CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Anomaly Detection in Logs by Utilizing Message Occurrence Analysis

A Machine Learning Approach to Log Analysis

Master's Thesis in Computer Science and Engineering

Edin Tataragic

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Anomaly Detection in Logs by Utilizing Message Occurrence Analysis

A Machine Learning Approach to Log Analysis

Edin Tataragic

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Anomaly Detection in Logs Utilizing Message Occurrence Analysis
A Machine Learning Approach to Log Analysis
Edin Tataragic

Supervisor: Patrik Olesen, Ericsson
Advisor: Maryam Lashgari, Chalmers University of Technology
Examiner: Paolo Monti, Chalmers University of Technology

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2020

Anomaly Detection in Logs Utilizing Message Occurrence Analysis
A Machine Learning Approach to Log Analysis
Edin Tataragic
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The process of analyzing logs and errors from software tests is a vital part of the maintenance of any system. As anomalies can exist in these logs, detecting them is critical in order to ensure efficiency of the software and the future development of it. With the evolution of machine learning algorithms and computer hardware, more tools are created to automate the log analyzing process. In this thesis, we explore anomaly detection in the logs by using different machine learning algorithms, and compare their performance in terms of accuracy, detection ability, and required time to train. In addition to the comparison of different machine learning models, a new method is proposed for pre-processing the data to handle differences in the logs, which are caused by concurrency. The anomalies we aim to detect are sequential anomalies, and the parameter anomalies are not considered in this thesis. The order of the messages in the logs can change, but we proposed to observe the occurrences of each log message in different windows over the log. Our proposed method leads a long short-term memory (LSTM) recurrent neural network to be able to accept permutations of the order of the messages rather than requiring a fixed order for predictions. Applying LSTM in this way is compared with a traditional LSTM, random forest, and classification by using a transition matrix. The results show improvement in the performance by using the proposed method to apply LSTM as more anomalies can be detected while requiring less time to train. Moreover, using a transition matrix is proposed to save a lot of time for training, although, because of concurrency the number of false positives will be higher which might increase the required time to analyze the logs.

Keywords: Machine learning, neural networks, log analysis, anomaly detection, LSTM, Random Forest, Markov-chain, Transition matrix.

Acknowledgements

My warmest regards to Lucas Jönefors, Patrik Olesen, and Simon Bood for giving me the opportunity to do this thesis at Ericsson. I would also like to extend my gratitude to Patrik Olesen and Simon Bood for supervising my work, sharing their crucial domain expertise, their valuable inputs and discussions that helped guide my work. I'm also grateful to Kent Persson for his help with understanding the logs that were a core part of this work. A big thanks to my advisor Maryam Lashgari whose constructive and helpful criticism helped guide my writing and Paolo Monti for his kindness and feedback.
Finally, special thanks to my family for whose support made the completion of this work possible.

Edin Tataragic, Gothenburg, November 2020

Contents

List of Figures

List of Tables

1
Introduction

Artificial intelligence (AI) has been a popular topic among scientists and businesses alike for a long time. Today, a subset of AI called machine learning (ML), is one of the hottest topics. As machine learning gains more traction in digital businesses, it becomes a must to embrace it and use it as a tool in order to get or keep a competitive edge. One of the areas where machine learning shines compared to AI is data analysis since machine learning is more than a technique for analyzing data. It is a system that has the ability to learn and improve with the help of algorithms that provide new insights without being explicitly programmed to do so. But in order for this to be successful, a lot of data is needed. As ML systems are fueled by data, they go hand in hand with big data [1]. Some organizations with access to overwhelming amount of data are using multiple ML frameworks to increase operational efficiencies and achieve greater business agility. ML is also used to add elements of intelligence to software development and information technology (IT) operations to improve operational efficiency.
By creating ML tools to process the heaps of data available, which would take a lot of time to be processed manually by a human, companies can immensely increase the value they get from the data while decreasing the costs of processing it.

1.1 Problem Description

Ericsson is an example of a company with large quantities of data that is investing in the development of internal ML tools in order to boost the business values gained from data. One of the areas where these kinds of tools are being developed for is *log analysis*. Logs are computer-generated records, and the analysis of these records refers to the process of reviewing, interpreting, and understanding them. Logs can be generated from many different sources and one such example is from running tests on software. With the growing complexity of telecommunication systems, the generated logs from testing these systems also become more complex which makes analysing them more difficult.
Currently, the logs that are being produced in the automated testing of Ericsson's MINI-LINK [2] contains copious amounts of data. Certain tests run continuously while others only run once a day or even more infrequently. The tests are grouped into test-suites depending on the functionality and area. On average, each suite takes about 2-4 hours to run and if one test fails, the result is logged while the execution of the suites continues. The tool JCAT (Java Common Auto Tester) is used to create reports from these logs. JCAT is a feature and system testing tool used by

various Ericsson organizations and the reports created with this tool show each test step and the result of that particular test. The logs contain information about each performed test step, including errors from the test system, and other printouts from the embedded software. Other than the highlighting of some specific lines in the log where an assertion has failed, the logs can seem overwhelming. Many errors can be caused due to something that has happened some time before the actual error happened, and new unknown problems will occur with a growing and continuously updated system. Therefore, it is of interest to not only analyse the errors, but also, the logs leading to the error to find suspicious anomalies and understand differences between a passed and a failed execution of a certain test. In addition, as the order of the log message can differ between different executions of the same test due to concurrency, it is possible that this will have an effect on the predictive ability of ML algorithms.

To fully understand an error that has occurred, a lot of time is required for the engineer to understand what has caused the error and which parts of the system are affected. Since this might require insight into many different parts of the system, more resources might be needed to understand the error before solving it. Therefore, it is valuable to add automation to the log analysis process and make it easier for the engineer to understand the reason of the error.

However, it is not always the log of a failing test case that we would like to analyse. Most of the automated tests perform a number of configurations on real hardware and then compare the results with the expected results, sometimes side effects occur that are not directly tested for in the exact test case. This can cause errors which are not looked for in the aforementioned comparison. It is unreasonable to expect an engineer to analyse logs of passed test cases with no indication of where to even start looking as this would require a substantial amount of time and have a higher cost compared to the value that can be obtained. The free-form nature of the logs makes this a very challenging task. In addition to this, the order of the log messages inside the log can change in each run as several processes can run at the same time. By using ML techniques to do automated log analysis, many correlations can be learned by the ML model that result in a better log analysis.

1.1.1 Related work

Much work has been done in the field of log analysis leading to the development of many anomaly detection techniques. It is therefore possible to find numerous implementations of methods meant to aid and improve the log analysis process. As logs can contain vast amounts of data, a suitable methodology is to use machine learning models and automate the anomaly detection process as ML models go well with big data.

Some examples of different solutions are the works that use a principal component analysis based anomaly detection [3, 4], invariant mining [5] (for an overview and comparison of these methods, see [6]), and time series forecast using random forest [7, 8, 9].

Supervised methods [10], which use normal and abnormal vectors, train binary clas-

sifiers to be able to detect anomalies. Due to the usage of labeled abnormal data it is very possible for new unknown anomalies, that the models are not trained on, to pass through the analysis process undetected. As we want to be able to detect unknown anomalies, this method is not useful for us. In addition to this, it can be very difficult to collect anomalous data for the training, especially if there are many different types of anomalies that needs to be detected by the model. For this reason, it is more beneficial to not limit the types of anomalies that the models can detect. One example of the difficulty of collecting anomalies is shown in the dataset used in several papers [11, 4, 12, 13]. The dataset used in these papers contained 24 million lines of free-text logs from a 48-hour run of a production open-source application, the Hadoop File System (HDFS) [14], running on a 203-machine cluster.
This work differs from the aforementioned research as it seeks out to tackle the negative impact that concurrency in logs have on the performance of the LSTM. Additionally, we seek to achieve a high detection rate and keep a low false positive rate, while limiting the amount of logs available for training in order to reduce the time needed to train and make it possible to use in a quickly changing production environment.

1.1.2 Research questions

In this thesis we seek to provide insight for the following questions:

- Is it possible to use ML to learn what is expected from a log of a successful test and highlight the differences in a failed test log?
- Can ML be used to detect abnormalities in logs?
- Will the usage of ML reduce the complexity or required time to analyse the logs?
- How can this be used in other contexts? (e.g. customer logs, built-in software logs)
- How does concurrency in logs affect the ML models and how can we avoid it?

1.2 Limitations and challenges

There are many possible areas and implementations of ML algorithms on logs, so a defined scope is needed. Due to the massive amount of information available in the logs, we focus on the printouts regarding to the test. The setup before and all the printouts after a test case can vary to a great degree. By leaving this out, the complexity of the task can be reduced while still resulting in a useful solution.

1.2.1 Ever-changing logs

As the tests keep changing and evolving, the logs generated by the tests will also change. Therefore, a model trained on 'older' test logs might not perform well on 'new' logs. Consequently, a limit will be set for the project of how much time has passed between the test executions of the logs in the training set, that the models will be trained on, and the test set which will be used to test the models' performance. This limit will be set to one week.

1.2.2 Anomalies

There are many different types of anomalies that can be found in the logs, this project will limit the scope to sequential anomalies that are caused by either missing lines, as a result of an incomplete message sequence in the logs, or new lines, caused by errors, that differ from what is expected based on logs from a successful test case.

1.2.3 Other Challenges

A big challenge lies in the vast amounts of text that can be found in the logs. Since the logs are "raw" logs, they most likely contain a great deal of irrelevant data, and there is a possibility that some information can cause more harm than benefit in training the model. There are also many variables related to changes in parameters for the test that might differ vastly in each test. A solution to mitigate this problem is implementing various data preprocessing techniques.
Another challenge is selecting the ML model to use and then tune the hyperparameters to improve the accuracy [15]. This can be mitigated through the testing and evaluation of various algorithms.
As mentioned in section 1.1, because of concurrency, the order of the log messages in the log files can differ between various test executions. In addition, developers are continuously working on the tests that create the logs. Therefore, we are faced with two additional challenges: *(i)* the order of the log messages that appear in the logs are varying, and *(ii)* new log messages will appear in the logs. Both of these challenges can possibly have a negative impact on the predictive ability of a model. In order to handle the first challenge (i.e., the issues caused by concurrency), we need to investigate methods of how the data can be pre-processed and find a way for how the selected ML model can be adapted to said data while minimizing the effect concurrency has on the predictions. The second challenge can be handled if the models are re-trained with good enough frequency. However, the time that is required to train the model(s) needs to be taken into consideration in order to *(i)* find a balance between the time and resources that are used for training, and *(ii)* estimate the duration that the model(s) can be used as predictors.

1.3 Sustainability and ethical aspects

As with other research, it is important to analyze and reflect the direct and indirect affects the work can have on different aspects of our society and environment.

1.3.1 Societal effects

As more technological advancements are made in the fields of machine learning and AI in general, we move towards a time where the workforce for certain professions can be reduced in favor of adding some AI. The affect of this on the society is comparable to the impact that machines have had on the manufacturing industry. The automotive industry is one prime example. Some may see this as a positive step forward as AI and ML are mostly used to do menial tasks, while others fear the prospects of losing their jobs. This is an ongoing debate [16, 17, 18, 19] whose discussion is beyond the scope of this thesis.

In regard to log analysis, it is not very likely that the current methods will completely be replaced by ML models. Instead, the ML models will complement current methods and make it easier for the engineers to find bugs and errors.

Since log analysis is used to find bugs and other errors, creating and using faulty tools can lead to bugs being missed. This, in turn, can cause the final product to ship with bugs. Today we rely heavily on telecommunications for everything from entertainment to emergency services. In the case of emergency services, faulty software can potentially cause loss of life. But as previously mentioned, it is highly unlikely that a faulty ML analysis could cause such drastic effects on the society. Unless, of course, if the current methods were to be completely replaced by a faulty ML implementation. If this were the case, then there is a good chance that buggy code could pass through undetected and therefore be used in applications where the bugs could prove harmful to the society.

There are also many other positive applications of anomaly detection, like for instance in fraud prevention and intrusion detection in security systems.

1.3.2 Ethical Aspects

The information in the logs are from automated test on software and hardware. This data contains no personal or identifiable personal information. Therefore, there are no privacy concerns regarding the handling of data from/about humans.

1.3.3 Ecological Aspects

As the possible applications of computers increase, so does the energy consumption. Whether it is for mining bitcoins, running complex protein folding simulations, or training machine learning algorithms, the public power grid is getting more taxed. In recent years, more companies have started to utilize AI and ML. The increase in power consumption shows that more renewable energy production is needed to avoid the possible climate-related consequences. Due to the scope of this thesis being relatively small scale, the negative effects are negligible.

1.3.4 Economical Aspects

The effect of anomaly detection systems on organizations is positive. As mentioned before, they can increase the business value that can be gained from large amounts of data. This in turn results in a monetary gain. Other applications of anomaly detection systems, like fraud detection, can prevent monetary loss.

1.4 Contribution

The logs that are used in this project change over time, as mentioned in section 1.2.3. This project's contribution is a new method of applying a Long Short-Term Memory, LSTM, to avoid the negative effect of the different logs sequences that can appear. By using a sliding window approach and only considering the occurrences of each log message in a log, we are able to completely ignore the issue of different sequences caused by both concurrency and a change in the test. The dataset created based on this approach consists of each window's observations and the LSTM is applied on this dataset as a time series predictor where a window of log message occurrences is predicted based on the previous 3 windows. This approach allows us to improve the predictive ability of the LSTM in regard to anomaly detection on logs whose log messages can appear with different order over time. The experiments compare the performance of this methodology with a 'traditional' LSTM approach, where each prediction is based on x previous observed datapoints, as well as, the usage of a random forest classifier and a transition matrix classifier. Two important metrics in these comparisons will be the true positive rate and the true negative rate as the number of correct classifications that the model does has great importance.
The proposed method proves to work better than the other models that were compared with it and requires less time than the 'traditional' LSTM to train. In instances where new log sequences are present, the proposed method's predictive ability stayed unaffected while the other models had several false positive predictions.

1.5 Thesis structure

The rest of the thesis adheres to the following structure. Chapter 2 introduces relevant theory and concepts that are required to understand the work done in this project. Chapter 3 provides a description of the methods used to answer the questions mentioned in section 1.1.2. Chapter 4 presents the results gathered from testing and evaluating the models and methods described in chapter 3. Chapter 5 contains a discussion and conclusion based on the results gathered while also offering some ideas for future work that can be done.

2
Theory

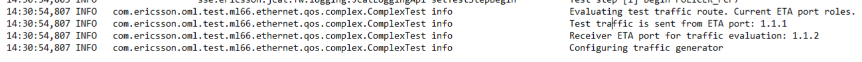
In this section, theory relevant to this project is presented to give readers an understanding of the logic used behind the choices made. We first introduce the state of the logs used in this project, methods for parsing them and how they can be used by a ML algorithm. After that, the definition of anomalies used in this project are presented. Next comes a brief description of how neural networks, NNs, handle sequential data and why the Long Short-Term Memory model is superior in this use case. Two classifiers are then presented and their performance will be tested as well. In addition to the algorithms and classifiers, some metrics used for comparing their performance is presented. Finally, a method of improving the readability of the logs is presented.

2.1 Logs

Logs refer to files that contain the records of one or multiple events. These events can be that of an operating system, software, hardware, or whatever the implementer chooses. There are standards for these types of message logging and one of these standards is *syslog*. By using a standard like syslog, each message is labeled with a facility code that indicates what software type generated the message. Some examples of this are "0" for kernel messages, 1 for user-level messages, and 14 for log alerts. By doing this, it is possible to add separation of the system storing the logs and the software generating them.

2.1.1 Logs used in the project

The logs that have been used for this project are the test logs from tests run on Ericsson's MINI-LINK™. The logs from these tests come in the shape of a *console log* which, in many ways, is similar to the aforementioned syslog as the previously described separation is present. These console logs contain the following information: timestamp - Level/severity - Component - Level2 - Content. But these console logs contain not only the logs from the test cases but also from the system setting up and initializing the hardware. The console log as shown in Figure 2.1 is structured and stored as plain text.



```
14:30:54,805 INFO                  sse.ericsson.jcat.fw.logging.JcatLoggingApi setTestStepBegin        Test step [1] begin POLICER_PCP7
14:30:54,807 INFO   com.ericsson.oml.test.ml66.ethernet.qos.complex.ComplexTest info                   Evaluating test traffic route. Current ETA port roles.
14:30:54,807 INFO   com.ericsson.oml.test.ml66.ethernet.qos.complex.ComplexTest info                   Test traffic is sent from ETA port: 1.1.1
14:30:54,807 INFO   com.ericsson.oml.test.ml66.ethernet.qos.complex.ComplexTest info                   Receiver ETA port for traffic evaluation: 1.1.2
14:30:54,807 INFO   com.ericsson.oml.test.ml66.ethernet.qos.complex.ComplexTest info                   Configuring traffic generator
```

Figure 2.1: Example log

There is certain run-time information such as a port number or Internet Protocol address (IP address) that can differ from run to run. This means that the same log message from the same test case can differ between runs. These variations will cause an increased complexity in the anomaly detection and are not a part of this project. These run-time variables are redundant as they do not relate to the sequence of the logs, and they can be omitted from the log messages when creating the dataset.

2.1.2 Parsing

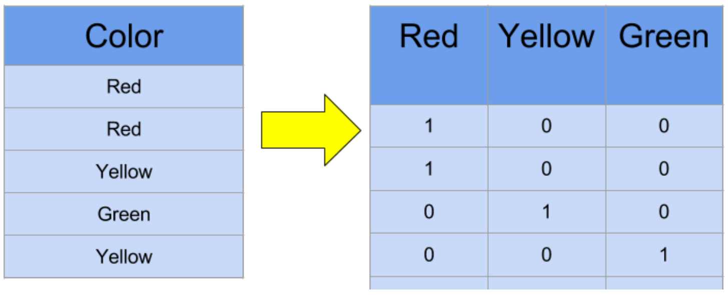
Performing data analysis on these logs, as is, would greatly complicate the task as every row is a string containing different feature sets. By parsing the free-text log entries into a structured representation, with each feature separated, a sequential model over the structured data can be learned. The parsing can also be used to extract the redundant run-time information and result in a file containing simpler log messages.

2.1.2.1 Drain

LogPai's Logparser is the toolkit that was used to parse the logs. LogPai's Logparser can learn event templates from unstructured logs and convert raw log messages into a sequence of structured events. This is sometimes referred to as *message template extraction*, *log key extraction*, or *log message clustering.*
Drain is one of the logparsers available in the aforementioned toolkit, and it is one of the most accurate and efficient open-source online logparsers [20, 21]. It is also possible to fine-tune the parsing by formulating several regular expressions to aid in the parameterization of run-time variables. With carefully created regular expressions that fit the data well, many if not all of the run-time variables that are seen redundant can be omitted resulting in a better dataset. With every variable found in the log messages extracted and the remaining content sorted according to predefined columns, a dataset can be created. The difference this makes to the dataset is best explained with a simple example: given two log messages "Connecting to IP: 1.1.1" and "Connecting to IP: 1.1.2", by extracting the variable part, the IP address, both messages will look like this "Connecting to IP: <*>". Given a similarity threshold to fit our needs, these messages will be considered the same. When training a model on the dataset, less training will be needed as there will be less messages to train on and the reduction in unique log messages makes it possible to use a less complex model.

2.1.2.2 One hot encoding

In order to produce discrete outputs from this discrete classification problem, *one hot encoding* needs to be applied. One hot encoding is a process by which all categorical features with n distinct values are transformed into equally many binary features. This is because ML algorithms require all input and output variables to be numeric. In this strategy, each category value is converted into a new column and assigned a 1 or 0 (notation for true/false) value to the column. an example of this is shown in figure 2.2.


Color
Red
Red
Yellow
Green
Yellow

Red	Yellow	Green
1	0	0
1	0	0
0	1	0
0	0	1

Figure 2.2: Example of one hot encoding. [22]

In this example there are three nominal groups, red, yellow, and green, and the columns follow the same order. If a datapoint is 'red' then the first column is populated with a one and the other ones are populated with zeros.
Using the same principle, each unique log message can be considered its own category and with K unique log messages there will be K classes, and columns.
The benefits of the parsing is now even more obvious as the total amount of classes can be reduced with the help of the parameterization described before.
Another way one might encode the categories is with *label encoding*. Label encoding would encode the above categories as:

- 'Red': 1
- 'Yellow': 2
- 'Green': 3

But due to the ordered relationship of integer values, the predictions of the ML algorithms will be affected as this can be learned through training. The integer values have a natural ordered relationship between each other and machine learning algorithms might be able to understand and harness this relationship. The training can also result in a higher categorical value being interpreted as better or that there will be predictions between categories, for instance, a prediction of 1.5 which would be between the categories '1' and '2'.
The data used here does not have any ordinal relationship and the predictions should not be in between categories. The label encoding is therefore not a suitable method of encoding and one hot encoding is preferred.

2.2 Types of anomalies

Anomalies in the test logs are considered to be patterns or characteristics which do not follow the normal behavior of a successful test. Any part of a log file that is deviating from the norm set by previous passing executions is considered an outlying observation. These deviations can be in the form of what we will call a sequential anomaly or a parameter anomaly.

2.2.1 Sequential anomalies

Sequential anomalies are log entries which do not conform to the standard sequence a log is expected to have based on logs of previous passing executions. An example of

this is that for instance: after a connection attempt is made, a successful execution would log the message "Connection successful".
This sequence would have the following flow:

1. "Attempt to connect"
2. "Connecting to IP: 1.1.1"
3. "Connection successful"

An anomaly in this example could be that the connection fails and the third message in the sequence is different, changing the flow to:

1. "Attempt to connect"
2. "Connecting to IP: 1.1.1"
3. "Connection failed"

This is an example of what is called an *insertion* anomaly as the anomalous message is inserted into the flow. Other than the insertion anomaly, the sequential anomalies can come in the shape of a *deletion* anomaly.
Insertion anomalies: If an error has occurred, then this would be documented in the logs. An error would cause one or more new lines to be added to the log which would not be present in a log of a successful execution. This sort of message, that has been inserted into the regular sequence, is classified as an insertion anomaly. The example given above will be referred to as an insertion anomaly.
Deletion anomalies: On the other hand, if an error has caused certain steps to be missed, or if part of a test has not been run due to an error, this will cause lines to be missing from the log. Compared to a log of a successful execution, this one will have one or more lines missing causing what we will refer to as a deletion anomaly. Both of these anomalies would evidently cause the same issue, a new and unknown sequence of logs that do not match with what has previously been observed.

2.2.2 Parameter anomalies

If the characteristics of a log message are wrong, such as wrong values in the message of the log, then this will be considered a parameter anomaly. If the same IP address is always used, then, by using the example above, the message "Connecting to IP: 1.0.1" would be considered a parameter anomaly. This type of anomaly is out of the scope of the project.
In order to detect the anomalies mentioned above, two LSTM implementations and two classifiers will be tested. In the following section, an explanation is given to why the LSTM was chosen in comparison to a regular neural network and even a recurrent neural network.

2.3 Neural Networks

Inspired by the neurons of a brain, a neural network (NN), is a ML model comprised of similar neurons (or nodes) that adhere to a mathematical formula in order to give an output. This formula contains variables that can be learned and with this the model is able to learn a behavior depending on previous inputs it has seen [23].
A feed-forward network consists of several of these neurons in three layers. The layers are called the input, hidden, and output layer. In this sort of network, the

neurons are fully connected, meaning that each neuron in a layer has its output connected to all neurons in the next layer. The flow of information is from the input layer, through each hidden layer, and finally to the output layer which produces the output of the NN.
This type of network can be used for supervised learning tasks, which are tasks where the network is trained to predict target outcomes that are known beforehand [24]. During the training of a supervised learning model, each input that is given is assigned a label. The prediction made based on the input is compared to the actual label and a loss function, that measures how far off the prediction was, is used to facilitate the learning process.
As the data used in this thesis is sequential, the following subsections will cover the downsides of how NNs handle this data and two possible 'alternatives'.

2.3.1 NN and Sequential Data

A given for NNs is that the data samples are independent. This is fine for many applications, but when dealing with time dependant data, this does not hold true. For many time dependant things like time series, videos, or languages, the individual samples hold some dependency to time. By using a regular NN, which will treat the samples as independent, to predict future data point will lose the value of analyzing the sequential information. Also, in several of these fields, the input sequences can vary in length which is something that a regular NN cannot handle [25].

2.3.2 Recurrent Neural Networks

Recurrent Neural Networks, or RNNs, are NNs that are capable of extracting the sequential information of data points [25]. It can do this by using a state vector in the hidden layers and with this keep a memory of all previous elements. A simple example of this can be seen in figure 2.3.

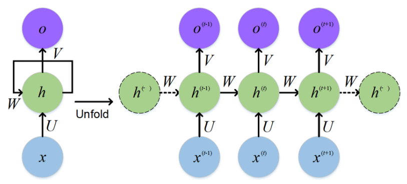


Figure 2.3: A RNN [26]

RNNs are able to keep the hidden neurons connected across time. At a specific time t, the model takes the current input x_t and the previous hidden state s_{t-1} for its calculations. The next hidden state s_t is calculated along with the output h_t and the hidden state is passed on to the next time step. Equations 2.1 and 2.2 summarize these calculations [27, 28].

$$s_t = \sigma(Ux_t + Ws_{t-1} + b_s) \tag{2.1}$$

$$h_t = softmax(Vs_t + b_h) \tag{2.2}$$

This way, RNNs can selectively retain relevant information and capture dependencies across several time steps. RNNs can also handle variable length sequences, meaning that they are better than regular NNs when using time dependant data.
The caveat with RNNs and time series is that they suffer from short-term memory, meaning that if a sequence is long enough, they will have a hard time carrying information from earlier time steps to later ones. From a long sequence of data, a RNN might leave out important information from the beginning of the data sequence. This is because the training is done through back propagation through time, BPTT [29, 30], which is the version of backpropagation [31] that is used for RNNs. As seen in figure 2.3, a RNN unfolds into a deep feed forward network that has many layers [26]. As a gradient is multiplied backwards through all of these layers, or time steps, it will tend to increase or decrease. The two extremes of this will cause the problems known as the vanishing gradient problem and the exploding gradient problem.
Vanishing Gradient Problem: There is a certain phenomenon that can occur during a RNNs back propagation [32] called the vanishing gradient problem. Gradients are values used to update a neural networks weights and this problem refers to it exponentially shrinking to 0 as it back propagates through time. If the gradient value becomes extremely small, it will not contribute to the models learning. This then leads to a model unable to learn correlation between temporally distant data points. Not all gradients might vanish, the ones local in time will still be present, but the gradient will not contain long term information [33].
Exploding Gradiant Problem: As shown by Benigo et. al. [34], it is possible for error gradients to accumulate which can result in very large gradients. The consequence of these large gradients is that, eventually, they will cause the network to become unstable as a result of the network weights getting large updates. Instead of exponentially shrinking to 0, like in the vanishing gradient problem, the gradients exponentially grow as the gradients repeatedly get multiplied through the network. Also unlike the vanishing gradient problem, here all gradients will explode. This is because now, instead of some gradient $g + 0$ which will equals to g, we have some gradient $g + \infty = \infty$.

2.3.3 Long Short-Term Memory

A LSTM model is a type of RNN. With it's introduction and inclusion of Constant Error Carousel (CEC) units by S. Hochreiter and J. Schmidhuber [35], and in combination with the forget gate introduced by F. Gers et. al. [36], the vanishing gradient problem could be solved. These aspects of the LSTM are what makes it so useful for these kinds of applications. The way LSTM overcomes this problem is by having no repeated weight application between internal state t and $t - 1$. A regular RNN, the derivative of an activation function during back propagation will be less than one. So over time, repeated multiplications of this value will lead to a vanishing gradient. In LSTM, the forget gate acts both as the weights and the activation function for the cell state and it is possible for the information of the previous cell state to pass through unchanged. An equation for this will be shown a bit further in this text to help explain this.

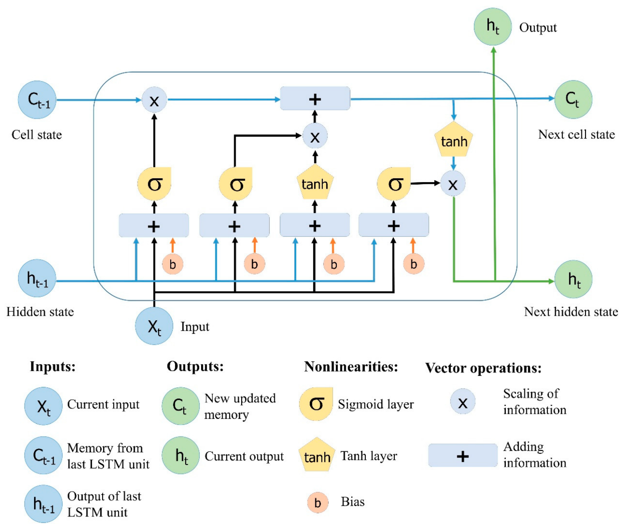


Figure 2.4: The internals of a LSTM [37]

Above, in figure 2.4, a visualisation of the LSTM can be seen. The top most horizontal line is the cell state, and at the start of it is a so called *gate*. The gate consists of a sigmoid layer and a point wise multiplication operation and there are several more in the LSTM. The sigmoid layer outputs numbers between zero and one. If the value is a zero, then nothing will be let through the gate while a one would mean that the whole component is let through. This very gate at the start of the cell state input is the forget gate and it was introduced by F. Gers et. al. [36]. The forget gate is used to decide what information is going to thrown away from the cell state which allows the model to reset its own state [38]. Equation 2.3 is the forget gates formula.

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \tag{2.3}$$

After deciding what is to be kept from the cell state, the LSTM decides what new information is going to be stored in it. Looking from left to right in figure 2.4, the second sigmoid layer is called the input gate layer and it is used to decide what values will be updated according to the equation 2.4. New candidate values are created in the tanh layer, see equation 2.5 and then these two are combined and the cell state is updated according to equation 2.6 [35].

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \tag{2.4}$$

$$\tilde{c}_t = tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{2.5}$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \tag{2.6}$$

Lastly, the output is calculated by first selecting what parts of the cell state will be in the output with another sigmoid function, equation 2.7. After that a tanh is used to push the values of the cell state between -1 and 1 and this is combined with the output of the sigmoid. The result is also the next hidden state, equation 2.8

$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \tag{2.7}$$

$$h_t = o_t \circ \sigma_h(c_t) \tag{2.8}$$

Almost all state of the art results based on recurrent neural networks are achieved with LSTMs. LSTMs can be found in speech recognition, speech synthesis, and text generation. They can even be used to generate captions for videos.

2.4 Classifiers

In addition to the ML algorithm, two classifiers will be tested as anomaly detectors as well. These two are the random forest classifier, and the Markov chain classifier.

2.4.1 Random Forest

A random forest classifier (RFC) [39], is known as an *ensemble learning method.* An ensemble learning method makes use of numerous learning algorithms in order to gain an improved predictive performance as opposed to that which could be obtained from any of the individual algorithms by themselves [40, 41]. In the case of the RFC, it is the multiple *decision trees* used that makes it an ensemble learning method. An example of a decision tree can be seen in figure 2.5, and it is a very intuitive decision support tool to understand.

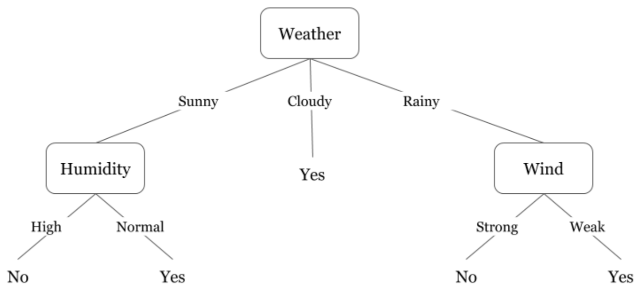


Figure 2.5: A decision tree for the concept "Go Outside" [42]

A decision tree is a tree-like graph with nodes representing the place where an attribute is picked and a question is asked. Edges represent the answers to the question, and the leaves represent the actual output or class label. They are used in non-linear decision making with simple linear decision surface. By adding together several of these tree-like graphs, we can create a "forest" of them where they operate as an ensemble. Each individual tree in the random forest outputs a class prediction and the class with the most votes becomes the model's prediction. An example of this can be seen in figure 2.6.

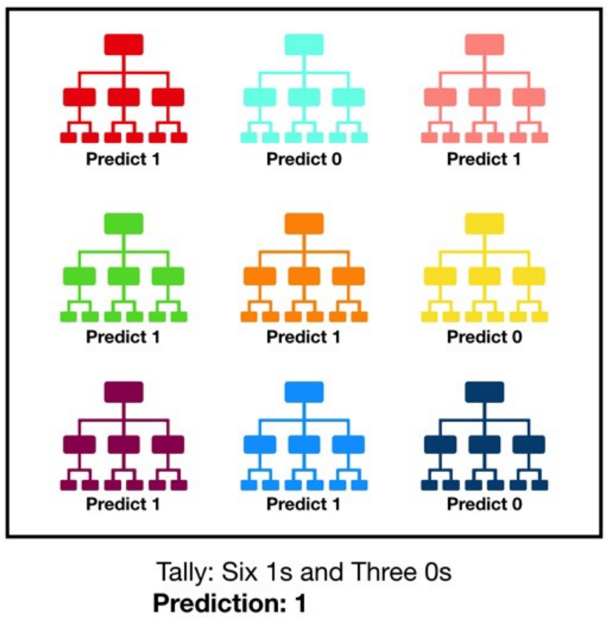


Figure 2.6: Visualisation of predictions using Random Forest [43]

An issue with the RFC is that it has no awareness of time as each observation is assumed to be independent and identically distributed, which is not similar to the serial dependence seen in time series data. Decision trees are not able to extrapolate and understand the trends in data. They use if-then rules based on the given inputs during training and due to this attribute of the RFC, they can not make predictions for values that fall outside the range of the values in the training data.
Even though this is the case, there are methods of pre- and post-processing of data in order to make the RFC work better for time series.
Some examples are:

- Statistical transformations (Box-Cox transform, log transform, etc.)
- Detrending (differencing, STL (Seasonal and Trend decomposition using Loess), SEATS (Seasonal Extraction in ARIMA Time Series), etc.)
- Time Delay Embedding

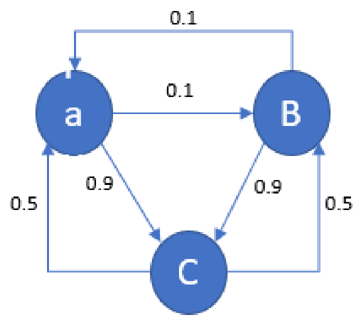
There are some works having good results in time series predictions using random forest classification and regression. [8, 44, 7, 45]

2.4.2 Markov Chains

Markov chain models are based on learning a transition matrix from training data [46]. If a log is considered as a Markov chain and with any sequence of 2 adjacent log messages considered a sequence, then each log message is considered a state and the probability of going from one state to the other is dependant only on the previous state. A stochastic matrix, or transition matrix, can be created with the probability of each transition. The basis of the probabilities can be from what is observed in the dataset. Here, the individual transitions $P_{ij} = P(X_{k+1} = j|X_k = i)$ are estimated by summing all transitions and dividing each sum by the total amount of times that log has been observed.

1. Let n_i be the number of times message i is observed in the sequence $\{X_1, ..., X_{n-1}\}$ (exclude the last message).
2. Let n_{ij} be the number of times we see message i go to message j. For the last message, there is no transition so that value is not incremented.
3. Then $P_{ij} = \frac{n_{ij}}{n_i}$

In this way, a transition matrix is created where each element $P_{i,j}$ is the maximum likelihood estimate for the probability that log message j directly follows log message i. [47, 48, 49]
A very basic example of a Markov chain and its corresponding transition matrix can be seen in figure 2.7



$$P = \begin{bmatrix} 0 & 0.1 & 0.9 \\ 0.1 & 0.0 & 0.9 \\ 0.5 & 0.5 & 0.0 \end{bmatrix}$$

Figure 2.7: Example of a Markov chain and transition matrix [50]

New log messages in transitions that we have not observed yet will have a $P_{i,j}$ of 0, resulting in a possible anomaly detection.
Using the example in section 2.2.1 the following flow is considered okay:

1. "Attempt to connect"
2. "Connecting to IP: 1.1.1"
3. "Connection successful"

Adding the possibility that the connection can be retried after a connection attempt without getting the "Connection failed" message, the following flow is also considered correct as it does not lead to a test execution failing:

1. "Attempt to connect"
2. "Connecting to IP: 1.1.1"
3. "Connection timed-out"
4. "Attempting to re-connect"
5. "Connecting to IP: 1.1.1"
6. "Connection successful"

If the "Attempting to re-connect" message was observed 5 times and the "Connection successful" message was observed 95 times, then the transitional probability from message "Connecting to IP: 1.1.1" to "Attempting to re-connect" is 5% and from "Connecting to IP: 1.1.1" to "Connection successful" is 95%. The probability of transitioning from message "Connecting to IP: 1.1.1" to "Connection failed" is therefore 0 and if this transition is observed in a log then according to the transition matrix, it is an anomaly.

2.5 Evaluation techniques

In order to properly evaluate different approaches to detect anomalies in the logs and make an informed decision on which one to further develop, different evaluation techniques will be used. For this purpose, *confusion matrices* [51] and the *F1-scores* [52] of the models predictions will be used.

2.5.1 Confusion Matrix

A confusion matrix has two dimensions, "actual" and "predicted", and is used to visualize if the system is confusing two classes by mislabeling one for the other. From the confusion matrix, several interesting rates can be calculated, two that are of interest are the *precision* and *recall*. Precision refers to how often the system is correct, or what proportion of positive identifications was actually correct. Recall, on the other hand, is the true positive rate or what proportion of actual positives was identified correctly.

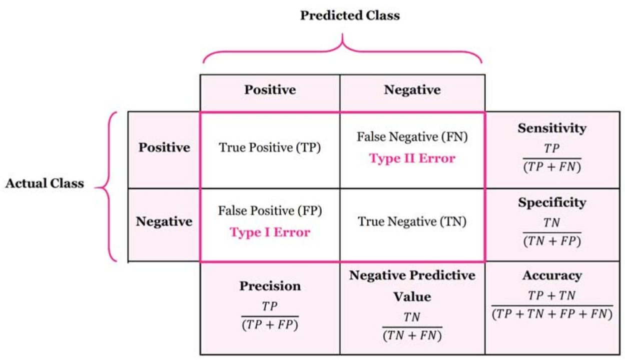


Figure 2.8: A confusion matrix and its equations [53]

2.5.2 F1-Score

The F1-score [54] is the harmonic mean of the precision and recall, where a score of 1 represents perfect precision and recall and 0 is the worst case scenario.
The F1-score if calculated using equation 2.9 [55].

$$f_1 = 2 \times \frac{precision \times recall}{precision + recall} \tag{2.9}$$

One issue with the F1 score is that it gives equal importance to precision and recall. These two metrics do not always have the same importance as the price of misclassification can differ depending on the application. In this application the True Negative rate has a big importance as it indicates how many of the anomalies in a log were found and as the F1-score ignores the True Negatives, it will be considered a less important metric.

2.6 Comparing logs using flow analysis

As logs can be difficult to read for humans, displaying the found anomalies in a very readable manner is of interest. By doing this, the engineer responsible to fix whatever error the anomaly has caused will have an easier time to understand the cause of the anomaly. The ease of understanding the logs can be furthered by displaying the expected behavior in addition to the location of the anomaly. One way of achieving this is by displaying the flow(s) that the anomaly occurred in. Each log message is the result of a printout from the code but a process can cause several of these. As several processes can be ran at the same time, this concurrency can cause the log messages of the different processes to be intertwined leading to an increase in the time needed to analyse a log.
By analyzing each log and trying to find the processes that are causing the log messages, it is possible to find each process' print sequence, or sub-flow, in the full flow of log messages. By displaying only the flows that contain any found anomaly, the whole log analysis process could be simplified.
Continuing on the example in section 2.4.2, given a failed connection attempt the expected flow would be "Attempt to connect" -> "Connecting to IP: 1.1.1" -> "Connection successful" while the gotten flow would have been "Attempt to connect" -> "Connecting to IP: 1.1.1" -> "Connection failed". Given a situation where this connection attempt was done at the same time as another test step, these messages would have been mixed with the other test step meaning that it would have been more difficult to not only find this flow, but see the cause of the error. The benefit of this would be even more clear with a more complex flow.

3
Methods

In this section, the methodology used to find answers to the questions in section 1.1.2 will be presented. We first introduce the log processing, then describe the details of two proposed LSTM approaches, a random forest approach, and a Markov-chain approach. After that, the method of finding the flows in the logs is presented. Finally, we describe our proposed method for evaluating the models.

3.1 Log Processing

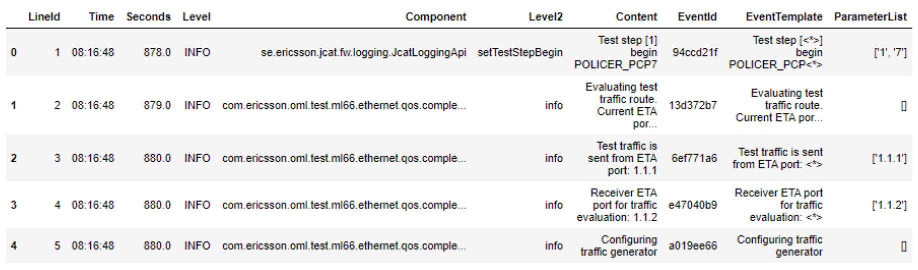
In order to create the dataset, one of the test cases in a given test suit was selected. After that, a multitude of logs were collected. The collected logs were the results of many runs of the selected test case and these runs ranged over a time span of several weeks. The dataset consisted of log from both passing and failing executions and none of the logs from failed executions were used for training.
Succeeding the completion of the log collection, as mentioned above, each log in the dataset was put through the log parser Drain as it has shown to be a very good tool for this purpose [21]. The output from the parsing of the logs was analyzed and the results were perceived as sub-par due to the parser overlooking several variables that needed to be extracted.
This oversight is problematic as it will lead to multiple log messages which are similar in nature, to be considered unique even though this is not the case. In order to further improve the parsing, a manual review of the parsing results was done, and based on these findings regular expressions were created to most of the variables that were printed by the system. These regular expressions were then added as an input to Drain. Adding these regular expressions was also done with the goal of further simplifying the dataset. An example will make this simplification clear:
Given the strings "Test step (1) start TestStep1" and "Test step (2) start TestStep2", they would initially be classified as unique by Drain. Putting aside the variable, showing which test step it is, and the name of the test step, these messages can be seen as one type of message: the message indicating the start of a test step. By formulating regular expressions to match the number in the parenthesis and the name of the test step, Drain would no longer overlook these parts of the string and therefore extract them during the parsing. A template would then be created by drain to match any variation of this message that would look like this: "Test step(<*>) start <*>". This analysis of the parsing and addition of regular expressions was done for many of the variables that Drain initially overlooked and the result of this was a heavy reduction of "unique" messages in the logs.

Using the given example above, all test log messages that indicated the start of a test step would be parsed as 1 message/template instead of 1 for every test step. The same is true for "Test step (1) end TestCase1". In a log consisting of 7 test steps, the sum of the starting and ending messages for these test steps was 14. Indeed, this is just two types of log messages, one for the start of a test step, and one for the end of the test step. With this abstraction of variables, the number was reduced to the correct amount of two.

Here follows some examples of certain regular expressions that were used to help Drain parse the logs:

```
r'blk_(|-)[0-9]+' , # block id
r'(/|)([0-9]+\.){3}[0-9]+(:[0-9]+|)(:|)', # IP
r'([0-9]\.)+([0-9])', # Port
r'([0-9]+[^\S\t\n\r])+([0-9]+)', # MAC addresses
r'(?<=[^A-Za-z0-9])(\-?\+?\d+)(?=[^A-Za-z0-9])|[0-9]+$', # Numbers
r'<null>', # null values
r'false',  # false
r'"([a-zA-Z]+)([^\S\t\n\r]|)([a-zA-Z]+)"', # Quoted w&w/o spaces
```

After a log had been parsed, Drain created two files of comma separated values, CSV. One of these files had the suffix *structured* while the other one had the suffix *templates*. The structured file contained all rows of the log with the parameters abstracted as asterisks and an example of this is shown in figure 3.1. In addition to this, every line had been given an *event id* and the abstracted parameters were stored in their own column.

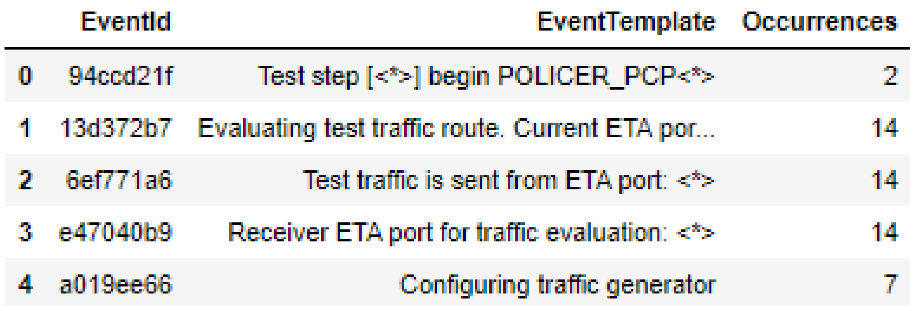


	LineId	Time	Seconds	Level	Component	Level2	Content	EventId	EventTemplate	ParameterList
0	1	08:16:48	878.0	INFO	se.ericsson.jcat.fw.logging.JcatLoggingApi	setTestStepBegin	Test step [1] begin POLICER_PCP7	94ccd21f	Test step [<*>] begin POLICER_PCP<*>	['1', '7']
1	2	08:16:48	879.0	INFO	com.ericsson.oml.test.ml66.ethernet.qos.comple...	info	Evaluating test traffic route. Current ETA por...	13d372b7	Evaluating test traffic route. Current ETA por...	[]
2	3	08:16:48	880.0	INFO	com.ericsson.oml.test.ml66.ethernet.qos.comple...	info	Test traffic is sent from ETA port: 1.1.1	6ef771a6	Test traffic is sent from ETA port: <*>	['1.1.1']
3	4	08:16:48	880.0	INFO	com.ericsson.oml.test.ml66.ethernet.qos.comple...	info	Receiver ETA port for traffic evaluation: 1.1.2	e47040b9	Receiver ETA port for traffic evaluation: <*>	['1.1.2']
4	5	08:16:48	880.0	INFO	com.ericsson.oml.test.ml66.ethernet.qos.comple...	info	Configuring traffic generator	a019ee66	Configuring traffic generator	[]

Figure 3.1: Structured file.

Though similar to the actual log file, the structured file differs as it contains the columns titled 'EventId', 'EventTemplate', 'ParameterList', and additionally, the content of the structured file is sorted in columns. This is useful when extraction of any data is needed.

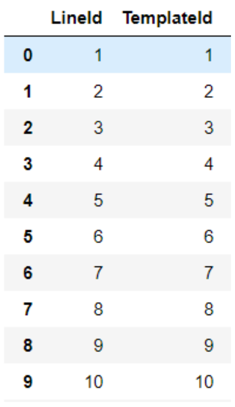
The content of the template file is also sorted in columns, but unlike the structured file, it is much shorter than the actual log file. This is because it only contains each unique log message once, and the log message stored in this file is the template of said message. Other than the singular instance of each unique log message, it also contains an event id and the number of occurrences of each log message. An example of the templates file can be seen in figure 3.2



	EventId	EventTemplate	Occurrences
0	94ccd21f	Test step [<*>] begin POLICER_PCP<*>	2
1	13d372b7	Evaluating test traffic route. Current ETA por...	14
2	6ef771a6	Test traffic is sent from ETA port: <*>	14
3	e47040b9	Receiver ETA port for traffic evaluation: <*>	14
4	a019ee66	Configuring traffic generator	7

Figure 3.2: Templates file.

The identifier "EventId", which can be found in both the structured and the templates files, is unique to every template/log message. In the templates file, only one of each identifier can be found due to the nature of the templates file. For the structured file, each identifier can occur several times as the system can output the same type of log several times and this is reflected in the structured file.
A simple integer compared to the unique identifier, which consists of a combination of numbers and letters, can serve the same purpose equally well in this application, but the "EventId" is quite complex in comparison with that. With this in mind, the unique identifier was changed from a combination of numbers and letters, to a simple unique integer. The contents of the structured and templates files were loaded as *pandas dataframes*. A new column was added to the templates content for the new identifier. The two dataframes could then be merged based on their 'EventIds' in order to map the new identifier to each line in the log.
By dropping all columns except for the LineId and TemplateId, a new simplified dataframe could be obtained which can be seen in figure 3.3.



	LineId	TemplateId
0	1	1
1	2	2
2	3	3
3	4	4
4	5	5
5	6	6
6	7	7
7	8	8
8	9	9
9	10	10

Figure 3.3: Dataframe for 1 file

By considering the content seen in figure 3.3 as the data for each log, a new dataset was created that was much simpler with the possibility of normalizing the data if needed as it is easier to map positive integers to values between zero and one than a combination of integers and characters.
Using this new dataframe, each log file can be visualized through a plot where the x-axis corresponds to what line the log message is in the log and the y-axis

corresponding to the log message type (TemplateId). An example of such a plot can be seen in figure 3.4. Naturally, as the new identifier is assigned incrementally, the value on the Y-axis will generally increase the deeper we go in the log and more new log messages we see resulting in a generally deterministic, but still, stochastic trend.

With time, and as more logs are being added to the dataset, new logs would contain new log messages that were not present in the logs of previous executions. This was due to the fact that new log messages are created as the tests are developed, and these new log messages need to be assigned a new unique identifier as well. This was solved by simply mapping each new log message to a integer that is bigger than the previous largest identifier for every new message. Given the latest identifier x, the identifier of the new log message would then be $x + 1$. The new log messages are not restricted to only show up at the end of the logs and could therefore be on any line of the log. As these log messages could occur anywhere in the log, the plot of the dataframe of a log would contain spikes in the places where the identifier would be much bigger than the identifiers of the messages surrounding the new one. This further increased the stochastic nature of the logs although it had no effect on the performance of the models, but did reduce the linearity of the plots.

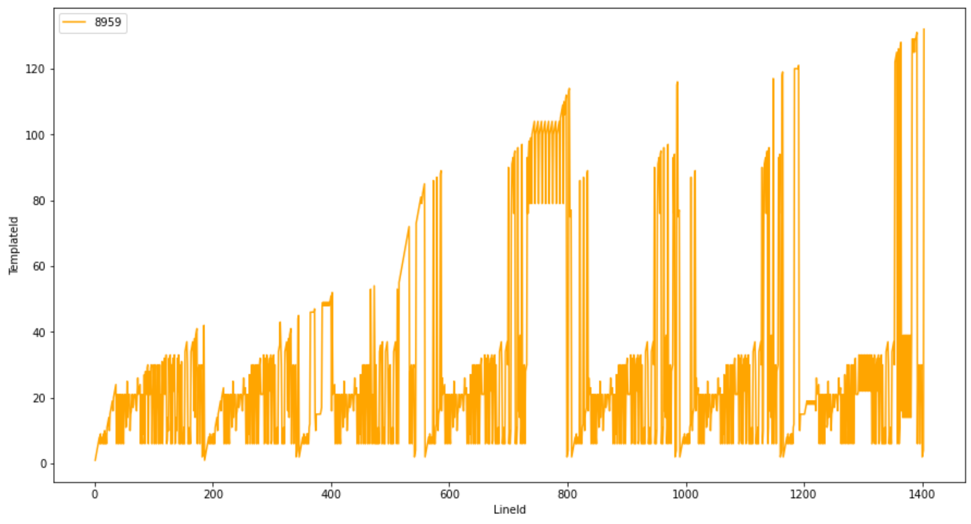


Figure 3.4: Plot of LineId and TemplateId

This process of parsing, changing identifier, and merging tables was then repeated for every file that would be in the dataset.

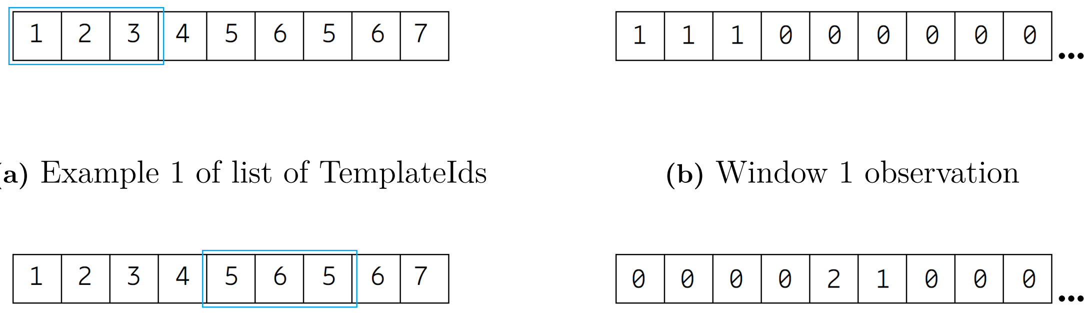
After the parsing each log file was completed, the data had to be pre-processed. The pre-processing was done by dividing the dataframe of each log into input and output signals. The input signals would be given to the model and the model was expected to give the output signal. By doing so, this task becomes a supervised learning task.

The way that the data was divided depends on what model is used. Further details for division of data can be seen in the respective method of each model.

3.2 Algorithms

3.2.1 First LSTM model

The process of creating the dataset for the first LSTM model used a sliding window approach with a stepsize of one. The sliding window was used to iterate over the logs and for each iteration, the number of occurrences of each message type in the window was stored. This process is visualized with two examples in figure 3.5.



(a) Example 1 of list of TemplateIds **(b)** Window 1 observation

(c) Example 2 of list of TemplateIds **(d)** Window 2 observation

Figure 3.5: Example of what the sliding window sees.

Figure 3.5 shows an example of the aforementioned window iteration where figures 3.5a and 3.5c represent the log file only showing each lines TemplateIds. Each box in these figures represent one line in the log, the content of this line is represented by the TemplateId. The blue box seen represents the sliding window. Figures 3.5b and 3.5d are the lists of occurrences of each TemplateId that the sliding window observes in the log and these lists will be referred to as the occurrence lists.
By using a window of size three, as shown in figure 3.5a, three log messages will be 'seen' by the window for each iteration. In the case where the observed messages are 1,2, and 3, then the occurrence list will be populated with a value of one in the positions representing message 1,2, and 3, as can be seen in figure 3.5b, due to each message only occurring once in the window. The occurrence list now represents what messages and how many times they have occurred in the window.
If the messages observed by the window are 5,6, and 5, as seen in figure 3.5c, then the positions representing message 5 in the anomaly list would be populated with a two as message 5 was observed two times in the window and the position representing message 6 would be populated with a one as message 6 was only observed once in the window. This is shown in figure 3.5d.
This process is repeated for every iteration and a new occurrence list is created for each iteration. For j iteration, j anomaly lists will be created. A list of length i is filled with the j number of occurrence lists resulting in a matrix of size $i \times j$. This matrix is the basis of the LSTM approach, hence, it will be called *LSTM Matrix*.
The data in this matrix is then split up into two sets, one to be used as the input values and one to be used as the expected output values. This division of data is done

by iterating through the matrix and for every $matrix_i$: $matrix_{i,0}$ until $matrix_{i,j-1}$ are the inputs and $matrix_{i,j}$ is the expected output of the model. The model is then trained to predict $matrix_{i,j}$ based on $matrix_{i,0-(j-1)}$. In this way, the model is trained on the matrix in order to learn how many times each message type will occur based on the previous x windows.

Anomaly detection: For the anomaly detection using this model, a list with a length equivalent to the length of the log is used. This list is initialized with zeros and will be referred to as the anomaly list. The model is first used to make predictions resulting in a list of predicted windows. These windows contain the predicted occurrences of each log message per window and should correspond to what would be observed if the method for creating the dataset described above was applied to the log being analyzed.

The sliding window approach is used here again, but for a different purpose. The sliding window is applied to the dataframe representing the log, as described in 3.1, and each predicted window is compared to the actual windows observed from the log. This comparison is continued throughout the whole log/dataframe. For each predicted window containing one or more faulty predictions, the whole area that the window is looking at is incremented with a one in the anomaly list. For instance, if the window is looking at lines 3, 4, and 5, in the log and the prediction of the occurrences in this window is wrong, then the positions in the anomaly list representing lines 3, 4, and 5, are incremented by one. An example of this is shown in figure 3.6.

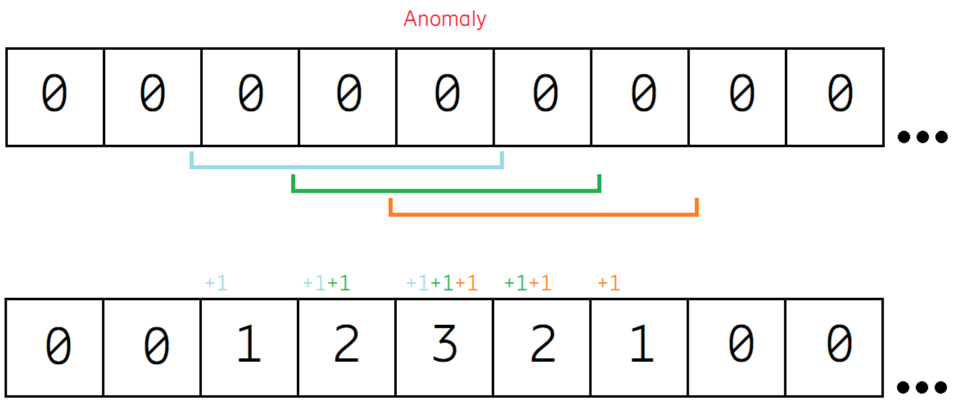


Figure 3.6: Example of how the anomaly list is incremented depending on a faulty prediction.

In the example shown in figure 3.6, the sliding window has a window size of three and a step size of one. An anomaly exists on line five in the log. In a log from a successful execution with no anomalies, the messages on the first 9 lines of the log might be 1, 2, 3, 4, 5, 6, 7, 8, 9. The anomaly could have a TemplateId of 100 meaning that the messages on the first 10 lines of the log with the anomaly would be 1, 2, 3, 4, 100, 6, 7, 8, 9. As the model would expect this sequence of logs to not contain a 100, the prediction on line 5 would not match the sequence seen in the log with the anomaly. As the sliding window is iterating over the log, three of the sliding windows will 'see' the faulty prediction and anomaly. In figure 3.6, these three windows are represented by the blue, green, and orange lines.

The positions in the anomaly list representing these lines would be incremented three times, once by each window that observed the anomaly. The result of the increments in the anomaly list is a gradually increase, peak, then gradual decrease in values. This gradual increase and decrease results in peaks where the prediction is most certain. After all the iterations of the sliding window, the anomaly list can be plotted to visualize the predictions. An example of this can be seen in figure 3.7.

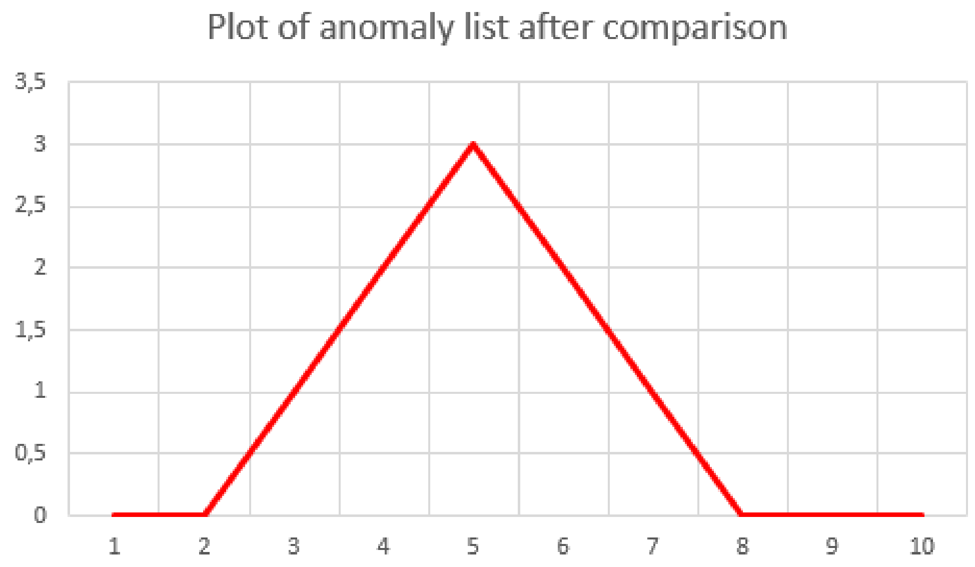


Figure 3.7: Plot of anomaly list.

The peaks in the plot are then considered anomalies and their corresponding x value, on the x-axis, indicate the line in the log where an anomaly exists.
If the window size were to be changed, then the highest possible value of the peak would also change. These two values are equal as the window size dictates the number of windows that will observe each point. The number of windows that observe each point will increase, and decrease, if the window size does. With a quick analysis of this methodology a problem becomes clear. Using this approach, anomalies at the ends of the log will not be detected as the peaks at those points will not be overfed by the same number of windows. But luckily, the solution to this problem is very simple.
For example, using a window size of 20 and a step size of 1, the expected peak value of a prediction on a anomalous log message is 20. But as the sliding window approaches the end of the list, this value would decrease as less windows would observe each log message. Given a list of length 1000, then at position 981, 19 windows have seen that point. At position 982, 18 windows have seen that point, and so on until we reach the final value in the list, in which only one window would have seen that point. Therefore, this attribute of the approach is taken into consideration when analyzing the peaks in order to avoid mistakenly interpreting a prediction of an anomaly as a prediction stating the absence of an anomaly. By making sure that the peaks at the end of the list can be of a lower value, this oversight can be avoided. The same logic applies at the start of the list. After this, the x values corresponding to the peaks are considered anomalies, and they were stored in a list to be used later.

3.2.2 Second LSTM model

The dataset for this model was created in a similar manner to the method mentioned in section 3.2.1, though differing slightly as it was done in a more traditional manner. Again, a sliding window approach used with a step size of one. The sliding window had a window size of x and the datapoints from 0 to x were considered as the input signals. The datapoint in position $x + 1$ was considered the expected output signal. The window size, or x, was obtained based on results from testing many values ranging from four to 100. Using this approach, each prediction is based on x previous observations.

Following the completion of the sliding window, each output signal and its corresponding input signals were divided in to separate lists to be used with the model during training.

Similarly to the previous method, mentioned in section 3.2.1, there is only one feature used in the dataset, the unique message identifier called 'TemplateId'. As all datapoints are of this type, the prediction will similarly be of this type, therefore the model is predicting what type of log message should occur after x observed messages. The model could now be used as a time series predictor.

In order to learn the characteristics of a log from a passing execution, only such logs were used in the training set. The logs of several successful executions were used when creating the training set and the model was then trained on this dataset. After that, the model was used as a predictor on new logs in order to find anomalous lines. Due to the training, the model will make predictions that align with the expected behavior of a log from a passing execution, therefore, if a prediction is faulty, i.e., the log being analyzed does not adhere to what the model is expecting, it can be assumed that the log contains some kind of anomaly. Each prediction made was compared to the actual log message in the log being analyzed and all faulty predictions were considered as an anomaly.

Same as the previously described model, the lines corresponding to the anomalies were stored in a list of anomalies.

3.2.3 Random forest model

Initially, the dataset used for the random forest classifier was the same as the one described in section 3.2.2, meaning that x number of TemplateId datapoints were used to predict the $x + 1$th datapoint. Similarly, the training set only consisted of logs from passing executions. The random forest was then trained on the training set and thereafter it was used for predictions on the test set. The anomaly detection was done using the same principle as the methods described in sections 3.2.1 and 3.2.2: As the model is trained on only logs from passing execution, the predictions will be biased towards those sequences and a acceptable sequence will be expected. If the prediction is wrong, it is assumed that the log does not follow a standard sequence and the faulty prediction is caused by an anomaly. These anomalies were stored in a list as well.

However, as mentioned in section 2.4.1, the random forest classifier is inherently bad for time series predictions. In an attempt to improve the performance of the classifier, the dataset was altered in a manner that would improve the accuracy of

the predictions. To do this, more content from the logs were added as features in order to enhance the classifiers understanding of the logs and therefore help it make better predictions. Some examples of the added features are the level of the log message and the component causing the printout.
As with all models, manual hyperparameter tuning was done in order to improve the models performance. In addition to this, and the increase of features added to the random forest classifier, an exhaustive grid search was carried out. The exhaustive grid search was done on a wide range of variables which required a long execution time, but helped verify and improve the results of the manual hyperparameter tuning.

3.2.4 Markov-Chain model

Other than the steps mentioned in section 3.1, no other pre-processing or steps were done. Instead, the results from the steps in section 3.1 were the basis for this method. Granted, naturally, the logs where divided into a training set consisting of only logs from passing executions and a test set. A square matrix, that would be used as the transition matrix, was created with a length equal to the number of unique log messages. Each log that was in the training set was analyzed and each transition between two log messages was logged in the matrix. After all the logs in the training set had been analyzed, the average probability of each transition was calculated. With these probabilities, the matrix had become a transition matrix. When a prediction needs to be done on a new log, every transition of said log would be compared to those of the transition matrix. As only acceptable transitions would be stored in the transition matrix, if any transition found in the log could not be found in the transition matrix, it would be considered as an anomaly. Specifically, the log message at the end of the transition would be considered the anomalous line. In the case when an anomaly was found, two transitions would be seen as new, the first one being the transition from the log message before the anomalous message, and the second being the transition from the anomalous message to the log message succeeding it. Due to this, the latter transition has to be ignored as this will cause false positive predictions since only the first transition is considered anomalous, by the definition given above, and the second one is a by-product of the anomaly. If more than one anomaly was present, one after the other, then the final transition would be the one considered faulty and therefore ignored.
The lines in the log, where these anomalies occurred were stored in a list, same as what is done for the other models.

3.2.5 Aggregation of predictions

Initially, we planned to examine the performance of a few models and continue the study using one of the models that displayed better performance in comparison to the others. Thereafter, the best performing model was to be further improved with better hyperparameter tuning. When it was observed that all models had many false positive predictions, we reassessed the plan. By considering the strengths and weaknesses of the models a new plan was devised. In an attempt to reap the benefits

of each models strengths, the predictions of all models was to be aggregated in a manner similar to how the random forest classifier operates. With this ensemble like methodology, the predictions that would be acted on were the ones which most, if not all, of the models agreed on. An example of this is if by combining three models in which one model considers lines two, three, and four, of the log to be anomalies, but the other models considers only line two and three as anomalies, then line four would not be considered as an anomaly.
Taking this into account, as each model make predictions, they will be stored in a list specific to that model. Then, these lists will be analyzed and a final list of aggregated predictions will be created in which only the predictions that all models agreed on will be stored.

3.3 Flow analysis

Before a flow analysis can be done, the flows need to first be found. Considering one log, the flows of this log should be similar to the previous execution. Therefore, the log that needs to be analyzed will be compared to the flows found in the log of the previous execution. The flows are generated from an analysis done on the TemplateIds of the log.
Using a sliding window approach, we are able to iterate over a log and analyze the contents of each window iteration. First, a window size has to be chosen and in this example the window size will be three. Considering a sequence observed by the window being "5->6->7", then the rest of the log would be searched to find this sequence. If this sequence was to appear multiple times in the log, the values after each occurrence of the sequence would be compared. If all of these values were the same, then the window would be moved one step ahead. Continuing the example, if all values after each occurrence of the sequence "5->6->7" are "8", then the next sequence to be analyzed is "6->7->8".
If this is not the case, and some of the values after one or more of the windows are different, then there is a possibility that the sequence is a shared segment from different tasks. In order for this to be confirmed, the window is extended one step back, and now of length four. The window will change from "5->6->7" to "4->5->6->7", if "4" is the values before the window. Following the increase of window size, the same forward check is done as described above, only now using a bigger window. In the case where the values in front of each occurrence of the, now extended, window are the same, then the window was reduced to its original size and moved one step forward. In this example, the value after would be "8" and the window would be changed from "4->5->6->7" to "6->7->8". In the case where the values after the sequence are not the same, then the last point of the window is a so called divergence point. In this example, "7" would be a divergence point.
This process was then repeated for the entire log.
As a divergence point can be cause by either concurrency or a new task, this has to be evaluated for each divergence point that is found. Continuing with the same example as above, if for the sequence "4->5->6->7" the next possible values are "8, 26", then sequences "4->5->6->7->8" and "4->5->6->7->26" have to be evaluated. If the value after each window was the other possible next value, i.e. "4->5->6->7->8-

>26" and "4->5->6->7->26->8", then this was most likely caused by concurrency. On the other hand, if the divergence point was caused by a new task, then the possible next values, "8" and "26", would not appear after each other.
After all of the divergence points, as well as their cause, was found, the sequences between these divergence points were considered sub-flows. The sub-flows were then converted to a list of lists for ease of use. When a prediction is done and an anomaly is found, the line of the log where this anomaly occurred could then be used to extract each sub-flow containing that line. These sub-flows are the expected behavior of the log and can then be compared to the actual behavior of the log containing the anomaly. Showing these two sub-parts of the logs results in an easier comparison.

3.4 Testing

The testing that was conduced was done in two parts where each part served to evaluate the performance of the models in different use cases. The first test was done using all the collected logs in one big dataset, while the second test was done in order to simulate a more realistic situation in regard to the amount of available data. These two tests are described in following subsections.

3.4.1 Test 1

Many ML tests are done on big datasets as it better shows the general performance of the model in question. With the aim of gathering such a result, test 1 was done in the same fashion using a larger training set consisting of most of the gathered logs. All logs from failed executions, and therefore containing anomalies, were added to the test set. For every log in the dataset containing an anomaly, a log without an anomaly, close in time-of-execution, was added to the test set as well. The remaining logs were then added to the training set. The training set was then used to train all the models and the time required to train each model was logged. After the training was concluded, the models ability to detect anomalies was then tested on the test set. The total number of anomalies, detection rate, as well as the total number of false positives were then compared.

3.4.2 Test 2

The purpose of test two was to simulate a more realistic situation in regard to the amount of available data. In addition to this, the more realistic scenario was set up in order to simulate how the models would perform in production. First, the logs were divided based on when the test that created them was executed. The division resulted in groups where each group consisted of logs from test executions done during the time span of one week. All passing logs from week x were then used to train the models and, after the training, the models would be used for anomaly detection on the logs from week $x+1$. The results from the predictions done on the logs from week $x+1$ was compared in the same manner as mentioned in section 3.4.1. Following that test, the trained models were further trained on logs from week $x+1$, meaning that the models were now trained on logs from week x and week $x + 1$. The models were then used for anomaly detection on week $x + 2$. Each time the models were used as anomaly detectors, the amount of false positives and number of anomalies that they detected were compared, i.e., the same sort of comparison done for test 1 described in section 3.4.1.

3.5 Evaluation

After each test was done on a test set, the evaluation was performed. The method of evaluating the results was identical for test 1 and test 2. After the anomaly detection was complete, the total false positives, false negatives, true positives, and true negatives were gathered for each model and used in a confusion matrix as described in section2.5.1. This was done for each model. These values were then used to calculate the precision and recall, using the formulas seen in figure 2.8. With these values calculated, the F1-score of each model was computed and all the results were plotted in bar charts to make the comparison easier.
For test 2, this procedure was repeated for each sub-test.

3.6 Re-evaluation of ’good’ logs

The testing of the models performance was done more than once. After the analysis of the the first round of tests was completed, it became clear that some of the logs though to not contain any anomalies, as the test executions that generated the logs had been marked as passed, in actuality had anomalous log messages in them. In order to expedite the detection and collection of these logs, a new plot was created. The plots were created for each log that a model had a faulty prediction in and displayed each models faulty predictions. This was done in order to differentiate the logs containing false positives, and the logs containing anomalies. An example of this can be seen in figure 3.8. The x-axis in the figure represents the lines of the log and the y-axis is omitted as it is only used to give a uniform spacing of the different models.

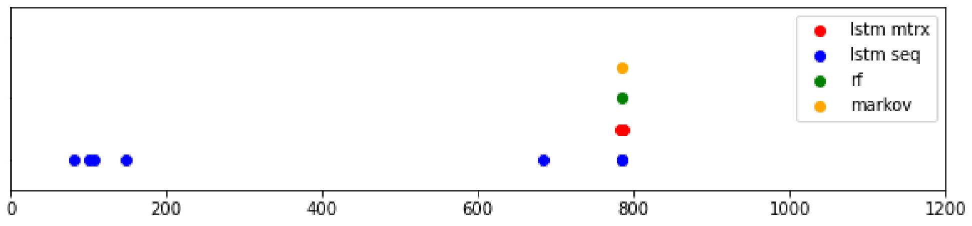


Figure 3.8: Anomaly spotted from predicting on a ’good’ log.

With a quick glance of these plots, it was more obvious for some logs than others that they contained anomalies. The most suspicious logs could be gathered and since every prediction of a possible anomaly had a corresponding line id value with it, the actual suspicious logs could be extracted. If the suspicious log was an anomalous log message, an automatic search was programmed to go through all logs and see which ones contained the anomaly, but had been marked as passing the test. These logs could then be removed from the training set and added to the test set.

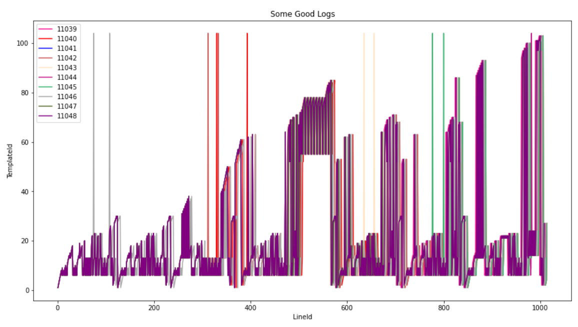
4
Results and discussion

In this section, the results from the parsing and the tests are presented with a corresponding evaluation of each individual model's performance. In addition to this, every possible permutations of the models predictions are also evaluated. The evaluation is done by examining how adequate each model is at finding anomalies in the logs. Furthermore, the time needed to train each model is compared and included in the evaluation. Finally, the results are discussed and conclusions are made based on the discussion.

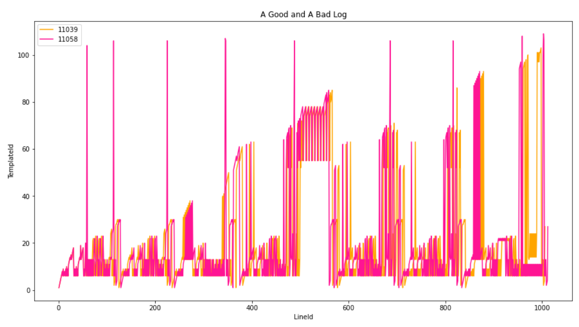
4.1 Parsing and Preparation of Dataset

The number of unique log messages would not stay consistent between test executions. With time, the logs would contain an increasing amount of new unique log messages. With the addition of each new unique log message, a similarly new and unique identifier had to be assigned. As the change of identifiers discussed in section 3.1 resulted in a mapping of each unique log message to an integer, each new unique log message could be assigned an identifier simply by following this mapping. Each new unique log message would be mapped to an integer of higher value than the most recent identifier assigned. These new log messages could occur at any point in the log. If the new log message was located anywhere other than at the very end of the log, it would result in a plot of said log to have a spike in the x-value corresponding to the line the new log message appeared in. This was due to the fact that the new identifier would have a much higher value than the identifiers of the log messages surrounding it. An example of this behavior is visualised in figure 4.1a. The same behavior was observed in logs that contained anomalies. As the logs were being parsed, the anomalous log message would be assigned an identifier with a higher value than the rest of the log as this log message was considered new and unique. The anomalous log message could also appear anywhere in the log, leading to these spikes. In figure 4.1b, this behavior is shown.

If the new non-anomalous log messages, which was causing these spikes, are rare enough, they could result in false positive predictions with certain models. The presence of a new log message that is not adhering to the expected characteristics of a log from a successful test execution, is considered an anomaly. A rare non-anomalous log message can therefore also follow this definition and cause it to be, wrongly, detected as an anomaly. This is possible because a model can be insufficiently trained to distinguish the non-anomalous log message from an anomalous log message.



(a) Plot of several good logs.



(b) Plot of bad log on top of good log

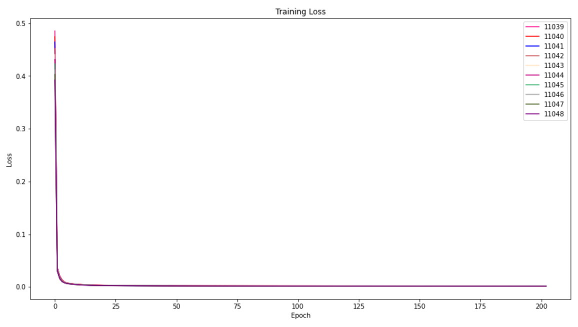
Figure 4.1: (a) Example of spikes caused by new non anomalous messages (b) Example of spikes caused by anomalies

4.2 Individual Model Performance

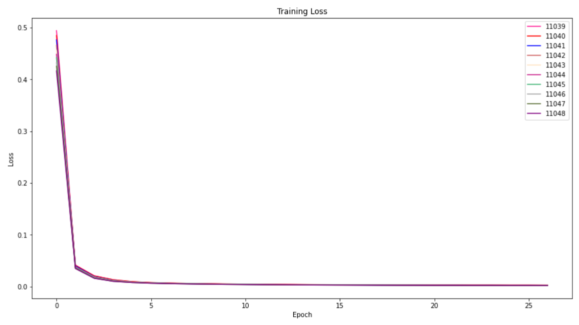
In the following subsections, the results of a test run with each model will be presented.

4.2.1 LSTM matrix

The performance of the LSTM matrix, LM, was evaluated and then reevaluated after training with fewer epochs with the purpose of lowering the time needed to train while maintaining desirable performance. When training for 200 epochs using a GTX1080, the training took around 34 minutes. The training loss of this training can be seen in figure 4.2a where the label shows the id of each log file. Coincidentally, the validation loss followed the exact same curve and resulted in the exact same plot. From the aforementioned plot, it is clear that less training can be done while still maintaining similar, if not the same, performance. Figure 4.2b shows the loss when training with 25 epochs. From this plot it is clear that further reducing the epochs to a value in the range of 5-25 epochs will have little to no negative impact on the accuracy of the model when using this dataset.



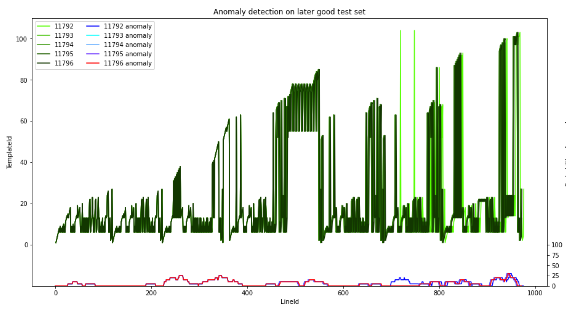
(a)



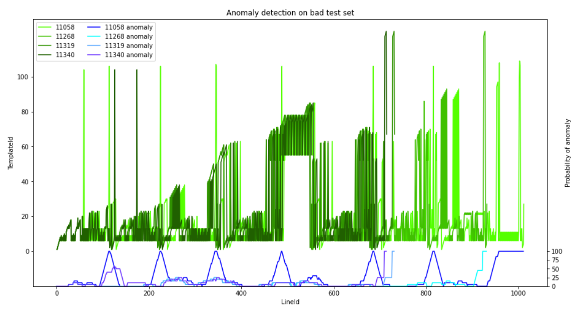
(b)

Figure 4.2: Training loss when training with (a) 200 and (b) 25 epochs.

After the training, the model's performance was evaluated using two different test sets. One test set contained logs without anomalies, while the other test set contained logs with anomalies. Figure 4.3 shows the results from these tests where figure 4.3a shows predictions done on the test set where no logs contained any anomalies and figure 4.3b shows the predictions on the test set which had logs with one or more anomalies.
The plots in figure 4.3 contain two y-axis. The first y-axis, on the left of each plot, represents the 'TemplateId' and is the similar in nature to the plots shown before in figure 4.1. The second y-axis, on the right side of each plot, represents how certain the model is that there is an anomaly on any given line (x-axis) in the log on a range from zero to 100%.
Different curves in the figures show which log is being plotted, indicated by the logs identifier number, and the resulting predictions done on that plot.



(a)



(b)

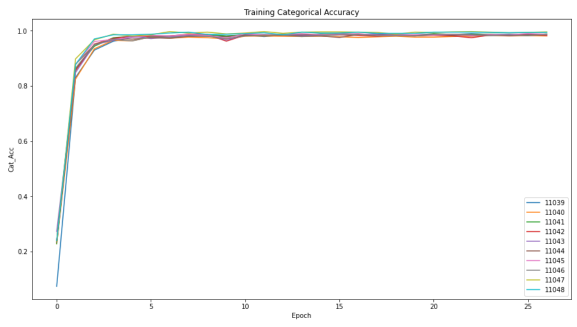
Figure 4.3: Prediction on test set (a) no anomalies, (b) with anomalies.

As can be seen in figure 4.3a, the spikes described in section 3.1 and shown in section 4.1 are present in the logs that are devoid of anomalies and still the model is able to classify those logs, causing the spikes, as non-anomalies. This means that they do not have any impact on the predictions. The logs shown in figure 4.3b have spikes that are caused by both anomalies and non-anomalous log messages. Yet the model is able to correctly classify these datapoints.

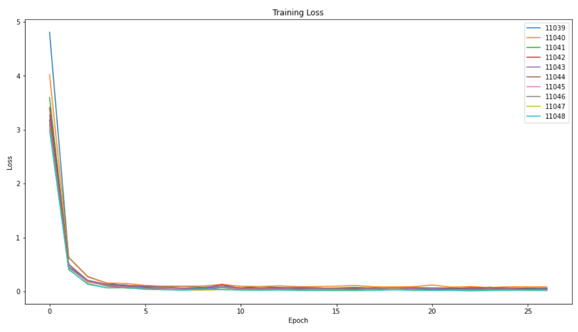
4.2.2 LSTM2

Similar to the evaluation of the LSTM matrix model's performance, the more traditional LSTM model was also evaluated, and then re-evaluated after training with fewer epochs with the same purpose of lowering the time required to train while maintaining desirable performance. When training for 100 epochs using a GTX1080, the training took 44 minutes and 40 seconds though it was immediately observed that the number of epochs could be greatly reduced, for this model as well, with no negative effect to the categorical accuracy. The number of epochs was lowered to 25 and the training was repeated.
Figure 4.4 contains two plots, one showing the categorical accuracy during the training, and the other one showing the training loss. As can be seen from the two plots,

the number of epochs could be further reduced, while still maintaining a good categorical accuracy. Reducing the number of epochs to 25 changes the time required for training the model to 11 minutes and 49 seconds.



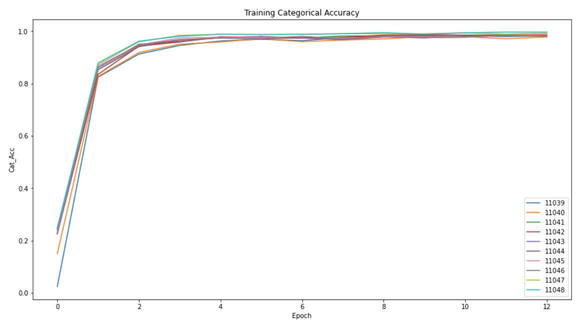
(a)



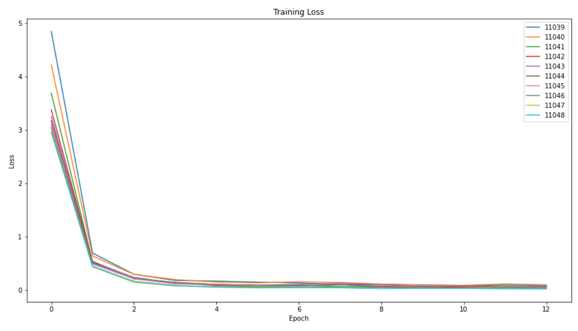
(b)

Figure 4.4: Plot of (a) categorical accuracy and (b) training loss.

Reducing the training to 10 epochs further reduced the training time to 5 minutes and 43 seconds. Figure 4.5 shows the categorical accuracy and training loss when training for 10 epochs. Here, we see that similar results can be achieved, as when training for more epochs, while saving time on the training of the model. However, further reductions in number of epochs lead to a decrease in model accuracy and training for 25 epochs proved to be a point with good trade-off between training time and accuracy.



(a) Plot of the categorical accuracy during training.

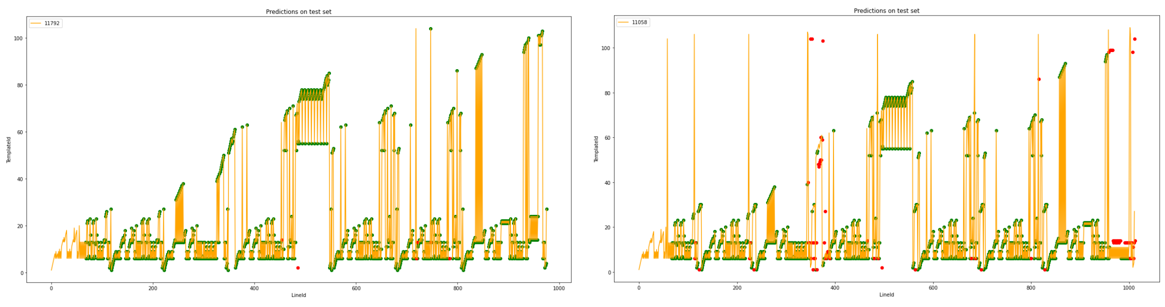


(b) Plot of training loss.

Figure 4.5: (a) Categorical accuracy. (b) Training loss.

After the training was complete, the model was tested on a test set. Figure 4.6 shows the results from using the model as an anomaly detector on the test set depicting the predictions as points on a plot of a log. The dots representing each prediction can be either green or red where green indicated that the prediction was correct and red indicated that the prediction was faulty.
Figure 4.6a shows predictions done on a test set containing no logs with anomalous log messages. Figure 4.6b, on the other hand, shows the predictions done on a test set which had logs with one or more anomalies. The number of wrong predictions

shown in figure 4.6b does not correspond to the number of anomalies in that log meaning that there are false positive predictions around each anomalous point.



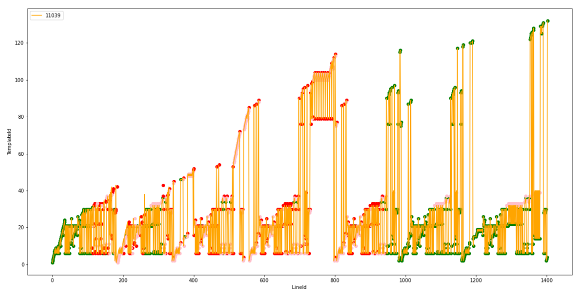
(a) **(b)**

Figure 4.6: Plots of predictions done on a test set (a) with no anomalies and (b) with anomalies.

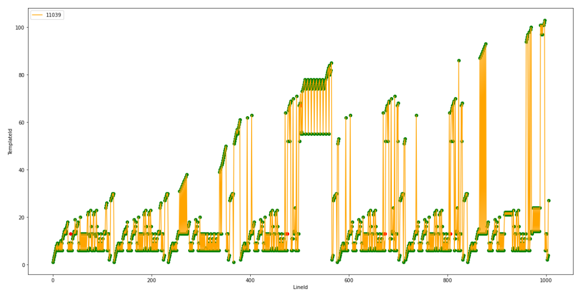
As described in section 3.2.2, the model requires x datapoints in order to make a prediction for datapoint $x + 1$. In the above figure, it is clear that no predictions are made on the first few log messages due to this. This is the caveat with this approach as no predictions can be done in the first window of x log messages. In the above figure, the size of the sliding window was 67, leading to no predictions on the first 67 log messages. The first log message that could be evaluated for anomalous content would be the log message on line 68 in the log. As anomalies can occur at the start of a log file, this will lead to any anomaly in the first 67 lines on the log to be undetected.

4.2.3 Random Forest

The random forest classifier is much quicker to train than the LSTM models. Using the same amount of logs for the training set the random forest classifier required only 5.26 second to train. In comparison to the performance of the LSTM models showed in sections 4.2.1 and 4.2.2, the random forest classifier performed poorly. It had many more false positive predictions and generally did not seem to be able to correctly predict non-anomalous lines. Due to this, the changes mentioned in section 3.2.3 had to be implemented in order to make the random forest classifier a viable option. Increasing the features to include the LineId, TemplateId, ComponentId, LevelId, and Level2Id from the log messages, the performance could be improved by a large margin.
Figure 4.7 shows the results from using the random forest classifier as an anomaly detector on the same test set before and after the improvement was implemented. Each prediction is shown as a dot on top of a plot of the log file. The dots representing each prediction can be either green or red where green indicates that the prediction was correct and red indicates that the prediction was faulty. Figure 4.7a shows the predictions before the improvement was implemented and figure 4.7b shows the predictions after the improvement was implemented and the random forest classifier was retrained.



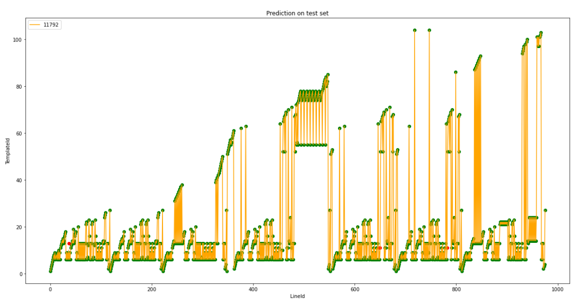
(a) Before increasing features.



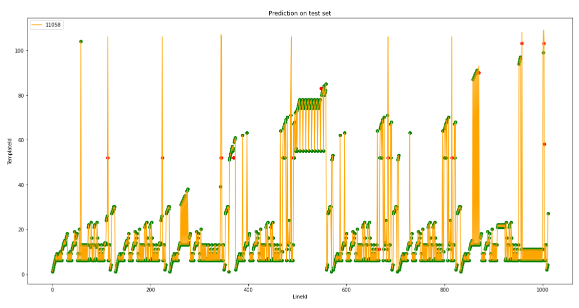
(b) After increasing features.

Figure 4.7: Change in RFC's predictions as a results of the increase in features and improved hyperparametes.

In addition to the increase in features, manual hyperparameter tuning was done. This further increased the performance of the random forest classifier. But as a final step, an exhaustive grid search was performed. The exhaustive grid search included a wide range of values and therefore required a log execution time. After completion, the hyper parameters found resulted in a slight improvement to the results gathered after the manual hyperparameter tuning.
The results shown above, in figure 4.7, were the results of training and testing on log files, whose tests that created them, were executed immediately after each other. This means that if the logs in the training set were from tests executed on during one week, then the logs in the test set were from test executions the week immediately after. This fact proved essential when determining how well the random forest classifier actually performed as an anomaly detector. From figure 4.7, it is clear that the applied changes greatly improved the ability of the random forest classifier to correctly predict non-anomalous log messages.
Following the test mentioned above, the random forest classifier was further tested on more logs where some of them were from failed test execution, meaning that they contained anomalies. The results of this test can be seen in figure 4.8 where figure 4.8a shows the results of using the random forest classifier as an anomaly detector on a log without anomalous log messages and figure 4.8b shows the results when the log contained anomalies.



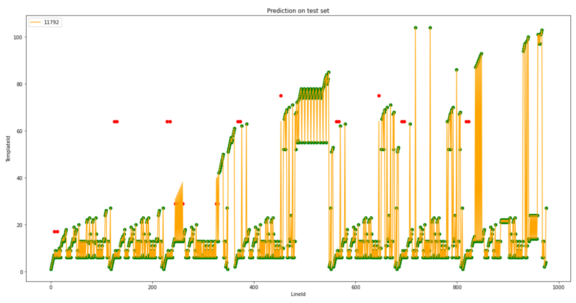
(a) Plot of predictions on logs without anomalies.



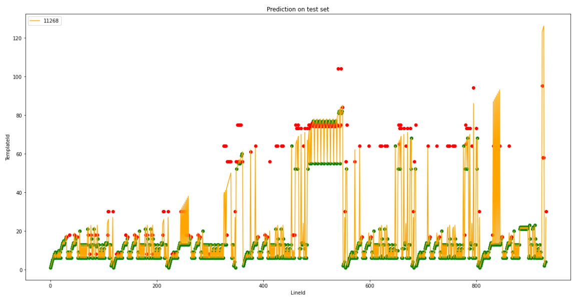
(b) Plot of predictions on logs with anomaly.

Figure 4.8: Results from using the RFC as predictor.

From the results depicted in the above figures, the random forest classifier seems to perform well as there are minimal false positives and all anomalies are detected. Although the results seem promising, they do not accurately depict the random forest classifiers performance in this use case. As mentioned earlier, if the logs in the test set and training set are from test executions close to each other, the random forest classifier will perform well, but when there is more time between the test executions generating the logs in the training set and test set, the classifier's performance will be greatly impacted. As more development is done and the tests change, so will the logs. Therefore, there will be more differences between the logs in the training set in comparison to the ones in the test set. This difference proved detrimental to the performance of the random forest classifier. Figure 4.9 shows the results of the same test as above but where the collection of training set logs and test set logs were spaced out a bit more.



(a) Plot of predictions on logs without anomalies.



(b) Plot of predictions on logs with anomaly.

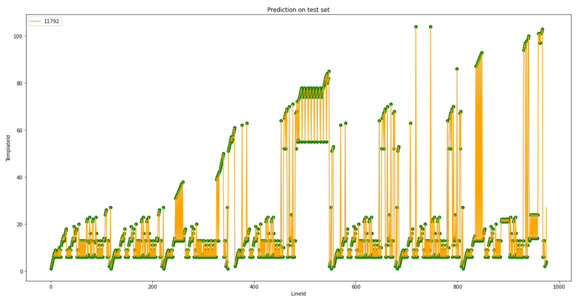
Figure 4.9: Results of RFC predictions when training set and testing set are further between in time on execution.

Comparing the results shown in figure 4.9 to the results shown in figure 4.8 the impact of the change in the logs has on the random forest classifier's performance is clear. There is a large increase of false positives in the predictions and it is especially noticeable when analyzing a log from a failed test execution. Trying to analyse the log of a failed execution where a third of the log messages are marked as anomalous will not increase the efficiency of the process. Even though the anomaly might be classified correctly, the amount this will help with making it obvious to a person analyzing the log is vastly diminished by the shear amount of false positives.

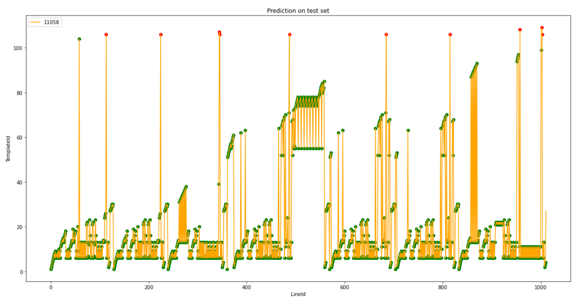
4.2.4 Markov Chain

The time required to 'train' the transition matrix classifier is by far the fastest in comparison to the time needed to train the other models tested. For this model, all that needs to be done is list iteration, addition, and division. Using 200 logs, it takes approximately 600ms to create the transition matrix needed. But if we look at the time complexity of these operations, we see that the time required to create the transition matrix will increase exponentially and there will therefore be a limit to the number of logs that can be used to create the transition matrix while the approach is still considered useful.

Using this transition matrix classifier as an anomaly detector proved to work quite well. The results from testing the classifiers performance can be seen in figure 4.10. The results of the predictions are again shown by dots where the green dots indicates that the prediction was correct and the red dots indicates that the prediction was faulty.



(a) Plot of predictions on logs without anomalies.



(b) Plot of predictions on logs with anomaly.

Figure 4.10: (a) Prediction on test set with no anomalies. (b) Prediction on test set with anomalies.

As long as the transition matrix is kept up to date, the accuracy of this approach is close to 100% with nearly no false positive predictions and all anomalies detected. The only problem that occurs is when there is a new log message introduced, then this will always be classified as an anomaly. Another problem, not shown in the figure, is when there is a change in the flow due to concurrency. Then, the new sequence of logs, while not anomalous, will be classified as an anomaly by the transition matrix classifier as it will contain transitions that are not in the transition matrix.

4.3 Aggregated predictions

The aggregation was done with the predictions of two, three, and all four models. All different permutations of two and three models predictions was also tested. This means that if two or more models agreed on the classification of a datapoint, then that classification was considered. This aggregation was only done for log messages classified as anomalous and if, for instance, the predictions of two models were aggregated, then both needed to agree on a prediction for it to be considered an anomaly. This method made it possible to reduce the total number of false positive predictions.

Figure 4.11 shows the results of aggregating the predictions of four models and the different models are color-coded for ease of comparison. The x-axis corresponds to what line in the log the prediction was made on and the y-axis corresponds to the number of models which classified that line as anomalous.

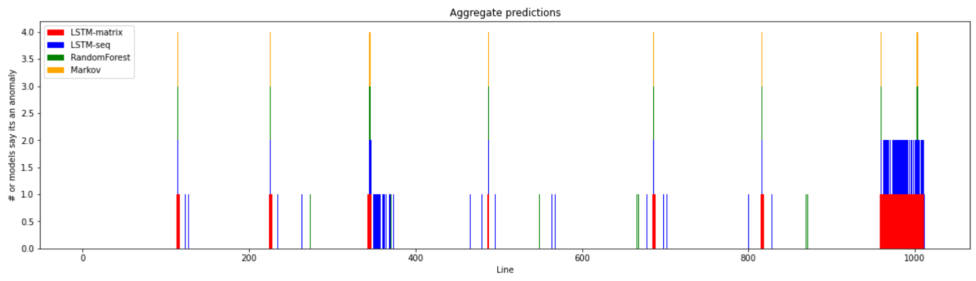


Figure 4.11: Aggregation of all predictions

As can be seen in figure 4.11, the number of points on the x-axis where 4 models agree is far less than the number of points where one, two or even three models agree.
Through testing, it was later observed that many of the false positives were caused by wrongly labeled logs. This meant that the models were trained on logs with anomalies which lowered the performance of all models and caused an increase in false positives and a reduction in true negatives.
From the figure, it becomes clear that the number of false positives will predominantly be affected by the model with the fewest false positives. The best performing model in the combination will largely dictate how many false positives the combination will have. By also considering the time and computations needed to train multiple models compared to only the best performing one, the usefulness of this aggregation diminishes, at the very least for this application and with these models. How this aggregation affects the number of true negative predictions is analyzed in the following section.

4.4 Test results

4.4.1 Test 1

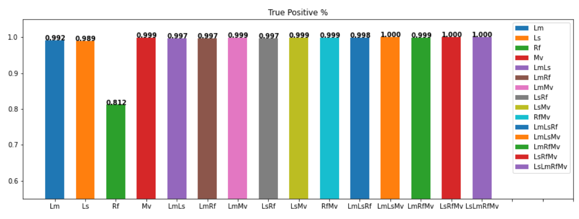
The first test aimed to compare the general performance of the models, as well as the different combinations of their predictions, by using a big dataset. For this test, the models were trained on a large amount of logs followed by testing them on a smaller test set. The test set consisted of fewer logs than the training set and some of the logs were from failed test executions which meant that they contained anomalies.
Supplementary to the performance evaluations of each model, it was also important to compare the time needed to train each model. Table 4.1 shows these times when the models where trained on the bigger dataset.

Platform	**LSTM Matrix**	**LSTM Sequential**	**Random Forest**	**Markov chain**
GTX1080	30min 17s	42min 20s	00min 38s	00min 01s

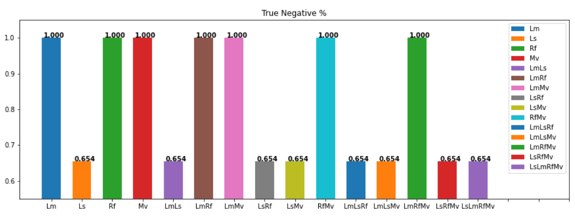
Table 4.1: Training time for test 1

As can be seen in the table, it is clear that the creation of the transition matrix is far faster than the training of any of the other tested approaches. Following the time of the transition matrix classifier is the random forest classifier, requiring less than a minute to train. Lastly are the two LSTM models, but noticeably the LSTM matrix approach is ≈ 28.5% faster to train than the 'traditional' LSTM approach. Though this time difference will matter very little if the performance of the faster models is worse.

Figure 4.12 contains two plots, figure 4.12a shows the true positive rate of each model, and all combinations of the models predictions, while figure 4.12b shows the true negative rate for the same models. The labels indicate what model, or combination of models, each bar is for. The names of the models have been shortened where Lm is for LSTM Matrix, Ls is for LSTM Sequential, Rf is for the random forest classifier, and Mv is for the Markov chain inspired transition matrix classifier. For any combination of these models the label is simply the concatenation of the shortened names.



(a)



(b)

Figure 4.12: Test 1's (a) true positive rate and (b) true negative rate

As can be observed in figure 4.12b, the true negative rate is dependant on every model in a combination to be able to detect all anomalies. If one model is unable to detect all anomalies, then any combination of predictions done with this model will likewise not be able to detect all anomalies. Any combination of predictions is only as strong as the weakest part of the combination. As the 'traditional' LSTM model was unable to detect ≈ 35% of the anomalies, any combination done with this model did not find those anomalies either. The reason for the 'traditional' LSTM model not detecting some anomalies is, as mentioned in section 3.2.2 and shown in section 4.2.2, due to there being no predictions done in the first window and this test containing logs whose anomalies where within the first 67 lines of the log. This meant that even if another model did detect those anomalies, this model would not agree on those predictions leading to the predictions being dismissed when aggregating predictions.

The actual benefit of the aggregation of predictions can be seen in figure 4.12a for any combination containing the random forest classifier. The true positive rate for the random forest classifier is much lower than any combination containing it and another model. This indicates that the false positive predictions of the random forest classifier are, mostly, not for the same log messages as the false positive predictions of the models whose results are being aggregated with the results of the random forest classifier.

Metric	**LSTM Matrix**	**LSTM Sequential**	**Random Forest**	**Markov chain**
FP %	0	100%	100%	0

Table 4.2: Number of logs without anomalies that had false positive predictions.

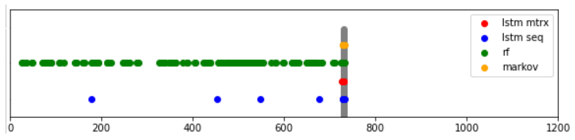
Another observation made was that the false predictions done by the models were spread out differently. The LSTM matrix model and the transition matrix classifier made no false positive predictions in logs that contained no anomalies. All of the false positive predictions made by these models, albeit few, were in logs that contained anomalous log messages. In addition to this, all of the false positive predictions where close to the line where the anomalous log message was present. To give an example, if the anomalous log message was on line 55 of the log, then the false positive prediction were usually on lines 54 and 56.

The ’traditional’ LSTM and the random forest classifier, on the other hand, had false positive predictions in all the logs in the test set. Although the amount of false positive predictions were relatively low, they occurred in every single log tested.

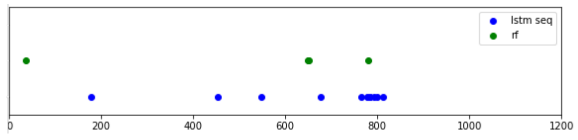
The results in figure 4.12 makes it seem as if a combination of the random forest classifier and any other model is a good idea, but the fact that the classifier is very inaccurate means that the benefit of the combination is one-directed. For example, if the random forest classifier is combined with the LSTM matrix model, then the LSTM will improve the results compared to only using the random forest classifier. If we compare the results of the LSTM with the results of combining the LSTM with the random forest classifier, then we see no improvement. Figure 4.13 shows the aforementioned inaccuracy of the random forest classifier. In this figure, each line that the models classify as an anomaly is represented by a color coded dot. A gray vertical bar is drawn where, and only if, an anomaly is present. The y-axis can be disregarded as it is only used to give a uniform spacing of the models predictions. in figure 4.13a, we see the predictions of each model on a log that contains an anomaly. It is obvious that the predictions of the random forest classifier do not help the overall accuracy of any model if their predictions were to be combined with those of the random forest classifier. From this, we can conclude that the combination of predictions can seem to help, although in reality it is just the better performing model that is dictating what is classified as an anomalies.

What has been stated above likewise holds true for predictions done on logs without anomalies. Figure 4.13b shows an example where the ’traditional’ LSTM’s predictions contained more false positives than the predictions of the random forest classifier. Here, the total number of false positive predictions can be reduced by only considering the log messages that are classified as anomalous by both the random forest classifier and the ’traditional’ LSTM.

In the case where two models perform similarly in regard to the number of false positive predictions and the predictions rarely occur on the same lines, then the aggregation is very beneficial. From these tests, any combination usually consists of two, or more, models where one model greatly outperforms the other ones. Therefore, it is more beneficial to simply use the better performing model rather than deal with the increased complexity of training more models and aggregating their results. An as seen in figure 4.12 the benefit of the combination is at most 1.1% (disregarding the random forest classifier).



(a) More random forest false positives.



(b) Less random forest false positives.

Figure 4.13: (a) Example where random forest predictions do not help reduce false positives. (b) Example where random forest predictions help reduce false positives.

From the results of test 1, some conclusions can be made, mainly that the LSTM matrix model and the transition matrix classifier have the best performance both in regard of true positives and true negatives. They are the best options when trying to detect anomalies in any new log which has not been present in previous passing tests. The transition matrix classifier gets the edge here as it is the fastest to train and therefore has a higher gain vs. cost value. If the amount of data was to be increased, then the transition matrix classifier would simply not be an option as it would require much more time to train.
Moreover, from the test we can conclude that there is no benefit to combining the models predictions and a better option is to chose either the LSTM matrix or the transition matrix classifier based on the amount and size of the logs as these models had the best precision and recall.
The caveat with this testing methodology is that the effects of concurrency that can occur in the logs cannot be properly tested. In addition to this, the effect that the continuous development of test has on the logs cannot be tested either. For this reason, test 2 is needed to show how the models perform during those circumstances.

4.4.2 Test 2

Test 2 is supposed to simulate a possible application of the anomaly detection and aid in the evaluation of the models performance in such a situation. In this test, less data will be used to train the models and with time, this amount will be increased and the models will be further trained. As the amount of data increases, so does the amount of new log files and other differences between the logs. The use case which is being simulated is that the models are being trained, and used, on the automatically run tests. Logs from these test executions will be collected during one week, and then the models will be trained and the models will be used for anomaly detection the next week. Thereafter, the models will be further trained on the second week's logs and used as an anomaly detector on the logs of test executions executed during week three.
The big dataset used in test 1 is split up into groups depending on when the logs were created in which each group represents one week of test executions. Each week of test executions consists of seven days worth of logs where each day contains 24

logs. With this, the testing can be done to analyze the performance on a week by week basis.
Due to the ratio of logs without anomalies to logs with anomalies being very high, the faulty predictions caused by the presence of an anomaly can be obscured by the amount of non-anomalous log messages. In order to not diminish the false positive predictions on this dataset, the tests were run twice. First the models were used as anomaly detectors on the full test set, then the models were tested again, but on a subset of the test set that contained equal amount of logs with and without anomalous log messages.

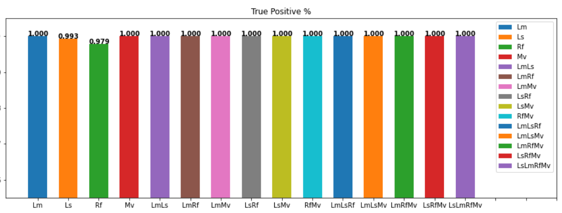
4.4.2.1 Train on week 1, test on week 2

In this test, the models were trained on logs from the tests executed during week 1. After that they were used as anomaly detectors on logs from tests executed during week 2.
The time needed to train each model is shown in table 4.3.

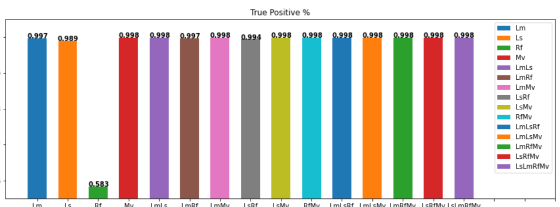
Platform	**LSTM Matrix**	**LSTM Sequential**	**Random Forest**	**Markov chain**
GTX1080	08min 02s	10min 59s	00min 09s	00min 0.5s

Table 4.3: Training time for test 2-1

Here we see a reduction of 74% for the time needed to train each model, while the dataset decreased by 73% showing a linearity between the size of the dataset and the time needed to train.
The results of the test can be seen in the two plots of figure 4.14. Figure 4.14a shows the true positive rate when testing the full test set of logs from tests executed during week 2 while figure 4.14b shows the same metric but when the subset of the test set was used.



(a) Prediction on full week 2 test set.



(b) Prediction on subset of week 2 test set.

Figure 4.14: (a) Prediction on full week 2 test set. (b) Prediction on subset of week 2 test set.

In this test, all models correctly classified all anomalies. No anomalies were present in the first 67 lines of the log, meaning that the issue with using the 'traditional' LSTM was not present. A stark difference can be seen in the true positive rate of the random forest classifier between the two test sets. This is due to the fact that the random forest classifiers predictions on any log with an anomaly follows the behavior seen in figure 4.13a. This becomes more obvious when the ratio of logs with and without anomalous log messages is closer to one.

Metric	**LSTM Matrix**	**LSTM Sequential**	**Random Forest**	**Markov chain**
subset FP %	0%	100%	100%	0%
full FP %	6%	100%	100%	3.6%

Table 4.4: Test 2-1: Number of logs that had FP and no anomalies

Similar behavior, as seen in test 1, in the spread of false positive predictions is seen here, although we see a slight increase for the LSTM matrix and the transition matrix classifier. Some of the false positive predictions made by the LSTM matrix were caused by completely new log messages that it could not predict. The remaining false positive predictions of this model were caused by faulty predictions at the end of the log. This happened because the predictions made by the model are weaker at the end of the log, as described in section 3.2.1.
The false positive predictions made by the transition matrix classifier were caused partly by completely new log messages, and partly because of new log sequences.

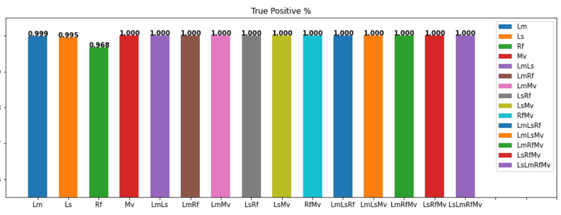
4.4.2.2 Train on week 1&2, test on week 3

In this test, the first two weeks had passed meaning that the models could now be trained on the logs from the tests executed during week two. With this two-folded increase in training data, the models were used as anomaly detectors on the logs from tests executed during week 3.
In table 4.5 we see the time needed to train the models. With the two-folded increase of logs in the training set, we see a similar increase in time required to train the models showing the same linearity between the amount of logs used to train the models and the time required to train them as seen in the reduction of logs in the dataset between test 1 and the first part of test 2. The transition matrix classifier is the one that is least adhering to this linearity.

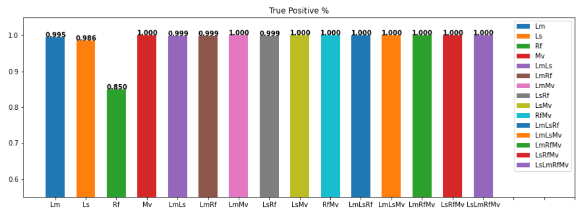
Platform	**LSTM Matrix**	**LSTM Sequential**	**Random Forest**	**Markov chain**
GTX1080	15min 52s	21min 45s	00min 18s	00min 0.8s

Table 4.5: Training time for test 2-2

After the training, the models were yet again used as anomaly predictors, but now on the logs from week 3. This test, much like the one above, was repeated using a sub section of the test set with equal amounts of logs with and without anomalies. Figure 4.15a shows the true positive rates of the models when using the full test set, and figure 4.15b shows the true positive rates when using the smaller test set.



(a) Prediction on full week 3 test set.



(b) Prediction on subset of week 3 test set.

Figure 4.15: (a) Prediction on full week 3 test set. (b) Prediction on subset of week 3 test set.

These results show a repetition of the performance difference seen in previous tests. The LSTM matrix model and the transition matrix classifier outperforms the 'traditional' LSTM and the random forest classifier when comparing the true positive rates.
Where the results from this test differs from the one above is in the true negative rate, i.e., the number of anomalies detected. In the logs from the tests executed during week 3, some anomalous log messages appeared in the first few lines of the logs. As mentioned before, these anomalies cannot be detected by the 'traditional' LSTM. Figure 4.16 shows the true negative rate of each model.

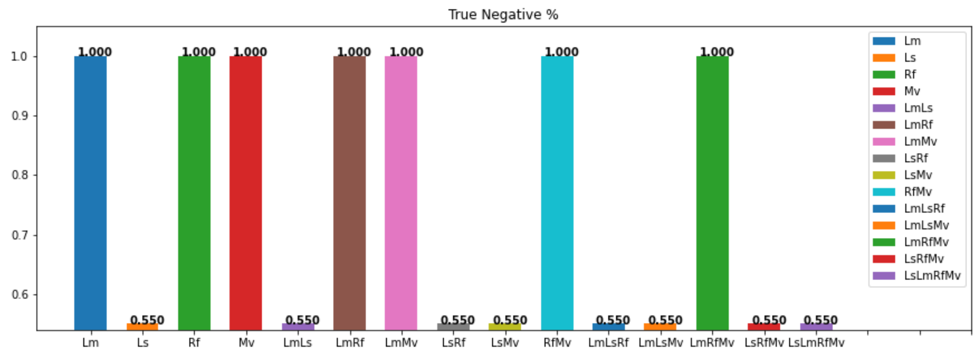


Figure 4.16: Plot of true negative rate per model and combination of predictions.

The true negative rates seen in the above figure correspond to the results seen in section 4.4.1, except that the number of anomalous log messages in the first 67 lines of the log is greater in this test set. The results of combining the logs reflect the results seen, and commented on, in section 4.4.1. Due to the large number of datapoints and slight rounding, it is evident that the transition matrix classifier had a true positive rate of 100% even though it did make false positive predictions. A point of interest in regard to those false positives is what caused them. Most of the false positives where the results of new sequences of log messages caused by concurrency. From this, we can see that the transition matrix classifier might not be an ideal option when used on logs that can change over time.
Figure 4.17 shows an example of faulty predictions done by the transition matrix classifier, random forest classifier, and the 'traditional' LSTM, but not by the LSTM matrix.

Given that the transition matrix is updated very regularly, most of the false positives can be avoided, but not in all cases. Since concurrency can cause logs without anomalies to have new transitions between non anomalous log massages, more false positive predictions will occur. Given the two non anomalous sequences "1-2-3-4-5-6" and "1-2-5-6-3-4", if every transition in the first sequence is known by the transition matrix, but not the other sequence, then the transition matrix classifier will classify the second sequence as anomalous. Furthermore, for the transition matrix to be updated with a new transition, the new transition has to occur, meaning that a false positive prediction will be made before it can be avoided.
This is where the LSTM matrix performs well. Given a big enough window, in this example 6 is fine, then the window will see that all the same messages are present in the window, just in a different order. As the predictions are made on the occurrences of each message in a window, they will be exactly the same for the two sequences as both sequences contain the same number of each log message. If a window only sees part of the sequence, then that window will assume the presence of an anomaly, but due to other windows not agreeing, this will not be wrongly classified as an anomalous sequence.

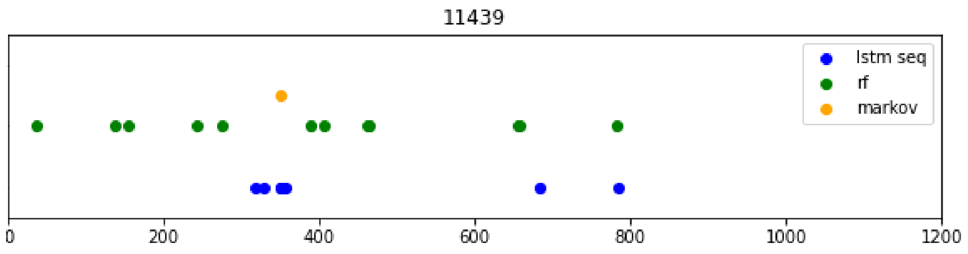


Figure 4.17: Example of markov FP due to concurrency

The few false positive predictions caused by the LSTM matrix and transition matrix classifier where spread out a bit more across the logs without anomalies compared to previous tests while the random forest classifier and the 'traditional' LSTM still had false positive predictions in all the logs of the test set. These numbers are shown in table 4.6

Metric	LSTM Matrix	LSTM Sequential	Random Forest	Markov chain
subset FP %	0%	100%	100%	0%
full FP %	11.8%	100%	100%	10.5%

Table 4.6: Test 2-2: Number of logs that had FP and no anomalies

4.5 Flow analysis

When an anomalous log message was detected, the first step of the flow analysis was to find the sub-flows. This was done as described in section 3.3. Thereafter, the list of sub-flows could be used to display what flow was expected compared to the flow gotten in the log being analyzed. The manner of how the differences in the flows was displayed can be seen in table 4.7.

Expected	**Got**
1-Test starting	1-Test starting
2-step 1	2-step 1
3-step 2	3-step 2
4-step 3	4-step 3
5-step 4	**107-Total traffic loss.**
6-step 5	6-step 5
7-step 6	7-step 6
8-step 7	8-step 7

Table 4.7: Results of flow analysis.

In the example given above, only one sub-flow is relevant to the anomaly found. The sub-flow is shown in comparison to the log messages, of the log with the anomaly, that appeared on the same lines. This means that if the relevant sub-flow appears on lines 1-8, then lines 1-8 of the log with the anomaly are shown in comparison, even though they might contain log messages from different sub flows.
If there would be more relevant sub-flows, then these would be displayed on the left side in a 'or chain'. For example, if there were three possible sub-flows then these would be displayed as "sub-flow1 or sub-flow2 or sub-flow3".

4.6 Discussion

As discussed earlier in this chapter, combining the prediction of different models has little to no observable positive effect on the overall accuracy, performance, or true positive rate of the models. Whether two, three or even four models are combined, the same result was always observed. The reason of this is that the combination of the models can only perform as well as the worst performing model when considering the number of detected anomalies. If two models are combined and only one of them cannot detect all anomalies, then, since the models need to agree on the predictions, the anomalies that one of the models was unable to detect will also be overlooked by the combination. Although, it was observed that in some instances, the combination led to a reduction in total number of false positives. Still, this method was further discredited as a viable option due to the increase in time required to train several models, for the purpose of aggregating the results over just training one. In some applications, reducing the number of false positives has great value, but in the application described in this thesis, the value of a small decrease in number of false positives diminishes as the time required to train increases.
This is further proved by the fact that the largest decrease in the number of false positives happened when a good model is combined with a bad one. As this can cause anomalies to be overlooked, no real value is seen in reducing false positives at the cost of reducing true negatives. It is much simpler, and of less cost, to train the

better model and use only that one without the need of adding the extra steps in order to aggregate predictions of several models.
The results showed that the LSTM Matrix implementation was not affected by the shifting order of logs caused by concurrency. In addition, the implementation showed a reduction in total false positives in comparison to the 'traditional' LSTM approach. Another aspect where the LSTM Matrix implementation performed better than its 'traditional' counterpart was when an anomaly was present in the beginning of a log. Due to the window size of the 'traditional' LSTM, any anomaly appearing within the first window will be overlooked by the model as these data points are needed to make a prediction. Reducing the window size was shown to worsen its performance, and therefore, it is not a viable solution to avoid this issue. Not only did the 'traditional' LSTM approach perform worse, but also, it required more time to train compared to the LSTM Matrix.
The random forest classifier, RFC, was very quick to train, but showed to be useful only when the logs that were analyzed were similar to the training data, i.e., the test set was similar to the training set. This means that the RFC can be used if it is retrained often, but the logs would still have to remain similar in nature. As development is constantly done and the logs change with time, every time such a change occurs, the RFC would not perform well and would have to wait for enough executions to be done in order to create a new dataset.
The use of a transition matrix classifier, TMC, outperformed the RFC and it even needed less time to train. Therefore, it made the RFC obsolete when comparing these two approaches. Compared to the RFC, the TMC still performed well when it was used as an anomaly detector on logs from executions which were not that close to the ones in its training set.

5 Conclusion

In this section, we start by focusing on the context and importance of the topic covered in this thesis followed by addressing the main parts of the thesis in a summarized way. This is then followed by a review of the key points in this thesis and motivation and the methods that are used to tackle the challenges. After that, we revisit the main results that were achieved in this work and provide a final take-home message. Finally, we comment on some possible future works that can be done.

5.1 Motivation and Importance of the Work

There are many applications of machine learning models. As their usage is increased, we allow more humans to move away from tedious work in areas where these models thrive. Additionally, as more of the work can be done by machines in a more efficient manner, we increase the value we gain from this work. In a world swimming in big data, it is unfeasible both from an economical and a social aspect to have humans manually go through heaps of information, but we should instead focus on the development of machine learning tools for this purpose.
Complex systems in the telecommunication world that generate vast amounts of data are continuously being updated and further developed. This means that with time, not only will the amount of data they generate increase, but the complexity of this data will increase as well. This is therefore a key area where we can apply machine learning models to increase the business value we can extract from this data. Humans would need constant training and a lot of experience to be efficient in analyzing the logs that these systems would generate which will burden companies with large cost and require many human resources. This is therefore a perfect application for machine learning models whose cost of training is the hardware, electricity to run the hardware, and the comparatively few people to create and maintain the models.
Many have seen the potential and spent time and resources in researching this topic. The work done in this thesis has specifically researched a method to tackle the negative impact that concurrency in logs have on the performance of the LSTM. Key aspects to design this method include maintaining a high detection rate and keeping a low false positive rate, while limiting the amount of logs (which are used for training) in order to reduce the time needed to train and make it possible to use the model in a quickly changing production environment.

The main questions answered by this work have been:

- Is it possible to use ML to learn what is expected from a log of a successful test and highlight the differences in a failed test log?
- Can ML be used to detect abnormalities in logs?
- Will the usage of ML reduce the complexity or required time to analyse the logs?
- How can this be used in other contexts? (e.g. customer logs, built-in software logs)
- How does concurrency in logs affect the ML models and how can we avoid it?

These questions have been answered while focusing on the logs from regularly run test cases which change over time as development is done on the tests producing the logs. As there are different types of anomalies, this work has been done on sequential anomalies, meaning anomalies that are caused by either missing lines (as a result of an incomplete message sequence in the logs) or new lines (caused by errors) that differ from what is expected based on logs from a successful execution of a test case.
There are many systems that generate logs of which we would like to analyze. As many of these logs are written in real-time, there is a possibility that they will be affected by concurrency. If we wish to use machine learning to analyze the logs, this concurrency will greatly reduce the usefulness of the machine learning analysis. By using the method proposed in this thesis we can avoid the issues caused by concurrency in the logs and with that gain more value from the logs. Additionally, we can relieve the tedious work from humans and let machines do this work. Many man hours can be saved while also increasing the efficiency of which the logs are analyzed.
With all of this said, it is clear that concurrency in logs when using machine learning to learn expected 'good' sequences has a negative impact on the predictive ability of a LSTM. But as we can work around this negative impact with the method proposed by this work, this obstacle is easier to overcome and both will and should not prevent us from increasing the efficiency of which we analyze logs.

5.2 Key Steps and Intuition

One of the first steps in this thesis was to research how to parse and handle the data in order to properly prepare it for the models that we tested. The log messages that we were interested in, first had to be extracted and then parsed into a structured format. The log parser Drain from LogPai's Logparser toolkit was used as it is one of the most accurate and efficient open-source online log parsers. Secondly, it was important to compare some models to see which one would fit our application the best. The LSTM was one of the main models used as it has very good performance when dealing with sequential data. The performance of the LSTM was compared to the random forest classifier and transition matrix classifier as they are lightweight in comparison and therefore have a shorter training time. The new method of parsing the data and use a LSTM was researched as a solution to the problem perceived when dealing with concurrency in the logs. Finally, the flow analysis was done to research a method of visualizing the discrepancies found in logs to a user in order to ease the log analysis process.

5.3 Main Results

One key takeaway message from the results is the difference in how many false positive predictions each model made in the logs that contained no anomalies. Both of the traditional LSTM and random forest implementations had false positive predictions in 100% of the logs with no anomalies while the LSTM matrix and transition matrix classifier had false positive predictions in a range from 0-12%, depending on which circumstance was tested, if we round up. Additionally, the traditional LSTM failed to find anomalies in the first 67 lines of the logs as these datapoints are required for the model's first prediction. The LSTM matrix, mostly, circumvents this problem as well as only the first three datapoints would have no predictions.

From the results it is clear that it is possible to use ML to learn what is expected from a log of a successful test execution by only using the logs of successful executions as the training data. Then, the predictions on other logs could be used to highlight how a log from a failed execution differs from the logs of successful executions. The ML algorithms showed good performance in detecting abnormalities in logs, and even in logs that were thought to not have anomalies. This means that ML can be used to reduce the complexity or time required to analyse the logs since the predictions can be used to highlight the anomalies in a log. With the addition of showing what sub-flow is expected, it will be even clearer what behavior the test was supposed to have, and this can be compared to the actual behavior it shows. This would reduce the total number of log messages needed to be read by a person to analyze it, and therefore, decrease the total amount of time needed for the analysis of the log.

In order to apply this process of anomaly detection to other logs, only some understanding of the logs is required, so that the pre-processing can be done properly. Then, the same methodology, and most of the same code, can be used to find anomalies in the new logs.

This thesis has shown that it is of no use to try and use multiple models and that effort should be put to finding one model that works best for the given situation and then work with improving that one. In addition to this, this thesis' contribution is a new method of applying a LSTM to avoid the negative effect of the different logs sequences that can appear. By using this method, we are able to completely ignore the issue of different sequences caused by both concurrency and a change in the test case. In addition, it allows us to improve the predictive ability of the LSTM in regard to anomaly detection on logs whose log messages can appear with different order over time. Additionally, the proposed method proved to work better than the other models that were compared with it, and required less time to train than the 'traditional' LSTM. In instances where new log sequences were present, the proposed method's predictive ability stayed unaffected while the other models had several false positive predictions.

5.4 Future work

The collected logs, that were used for this project, were the results of automated test runs that were executed hourly. These logs were only stored in the system for a limited time before they were automatically deleted. Due to the automatic deletion, it was not possible to collect logs from a long period of time. In addition to this, these tests have a very high success rate, close to 100%, meaning that it was also difficult to collect logs that contained anomalies. Therefore, a point of interest is to collect logs for a longer period of time and use them to redo the testing done in this project. By doing this, a larger dataset can be gathered with the additional benefit of having a larger variety of anomalies. From this larger dataset, more trends in how the development affects the accuracy of the models can be observed since there will be bigger changes from the first to last log. The results gathered from the tests would also be more accurate due to the larger collection of data. As more anomalies would also be present, it would be possible to see if all the types of anomalies can be detected, and if the results gathered here were due to a small sample size.

Due to time limitation, a part of the development of the flow analysis was left out. The current implementation only uses one log file to create flows which limits the usage of the sub-flow comparison as many more sub-flows can be found when analyzing several logs. By increasing the amount of log files used in the creation of sub-flows, more general and accurate sub-flows can be created which would greatly increase the usefulness of the flow analysis.

Another point of interest is to further develop the pre-processing in order for the LSTM implementation to not only be used for anomaly detection in the sequence of the logs, but also for the content in the log messages that they analyze. With this addition, one more type of anomaly can be found which is parameter anomalies. While on the topic of anomalies, it would be interesting to study the possibility of generating anomalies and how this would affect the test results. If the generation of anomalies shows similar results to tests run with real anomalies, then, this would offset the need to wait for real world anomalies, and real test executions, which would make it easier to evaluate the performance of each model.

The random forest performance could be improved more with a deeper analysis of the methods recommended when using random forest for time series data. It would be interesting to see how well random forest could perform with a dataset which is more pre-processed in a manner which would make it more suitable to use with a random forest classifier.

References

[1] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. *Mobile networks and applications*, 19(2):171–209, 2014.

[2] Ericsson. Microwave. `https://www.ericsson.com/en/portfolio/networks/ericsson-radio-system/mobile-transport/microwave`. Accessed: 2020-02-03.

[3] Anukool Lakhina, Mark Crovella, and Christophe Diot. Diagnosing network-wide traffic anomalies. *ACM SIGCOMM computer communication review*, 34(4):219–230, 2004.

[4] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Online system problem detection by mining patterns of console logs. In *2009 Ninth IEEE International Conference on Data Mining*, pages 588–597. IEEE, 2009.

[5] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining invariants from console logs for system problem detection. In *USENIX Annual Technical Conference*, pages 1–14, 2010.

[6] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 207–218. IEEE, 2016.

[7] Jie Mei, Dawei He, Ronald Harley, Thomas Habetler, and Guannan Qu. A random forest method for real-time price forecasting in new york electricity market. In *2014 IEEE PES General Meeting| Conference & Exposition*, pages 1–5. IEEE, 2014.

[8] Michael J Kane, Natalie Price, Matthew Scotch, and Peter Rabinowitz. Comparison of arima and random forest time series models for prediction of avian influenza h5n1 outbreaks. *BMC bioinformatics*, 15(1):276, 2014.

[9] Kenichi Tatsumi, Yosuke Yamashiki, Miguel Angel Canales Torres, and Cayo Leonidas Ramos Taipe. Crop classification of upland fields using random forest of time-series landsat 7 etm+ data. *Computers and Electronics in Agriculture*, 115:171–179, 2015.

[10] Ke Zhang, Jianwu Xu, Martin Renqiang Min, Guofei Jiang, Konstantinos Pelechrinis, and Hui Zhang. Automated it system failure prediction: A deep learning approach. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 1291–1300. IEEE, 2016.

[11] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceed-*

ings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 1285–1298, 2017.

[12] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Largescale system problem detection by mining console logs. *Proceedings of SOSP'09*, 2009.

[13] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132, 2009.

[14] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[15] Marc Claesen and Bart De Moor. Hyperparameter search in machine learning. *arXiv preprint arXiv:1502.02127*, 2015.

[16] Carl Benedikt Frey and Michael A Osborne. The future of employment: How susceptible are jobs to computerisation? *Technological forecasting and social change*, 114:254–280, 2017.

[17] James Manyika, Susan Lund, Michael Chui, Jacques Bughin, Jonathan Woetzel, Parul Batra, Ryan Ko, and Saurabh Sanghvi. Jobs lost, jobs gained: Workforce transitions in a time of automation. *McKinsey Global Institute*, 150, 2017.

[18] Michael Koch, Ilya Manuylov, and Marcel Smolka. Robots and firms. 2019.

[19] James E Bessen, Maarten Goos, Anna Salomons, and Wiljan Van den Berge. Automatic reaction-what happens to workers at firms that automate? *Boston Univ. School of Law, Law and Economics Research Paper*, 2019.

[20] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 121–130. IEEE, 2019.

[21] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40. IEEE, 2017.

[22] Dansbecker. Using categorical data with one hot encoding, Jan 2018.

[23] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[24] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.

[25] Michael I Jordan. Serial order: A parallel distributed processing approach. In *Advances in psychology*, volume 121, pages 471–495. Elsevier, 1997.

[26] Weijiang Feng, Naiyang Guan, Yuan Li, Xiang Zhang, and Zhigang Luo. Audio visual speech recognition with multimodal recurrent neural networks. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 681–688. IEEE, 2017.

[27] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 6.2. 2.3 softmax units for multinoulli output distributions. In *Deep Learning.*, pages 180–184. MIT Press, 2016.

[29] Paul J Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural networks*, 1(4):339–356, 1988.

[30] Michael C Mozer. A focused back-propagation algorithm for temporal pattern recognition. *Complex systems*, 3(4):349–381, 1989.

[31] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[32] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[33] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[34] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[35] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[36] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[37] Xuan-Hien Le, Hung Viet Ho, Giha Lee, and Sungho Jung. Application of long short-term memory (lstm) neural network for flood forecasting. *Water*, 11(7):1387, 2019.

[38] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2016.

[39] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.

[40] Robi Polikar. Ensemble based systems in decision making. *IEEE Circuits and systems magazine*, 6(3):21–45, 2006.

[41] David Opitz and Richard Maclin. Popular ensemble methods: An empirical study. *Journal of artificial intelligence research*, 11:169–198, 1999.

[42] Random forest template for tibco spotfire®.

[43] Tony Yiu. Understanding random forest, Aug 2019.

[44] PJ Moore, TJ Lyons, John Gallacher, and Alzheimer's Disease Neuroimaging Initiative. Random forest prediction of alzheimer's disease using pairwise selection from time series data. *PloS one*, 14(2):e0211558, 2019.

[45] Grzegorz Dudek. Short-term load forecasting using random forests. In *Intelligent Systems' 2014*, pages 821–828. Springer, 2015.

[46] Kevin P Murphy. *Machine learning: a probabilistic perspective.* MIT press, 2012.

[47] Paul A Gagniuc. *Markov chains: from theory to implementation and experimentation.* John Wiley & Sons, 2017.

[48] Peter H Peskun. Optimum monte-carlo sampling using markov chains. *Biometrika*, 60(3):607–612, 1973.

[49] Mr Matthew T Jones. *Estimating Markov transition matrices using proportions data: an application to credit risk*. Number 5-219. International Monetary Fund, 2005.

[50] Herman Scheepers. Markov chain analysis and simulation using python, Nov 2019.

[51] Stephen V Stehman. Selecting and interpreting measures of thematic classification accuracy. *Remote sensing of Environment*, 62(1):77–89, 1997.

[52] Yutaka Sasaki et al. The truth of the f-measure. *Teach Tutor mater*, 1(5):1–5, 2007.

[53] Abhigyan. Calculating accuracy of an ml model., Jul 2020.

[54] C.J. van Rijsbergen. Information retrieval, 1979.

[55] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information processing & management*, 45(4):427–437, 2009.