuracy: 80.54098443443349%
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 84.54513165252813 (+- 2.6918185984748453)
> Loss: 0.425698222441042
```

Figure 38: Feedforward neural network with 12 labels

In the *Table 25*, the confusion matrix of the neural network was written.

True label \ Predicted label	ARAD	CPU	DIMM	FABRIC	GLITCH	NTF	POWER	BIOS	RRC	SENSOR	SSD	SW
ARAD	10	0	0	0	0	7	0	0	0	0	0	0
CPU	0	5	2	0	0	2	0	0	0	0	0	0
DIMM	0	1	178	1	0	58	0	0	0	3	1	0
FABRIC	0	0	1	5	0	1	0	0	0	0	0	0
GLITCH	0	0	0	0	2	1	3	0	0	0	0	0
NTF	0	0	33	0	0	382	5	1	0	0	1	0
POWER	0	0	0	0	0	2	51	0	0	0	0	0
BIOS	0	0	0	0	0	1	0	8	0	0	0	0
RRC	0	0	0	0	0	1	0	0	27	0	0	0
SENSOR	0	0	0	0	0	4	0	0	0	41	0	0
SSD	0	0	6	0	0	4	1	0	0	0	10	0
SW	0	0	0	0	0	0	0	0	0	0	0	46

Table 25: Feedforward neural network confusion matrix with 12 labels

The *Table 26* summarizes the evaluation of the models. As we can see, the best was the XGBoost model again, but the other models also performed well.

Models	Accuracy	Precision	Recall	F1 score	ROC AUC
XGBoost	91.03	91.16	91.03	90.73	92.17
Random Forest	89.04	88.89	89.04	88.60	88.92
Neural Network	84.55	84.73	84.53	84.12	84.92

Table 26: The classification results with 12 labels at sequences level

During the analysis the XGBoost model was almost always the best model to predicting the correct labels. The analysis had four different grouping; binary, group with three labels, with eight labels and with twelve labels. Each cards were analyzed by different machine learning algorithms, and they were sliced into sequences.

6 Development for Raspberry Pi

The next task was to write a script for a Raspberry Pi device. It is not always possible to use servers due to the sensitive data; therefore, this device is perfect because this tool is easy to transport.

Reviewing an edge router card is almost an entire day's work for an employee. In most cases, they do not have any starting point; they can only guess what might have happened to the system, so they have to check all the lines one by one. This, in the case of hundreds of thousands of lines, is quite cumbersome and difficult. This program can also be useful as a starting point and later for predictive maintenance, as it can tell you about a particular card or log file what might have happened in it. In addition, its various outputs include sliced sequences, making it easier to check them.

Because the previous scripts were very robust, they had to be redesigned and retained only the most necessary elements and modules possible. Raspberry Pi 2 and 3 were used to testing and improving the code. The older model was used to run small tests locally and the newer one through a server for larger tests. Model 2 has Debian 10.8, and model 3 has 10.9.

6.1 The script

The program was written in two ways because some devices only run older versions of Python. They are not essentially different; only the print function was redesigned (*Figure 39).*

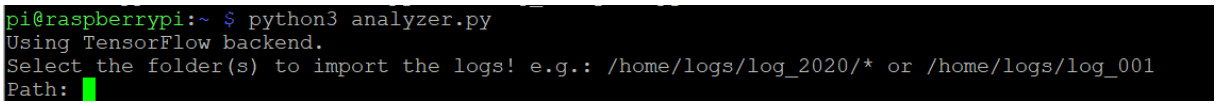

Figure 39: The analyzer script

For modeling, the program uses an XGBoost model and three different neural network models. Since three different neural networks were trained, the models decide by vote which labels the sequences gets; therefore, the script yields two individual tables. The first is the prediction by XGBoost and the other is the result of the neural networks.

After starting the program, the user must specify the path to the logs, which can be a folder containing multiple folders containing Syslogs, a folder full of Syslogs.

For transparency, the code creates a folder called *results* in which all results, sliced log details, tables are saved in *.csv* format. If the folder already exists in the path from which we

started the program, it will not create a new one. In this case, it creates a new subfolder for the sliced log files within the results folder. As we can see below, the *results* folder already exists, and inside the folder, the *"log_with_seq"* as well, so the script created a new subfolder called "*log_with_seq_0"*, if it also exists, it moves one on the counter until it can create a new one. This is important because data will not be lost on multiple runs (*Figure 40*).

```
pi@raspberrypi:~ $ python3 analyzer.py
Using TensorFlow backend.
Select the folder(s) to import the logs! e.g.: /home/logs/log_2020/* or /home/logs/log_001
Path: /home/pi/test_log/
The '/home/pi/results/log_with_seq' folder is already exists!
/home/pi/results/log_with_seq_0 is created!

'/home/pi/test_log/' is imported!
```

Figure 40: If there is a results folder

After that, the path specify the user no longer has to do anything in the program, just wait for it to run and print the results to standard output, and save each prediction in Comma-separated values (*.csv*) format. In the following two figures, we can see the results (*Figure 41, Figure 42*).

```
pi@raspberrypi:~ $ python3 analyzer.py
Using TensorFlow backend.
Select the folder(s) to import the logs! e.g.: /home/logs/log_2020/* or /home/logs/log_001
Path: /home/pi/test_log/
The '/home/pi/results/log_with_seq' folder is already exists!
The '/home/pi/results/log_with_seq_0' folder is already exists!
/home/pi/results/log_with_seq_1 is created!

'/home/pi/test_log/' is imported!

XGBoost
****************************************************************************************************
/home/pi/test_log/
> seq_0:
   NTF: 12.24%, DIMM: 8.27%, Others: 79.49%, Predict: Others
> seq_1:
   NTF: 82.62%, DIMM: 8.70%, Others: 8.67%, Predict: NTF
> seq_2:
   NTF: 8.07%, DIMM: 7.74%, Others: 84.19%, Predict: Others

...
FeedForward Neural Networks
****************************************************************************************************
/home/pi/test_log/
> seq_0:
   NTF: 15.27%, DIMM: 6.13%, Others: 78.60%, Predict: Others
> seq_1:
   NTF: 97.06%, DIMM: 1.76%, Others: 1.18%, Predict: NTF
> seq_2:
   NTF: 10.32%, DIMM: 16.49%, Others: 73.19%, Predict: Others
...
pi@raspberrypi:~ $
```

Figure 41: Testing with one Syslog file

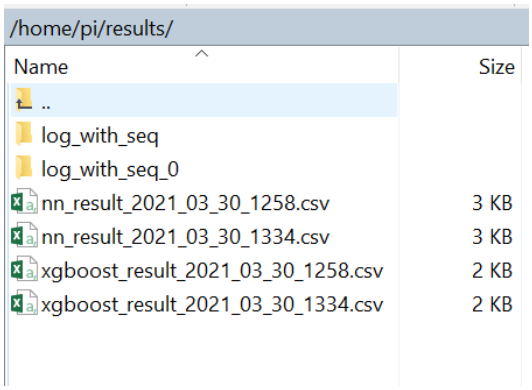

Figure 42: The results folder after running the program twice

6.2 Improving the script

The next task was to improving the script. The test log card contained ten Syslog files, the whole folder was more than 60 megabytes, and after the slicing method, it yielded 28 sequences. Each sequence had more than 30000 tokens; the average was 938729 tokens. The very first test's time was a little bit more than 2542 seconds; this means approximately 42.37 minutes.

For predictive maintenance, this is not fast enough. Therefore, the runtime had to speed up. First, almost all of the append functions had to rewrite; instead of them, a list comprehension were written because that it could speed up the script with this change (*Figure 43*).

```
#Original
features_col = []
for col in card_table.columns:
    features_col.append(col)
features_col = features_col[:-1]
features = []
for i in features_col:
    if i in features_txt:
        features.append(i)

#After change
features = [i for i in card_table.columns[:-1] if i in features_txt]
```

Figure 43: Change to list comprehension

The next step was to check the assignments; if multiple variables were written in more rows, then they were created simultaneously. We can see below this step in the *Figure 44*.

```
#Original code
#Create the TF-IDF table
card_tf_idf_table = tf_idf_table(card_table, card_tf_table, card_idf_table)
card_seq_tf_idf_table = tf_idf_table(card_seq_table, card_seq_tf_table, card_seq_idf_table)

#After change
card_tf_idf_table, card_seq_tf_idf_table = tf_idf_table(card_table, card_tf_table, card_idf_table), tf_idf_table(card_seq_
```

Figure 44: Multiple assignments

The modules were reduced, and if it was possible, only the necessary functions were imported. For example, instead of "*import math*", the *log10* function was imported with the "*from math import log10"* line. This helped to avoid the dot operation.

One of the last steps was to modify the regular expressions. They were used a lot during the tokenizing step, so it is best to use the *re.compile()* function beforehand, which we can use to transform regular expressions into pattern objects, which helps to search a pattern again without rewriting it (*Figure 45*).

```
#Original
date = r'([0-1]\d\/[0-3]\d [0-2]\d:[0-5]\d:[0-5]\d(\.\d{3}:)?)|([A-z]{3} {1,2}[0-3]?\d [0-2]\d:[0-5]\d:[0-5]\d)'

#After change
date = re.compile(r'([0-1]\d\/[0-3]\d [0-2]\d:[0-5]\d:[0-5]\d(\.\d{3}:)?)|([A-z]{3} {1,2}[0-3]?\d [0-2]\d:[0-5]\d:[0-5]\d)')
```

Figure 45: Regular expression modification

The current final time is 1290 seconds ~ 21.5 minutes; which is half of the original time, but in the "*Conclusions and future work*" some ideas were written to improve the time more.

7 Conclusions and future work

The main goal of this research was to create a System log files analyzer for predictive maintenance because there is currently no quick and effective program for this. Therefore; several publicly available programs were overviewed that are capable of individual log files, especially system log files, organizing, and analyzing them. It was investigated how they are structured and what types of logs are analyzed in what way.

Syslog files were examined with different labels in this dissertation, which were converted using text mining methods. Then they were analyzed with several models, including decision trees, random forests, the XGBoost model, and neural networks, and they predicted labels for each log file. To make this more successful, text-mining methods were applied and each Syslog was sliced into sequences. These transformations gave much more transparent and accurate results. After teaching and testing the models, the results were obtained, which were evaluated with different indicators, such as accuracy and a confusion matrix.

It was also part of this thesis the writing of a program for a Raspberry Pi that can give the professional workers guidance on what kind of failures happened in the system, thus reducing their time spent on maintenance and more efficient debugging. The program was written in Python and tested it on two different types of Raspberry Pi.

In conclusion, several types of errors can occur in a log file, and machine learning methods can significantly help the work of professionals by guiding analysis. With this AI technology, professionals know more precisely what error they need to look for in which sequence; they do not have to look through all the rows. This allows them to spend more time on improvements and upgrades, and not have to deal with maintenance.

As a personal impression it was worth doing this research. I have learned plenty of new things like neural networks, or how a text-mining project is built, how the process works in real situations. I have experienced that we always have to check whether the results are correct because many times, the results are wrong or unequivocal. Therefore, it is essential, not enough, if you have data mining skills, and you need to understand the data, in this case, how it works, how it looks like the system log files. The progress was challenging, with many failures and false results, or restarting the project from the beginning, but worth it because I learned many text-mining skills.

It turns out it is tough to clean a log file because many log files look different. It is not enough to clean the text; I had to pay attention to the fact that certain words and expressions have to be left out from the cleaning part because they are essential as in model building, and as well as for the Syslog file itself. When somebody starts to build a model, he has to pay attention to precisely what he wants to measure and how.

The task was complicated because I had a relatively small amount of data. Neural networks need a more extensive dataset, but fortunately, different models handle even the already more minor dataset well, such as decision trees and the XGBoost. I increased the dataset by slicing to sequences, but I had to review all the sequences. Therefore, these models, the results I wrote in this research are more accurate because I could eliminate the noise as much as possible.

I have several development suggestions for the future. First, the company has to write a Graphical User Interface or improve the runtime with some changes. I used the "*pandas*" module for creating and handling data frames, but if we can vectorize these tables with "*NumPy*", then the whole process could be significantly faster.

It is not a question nowadays why data mining is very popular. There are many areas where it could be used. Hidden information could be extracted from the data, and companies can gain an advantage from it. Data Science will be more in the focus of companies, as it will impact economic performance. With predictive maintenance, companies can gain a huge advantage too. They can anticipate when a device is not working correctly with a predictive model, saving time and money.

References

[1] SAS.com, *Data Mining*, https://www.ibm.com/support/knowledgecenter/en/SS3RA7_15.0.0/com.ibm.spss.crispdm.help/crisp_overview.htm*https://www.sas.com/en_us/insights/analytics/data-mining.html* (2019. May)

[2] Ray Li, *History of data mining,* *https://hackerbits.com/data/history-of-data-mining/* (2020. May)

[3] Gary Miner; John Elder; Dursun Delen; Robert Nisbet; Andrew Fast; Thomas Hill (2012. January) *Practical Text Mining and Statistical Analysis for Non-structured Text Data Applications* - ISBN 978-0123869791

[4] HP. Luhn (1958) *The automatic creation of literature abstracts.* IBM Journal of Research and Development. ISSN: 0018-8646

[5] National Research Council (1999) *U.S. Research Institutes in the Mathematical Sciences Assessment and Perspectives*, Washington DC, National Academic Press ISBN 978-0-309-06492-7

[6] Smart Vision, *What is the CRISP-DM methodology?* *https://www.sv-euro9pe.com/crisp-dm-methodology/* (2019. January)

[7] Chapman P, Clinton J, Kerber R, et al. *"CRISP-DM Step-by-step data mining guide.",* Chicago, IL: SPSS; 2000

[8] Tfidf.com, *What does tf-idf mean?,* *http://www.tfidf.com/* (2019. December)

[9] H. Wu and R. Luk and K. Wong and K. Kwok. (2008. June) *"Interpreting TF-IDF term weights as making relevance decisions"* - ACM Transactions on Information Systems

[10] Re, *Re - Regular Expression operations* *https://docs.python.org/3.8/library/re.html* (2019. December)

[11] Friedl, Jeffrey. Mastering Regular Expressions. 3rd ed., O'Reilly Media, 2009., ISBN-10: 0596528124

[12] Scikit learn, *1.10. Decision Trees* https://scikit-learn.org/stable/modules/tree.html (2020, January)

[13] J. Brownlee, *A Gentle Introduction to XGBoost for Applied Machine Learning* https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/ (2020. April)

[14] XGBoost developers, *XGBoost Documentation* https://xgboost.readthedocs.io/en/latest/ (2020. April)

[15] Mitchell R, Frank E. 2017. Accelerating the XGBoost algorithm using GPU computing. PeerJ Computer Science 3:e127 https://doi.org/10.7717/peerj-cs.127

[16] C. Lai, *XGBoost Regression* https://medium.com/@reinec/my-notes-xgboost-regression-d1992695f8fc (2020. January)

[17] Towards Data Science, *How does XGBoost Work* https://towardsdatascience.com/how-does-xgboost-work-748bc75c58aa (2020. March)

[18] R. Bekkerman, M. Bilenko, J. Langford (January 2012) *Scaling Up Machine Learning*, Cambridge University Press – ISBN: 978-0521192248

[19] I. Goodfellow, Y. Bengio, A. Courville (2016) *Deep learning*, MIT Press, ISBN: 978-0262035613

[20] A. Borovkov (April 2017) Image Classification with Deep Learning, University of Hamburg

[21] Towards Data Science, *Introduction to Artificial Neural Networks(ANN)* https://towardsdatascience.com/introduction-to-artificial-neural-networks-ann-1aea15775ef9 (2019. October)

[22] C. Nicholson, *A Beginner's Guide to LSTMs and Recurrent Neural Networks* https://pathmind.com/wiki/lstm (2019. November)

[23] Towards Data Science, *Understanding RNN and LSTM* https://towardsdatascience.com/understanding-rnn-and-lstm-f7cdf6dfc14e (2019. October)

[24] ITT Systems, *What is Syslog? A Quick Overview of Event Logging Protocol*, https://www.ittsystems.com/what-is-syslog/ (2020. January)

[25] B. Gavin, *What is Log File (And How Do I Open One)?*, https://www.howtogeek.com/359463/what-is-a-log-file/, (2019. July)

[26] R. Gerhards, *The Syslog Protocol*, https://tools.ietf.org/html/rfc5424#section-1, (2019. March)

[27] Raspberry Pi, *What is a Raspberry Pi?* https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/ (2020. February)

[28] Raspberry Pi, *Raspberry Pi 4* https://www.raspberrypi.org/products/raspberry-pi-4-model-b/ (2019. June)

[29] Lasse Espeholt, *Massively Scaling Reinforcement Learning with SEED RL* https://ai.googleblog.com/search/label/TensorFlow (2020. March)

[30] S. Ghoneim, *Accuracy, Recall, Precision, F-Score & Specificity, which to optimize on?*

https://towardsdatascience.com/accuracy-recall-precision-f-score-specificity-which-to-optimize-on-867d3f11124 (2019. April)

[31] N. Kawwa, *How to Calculate & Use the AUC Score* https://towardsdatascience.com/how-to-calculate-use-the-auc-score-1fc85c9a8430 (2019. February)

[32] K. Ping Sung, *Accuracy, Precision, Recall or F1?* https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9 (2020. September)

[33] J. Brownlee, *A Gentle Introduction to k-fold Cross-Validation* https://machinelearningmastery.com/k-fold-cross-validation/ (2019. May)

[34] X. Xu, D. Cao, Y. Zhou, J. Gao, *Application of neural network algorithm in fault diagnosis of mechanical intelligence*, Mechanical Systems and Signal Processing, Volume 141, 2020, 106625, ISSN 0888-3270, https://doi.org/10.1016/j.ymssp.2020.106625.

List of Figures

List of Tables

List of Equations