DEGREE PROJECT IN ELECTRICAL ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2020

Using NLP Techniques for Log Analysis to Recommend Activities For Troubleshooting Processes

MARTIN SKÖLD

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Using NLP Techniques for Log Analysis to Recommend Activities For Troubleshooting Processes

MARTIN SKÖLD

Master's Programme, Machine Learning, 120 credits
Date: December 15, 2020
Supervisor: Sahar Tahvili (Ericsson), César Soto Valero (KTH)
Examiner: Magnus Boman (KTH)
School of Electrical Engineering and Computer Science
Host company: Ericsson AB
Swedish title: Använda NLP tekniker för logganalys för att rekommendera aktiviteter för felsökningsprocesser

Abstract

Continuous Integration is the practice of building and testing software every time a code change is merged into its entire codebase. At the merge, the source code is compiled, dependencies are resolved, and test cases are executed. Detecting a fault at an early stage implies that fewer resources need to be spent to find the fault since fewer merges need to be checked for errors. In this work, we analyze a dataset that comes from a Ericsson Continuous Integration flow that executes test cases daily. We create models to efficiently classify log events of interest in logs from failing test cases. For all models, each word in the log events is exchanged with the corresponding word embedding. The embeddings come from the FastText Continuous Bag of Words and Skip-gram models that use character n-grams for each word. For Linear Regression, Random Forest, XGBoost model, Support Vector Machine, and Multi-layer Perceptron, the word embeddings of the words of the log event is merged by weighting the words with the corresponding frequency-inverse document frequency from the dataset. The best performance was achieved with XGBoost, with a mean F1-score of 0.932 and a standard deviation of 0.034 when evaluating 100 3-fold cross-validations with different seeds. The LSTM model, which takes sequential input, got a mean F1-score of 0.896 and a standard deviation of 0.061. These results demonstrate the suitability of our approach to facilitating log analysis and defects detection tasks, reducing time and effort from developers.

Sammanfattning

Kontinuerlig integration är när man bygger och testar mjukvara varje gång en kodändring är sammanslagen med kodbasen. När sammanslagningen utförs så är källkoden kompilerad, beroenden är lösa, och testfall är exekverade. Upptäckten av en fel tidigt betyder att mindre resurser behöver läggas på att hitta felet, eftersom färre kodsammanslagningar behöver analyseras. I denna studie analyserar vi ett dataset som kommer från ett kontinuerligt integrations flöde hos Ericsson som utför testfall dagligen. Vi skapar en model som effektivt klassificerar loghändelser av intresse i loggar i loggar från fallerande test fall. Gemensamt för alla modeller är att varje ord är utbytt mot motsvarande ordinbäddningar som kommer från FastTexts Continuous Bag of Words och Skip-Gram modeller som använder n-grams av tecken för varje ord. För linjär regression, Random Forest, XGBoost, Support Vector Machine och Multi-Layer Perceptron modellerna så är ordinbäddningarna för orden i varje log meddelande sammanslagna genom att vikta dem med motsvarande frequency-inverse document frequency värde. Det bästa resultatet uppnåddes av XGBoost, med ett medelvärde på F1-score på 0.932 och en standardavvikelse på 0.034 när vi evaluerar 100 st 3-fold korsvalideringar med olika frön. LSTM modellen, som tar ordinbinbäddningarna i en sekventiell ordning, fick ett medelvärde på F1-score på 0.896 och en standardavvikelse på 0.061. Dessa resultat visar lämpligheten i vårt tillvägagångssätt för att underlätta logganalys, vilket reducerar den tid och fokus som utvecklare och utövare behöver lägga på att på logganalys.

Acknowledgements

The work presented in this thesis was conducted at Ericsson's department Global Artificial Intelligence Accelerator (GAIA), Stockholm. The study was performed between March and September in the year 2020.

I would like to thank all the people I been working with during this project. I want to thank my supervisor Sahar Tahvili for her engagement in the project, and for always quickly responding to all my requests for advice and the need for resources. Secondly, I would like to thank my KTH supervisor César Soto Valero for giving guidance in the world of log analysis and being such good support. I also want to thank Raghotham Sripadraj for allocating a lot of time for discussing the problem at hand to find the solutions. Hamidreza Moradmand I want to thank for being the bridge between us and the department we been collaborating with. And last, but not least, I want to thank Pankaj Khapake and Bhagyashree Jain for all the insights in the data and for taking the time to label the data and help to create troubleshooting activities. Without all these people this thesis would not have been possible. Since we all have been working from home this all time, I hope to meet you all in person someday in the future.

I also would like to thank my fiancée Caroline Larsson for her constant support, all the way from my first day on KTH, when we left our hometown to go to Stockholm, to the day I will graduate. You've been supporting me throughout all late nights with assignments, projects, work and you keep giving me love. I also want to thank all my friends I've met at the university, whom I've shared great moments with laughs, setbacks, and opportunities. I hope that our friendships will be life long.

Stockholm, December 15, 2020

Martin Sköld

Contents

Acronyms

ANN	Artificial Neural Network
SME	Subject Matter Expert
BOW	Bag of Words
CBOW	Continuous Bag of Words
SVM	Support Vector Machine
CI	Continuous Integration
UMAP	Uniform Manifold Approximation and Projection
TF-IDF	Term Frequency–Inverse Document Frequency
CNN	Convolutional neural network
NLP	Natural Language Processing
MLP	Multi-Layer Perceptron
AI	Artificial Intelligence
WE	Word Embedding/s
UMAP	Uniform Manifold Approximation and Projection
RNN	Recurrent Neural Network
NN	Neural Network

Chapter 1

Introduction

Software and electronic devices become more and more an integral part of our lives. They seem to infiltrate every area that could potentially be simplified and improved by technology. The success of these depends greatly on how stable and robust the products are. In this regard, testing has become a crucial part of software development [1]. Stable software leads to more satisfied customers, and in many areas devices or software products are not allowed to be used without proper testing, such as in healthcare services [2].

Continuous Integration (CI) is an industrial standard practice to simplify the testing process. It consists of merging all developers working copies to a shared mainline several times a day, and each integration is verified by a pipeline that is built automatically and tested [3, 4]. This is done to get rapid feedback and catch errors as early as possible [5]. In this scenario, logs are generated by the test case executions and are created to give feedback so that anomalies are detectable [6–8]. The logs are mostly created to check the state of the system during operation. Logs are usually continuously appended to a file, which means that the file grows and become very large. The size of the logs is directly related to the test cases size, complexity, testing level (e.g., unit testing, integration testing). Going through large logs is time-consuming. It is hard since the logs also contain several entries from the system that are not related directly to the behavior of the software itself.

At Ericsson, many departments use Jenkins[1], an open-source automation server that builds, deploys, and automates tests execution. Mostly the builds and tests pass without any problems, but investigations need to be done when errors

[1]`https://www.jenkins.io`

occur. Currently, the faults that have been encountered before are found by searching with regular expressions, but it also gives false positives.

The developer must search millions of log lines as soon as a new error or a similar error with a different output occur, as these will not be identified with the regular expressions properly. The generated log events often have free text messages, accompanied by information about the time of execution, the log level, and what part of the software that generated the log event. One needs prior experience and need to know the details of the product being tested to be able to troubleshoot the logs. The department we collaborate with in this study needs to train employees at least 6 to 12 months before they can troubleshoot the logs independently and finding a fault may take hours to days. This means that the activity of troubleshooting is very costly and require multiple people to investigate the issue. Reading and analyzing a log manually for a failed test case requires solid domain knowledge. It might suffer from human judgment, ambiguity, and uncertainty.

Related to log analysis there exist a plethora of previous work. Some examples are, but not limited to, log template extraction, grouping log events based on time and order, clustering, test coverage, etc. [6]. A more in-depth discussion about previous work is presented in Chapter 2 of this thesis.

1.1 Problem Statement

The problem consists in simplifying the analysis of logs by classifying and grouping the log events generated after each test execution. Moreover, the goal is to suggest troubleshooting activities for each fault found. The troubleshooting time spent on log analysis can be significantly reduced by employing, for example, Artificial Intelligence (AI) techniques related to Natural Language Processing (NLP). The human work and mental load might be lowered by utilizing classification or clustering algorithms for those test cases that failed due to the same reason. The troubleshooting activities can be assigned when these groups have been formed.

By employing ML techniques, the number of different types of errors that developers need to look for can be narrowed down. This leads to less time finding the fault and making it is easier for a new developer to solve the issues. The importance of having large exposure to different errors could be lowered, saving both time and frustration.

The main goal of this thesis is to implement an automated approach for parsing and analyzing logs written as text. Moreover, proper troubleshooting activities need to be mapped to each log corresponding to the failed test case. That way software developers are given hints on how to solve the issue. Examples of errors one might find during test case executions are [9, 10]:

(i) the testing environment is not ready for test execution

(ii) there is a mismatch between test cases and the requirements

(iii) there are some errors in the code

(iv) there is a bug in the system under test

(v) any combination of the previous options.

1.2 Research Goals

This study investigates the possibility of creating a decision support system for mapping proper troubleshooting activities to failed test cases. We analyze different types of feature engineering on the logs. We then evaluate the performance of different classifiers on the features extracted from the dataset we collected. More specifically, the goal is:

To provide solutions for a more efficient log analysis and troubleshooting process, while decreasing unnecessary human effort and increasing the accuracy of the mapped troubleshooting activities.

1.3 Research Questions

This study investigates the possibility of classifying logs and suggesting a proposer action based on the failure causes. In this regard, the following research questions are answered in this thesis:

- **RQ1.** Which machine learning methods are appropriate to classify test case logs originated from a continuous integration pipeline?
- **RQ2.** What is the effectiveness, in terms of developer time reduction, of using the most appropriate classification solution?

1.4 Scope and Delimitations

During the process of software building and testing, one can encounter an almost infinite number of problems. This project will focus on grouping and classifying faults from test case execution logs. We specifically target one CI workflow at Ericsson with more than average failures. We do this to be able to collect and label data faster. Data have been collected, explored and labeled over a period of a couple of months and is a major part of the project. The data is limited to failing test cases that was produced during the project execution, since old logs are deleted due to storing constraints. We design and implement a pre-processing and pipeline for analyzing test case. The libraries that are we use that implement language models, dimensionality reduction and classifiers are referenced in Chapter 4. We hope to in the future extend this approach to more Jenkins test suit jobs at Ericsson, as the implementation can be used directly. Note that there are no barriers for implementing the same pipeline for another CI workflow. In this report we evaluate how well we can identify the different types of errors that occurs in the logs, with a supervised approach for a multiclass classification problem. We directly compare the classification performance. It would be feasible if we could produce a study to see how useful the tool is by sending a survey to developers. However, to develop a pipeline that integrates with the production is a project on its own. This means that we would have to estimate the time savings by consulting subject matter experts (SMEs).

1.5 Thesis Outline

The organization of this thesis is laid out as follows: Chapter 2 provides a background of the initial problem and an overview of research on log analysis and NLP, Chapter 3 describes some theories behind the conducted research. The structure of the proposed approach is depicted in Chapter 4. An industrial case study has been designed in Chapter 5. Threats to validity and delimitations are discussed in Chapter 6. Chapter 7 clarifies some points of future directions of the present work and finally Chapter 8 concludes this thesis. In Appendix A the hyperparameters for the models we use can be found.

Chapter 2

Background

This chapter presents a brief overview of the state-of-the-art research related to logs and logging, which is best summarized by looking at Table 2.1. Since this thesis is focused on log analysis, we present a summary of the past and current research within the area. We also mention relevant related research works within the area of NLP.

2.1 Log Analysis

Log analysis is about extracting knowledge from logs for a specific purpose, e.g. detecting undesirable behavior in a system, find the cause of system outage or analyze test cases [6]. It is challenging since the systems that produce the logs are complex and produce them for multiple purposes. Log analysis is further divided into multiple areas such as anomaly detection, security and privacy, root cause analysis, failure prediction, software testing, model inference, and invariant mining, and reliability and dependability [6] as in Table 2.1. Relevant areas for this thesis are anomaly detection, root cause analysis, and software testing. These are related since our goal is to classify the error type of the log events in a log file from a failing test case. We will regardless discuss neighboring topics to see how our work relates to the different sub-fields.

Log Engineering	*The development of an effective logging code.*	– Anti-patterns in logging code – Implementation of log statements – Empirical studies
Log Infrastructure	*Techniques to enable and fulfill the requirements of the analysis process.*	– Parsing – Storage
Log Analysis	*Insights from processed log data.*	– **Anomaly detection** ← *related of thesis* – Security and privacy – **Root cause analysis** ← *related of thesis* – Failure prediction – **Software testing** ← *related of thesis* – Model inference & Invariant mining – Reliability and dependability
Log Platforms	*Full-fledged log platforms.*	End-to-end analysis tools

Table 2.1 – An overview of the research topics related to logs and logging [6].

2.1.1 Log Anomaly Detection

Log anomaly detection is when techniques are used to detect undesirable patterns in log data. For example, a model is trained to only present these anomalies to a user by having a dataset with binary labels, abnormal or OK. An example of an anomaly detection techniques is the supervised model CloudSeer [11]. It compares temporal differences for different log events and evaluates if it is a normal execution flow. In their empirical tests they show an accuracy of > 92 % in detecting anomalies. DeepLog [7] has a similar strategy and claims that it works with logs that have multiple tasks executing and printing to the same log by using a Long-short Term Memory model. According to Candido et al. [6], there exist many other techniques within anomaly detection that aim for creating control flow graphs, finite state machines, doing dimension reduction, etc. Another work, LogAnomaly, modifies the Word2Vec algorithm into a method they call Template2Vec [12–14]. Shortly described, Word2Vec is a is an unsupervised predictive deep learning-based mode that learns the context of words and is described more in detail in Chapter 3. In their implementation they build a vocabulary of templates by first processing a list of synonyms and antonyms and use them to find log event templates, and then proceed to create WEs for the templates using Word2Vec. The templates are then matched with new data as it comes in [12].

2.1.2 Security and Privacy

The *Security and privacy* category is about prevent or detect intrusion and attacks on, for example, servers and databases. It also contains research regarding privacy logging, i.e. policies for what information is safe to log. Most of the logs analyzed here are network logs such as HTTP, router logs, etc. [6]. One study proposes a framework based on belief propagation, inspired from graph theory, to create a detector that searches web proxy logs to detect malware [15]. Another study uses Expectation-Maximization clustering to identify malicious activities by searching logs from DHCP servers, authentication servers, and firewalls. [16].

2.1.3 Root Cause Analysis (RCA)

Root cause analysis (RCA) is about detecting anomalous and unexpected behavior. Anomaly detection can highlight these log events, but a maintainer needs to investigate the given output. Root cause in this context can mean that we want to find the failing node, the failing job or failing software. That can be done by complementing logs with resource usage [17–19]. CRUDE complement the logs with resource usage and cluster nodes with similar behavior using hierarchical clustering. It use anomaly detection to detect jobs with anomalous behavior and an algorithm for linking these together. In their empirical evaluation they are able to detect 80 % of the errors [18]. Another algorithm, LogCluster cluster sequences of log events using Agglomerative Hierarchical clustering with their own distance measure designed for sequences of log events and match them with a knowledge base. The knowledge base is created by clustering known log events sequences of interest. When the available data is processed, the center log event of each cluster is set as the representation of each cluster, and a Subject Matter Expert (SME) put a label each cluster. To reduce the influence of log events with little value, they weights the them in a log with IDF (Inverse Term Frequency) [20].

2.1.4 Software testing

Software testing, in the context of log analysis, is about improving software development cycle when performing testing [6]. An example of such a work is LogCoCo, that estimates code coverage by analyzing execution logs and linking them to their corresponding code paths [21]. When evaluating the performance on 6 systems, they achieve above 96 % accuracy while estimating code coverage for methods, statements and branches.

2.1.5 Reliability, Dependability and Failure Prediction

Reliability and dependability is about estimating how reliable a software or hardware system is by digging in the logs. *Failure prediction* is used when faults have been found before and detect them by monitoring metrics. The last category is *model inference and invariant mining*. Model inference is the study of creating models from logs, such as state machines, client-server interaction diagrams or dependency models. State machines are used to detect bugs when the system does not act as intended [6]. A simple example of a software invariant is that the number of times a program open and closes a file should be equal. If a close statement is not present, then we conclude that something is wrong [22].

2.1.6 Log Event Template Extraction

The *Log parsing* step is very important and needs to be done in some way before the log is analyzed. The content in the log files need to be grouped so the dimension is reduced.

A common technique used is *Log event templates extraction*. It is about creating templates that matches different types of log events so that they are grouped. We will here go in a little deeper into the research in this area. Common for all these log event template extraction algorithms is that they first pre-process the logs by replacing uninteresting dates, urls, etc. with an identifier such as xxdate and xxurl [23]. One evaluation study evaluated the four log parsers SLCT (Simple Logfile Clustering Tool), IPLoM [24], LKE (Log Key Extraction), LogSig [23] and released corresponding open-source code implementation. They set out to study the accuracy and efficiency of the different log parsers and how effective they are on log mining and drew a couple of conclusions from their analysis on these tools.

- Current log parsing methods achieve high overall parsing accuracy (F1-score).

Log Parser	Year	Technique	Mode	Efficiency	Coverage	Preprocessing	Open source	Industrial Use
SLCT	2003	Frequent pattern mining	Offline	High	✗	✗	✓	✗
AEL	2008	Heuristics	Offline	High	✓	✓	✗	✓
IPLoM	2012	Iterative partitioning	Offline	High	✓	✗	✗	✗
LKE	2009	Clustering	Offline	Low	✓	✓	✗	✓
LFA	2010	Frequen tpattern mining	Offline	High	✓	✗	✗	✗
LogSig	2011	Clustering	Offline	Medium	✓	✗	✗	✗
SHISO	2013	Clustering	Online	High	✓	✗	✗	✗
LogCluster	2015	Frequent pattern mining	Offline	High	✗	✗	✓	✓
LenMa	2016	Clustering	Online	Medium	✓	✗	✓	✗
LogMine	2016	Clustering	Offline	Medium	✓	✓	✗	✓
Spell	2016	Longest common sub-sequence	Online	High	✓	✗	✗	✗
Drain	2017	Parsing tree	Online	High	✓	✓	✓	✗
MoLFI	2018	Evolutionary algorithms	Offline	Low	✓	✓	✓	✗

Table 2.2 – Summary of automated log parsing tools. Note that most of them are not for industrial use [25].

- Simple log pre-processing using domain knowledge (e.g. removal of IP address) can further improve log parsing accuracy.
- Clustering-based log parsing methods could not scale well on large log data, which implies the demand for parallelization.
- Parameter tuning for clustering-based log parsing methods is a time-consuming task, especially on large log datasets.
- Log parsing is important because log mining is effective only when the parsing accuracy is high enough.
- Log mining is sensitive to some critical events. Around 4% errors in parsing could even cause an order of magnitude performance degradation in log mining.

In a later paper, they extended the open-source code and the analysis by also evaluating AEL, LFA, SHISO, LogCluster, LenMa, LogMine, Spell, Drain, and MoLFLI [25] and a summary is visible in Table 2.2.

During the development of this work, we implemented different log parsers such as those mentioned in Table 2.2. However, when we used log template extraction on our logs, it gave us too many templates (in thousands), hence it1 was not useful. Therefore, we focused more on NLP-related techniques related to text classification. The details of our research methodology are described in Chapters 3 and 4.

2.2 Related Work

While reviewing the different topics mentioned above, we see that not much work have been done within the field of multi-class log classification, as logging systems often trigger very specific errors [6]. In our case, we want to categorize the type of fault in logs originated from an execution of tests cases, so it is possible to suggest troubleshooting actions. For example, instead of binary classification, one could use more labels such as timeout, build error, HTTP request error, etc. This work is related to anomaly detection, software testing and root cause analysis but also related to NLP. Therefore, we review works related to NLP and classification of test cases here. Root cause Analysis in logs for our context can include steps such as log template extraction, pre-processing, feature engineering, topic modeling, clustering, translation to word embeddings (WEs), classification, etc. depending on how one decides to solve the problem.

2.2.1 Feature Engineering

To classify test case log files, many different variants of feature engineering and features are used as input to different classifiers. There is no standard for feature selection and most of the investigated studies try different types of features. Recently, a similar master thesis report was published where they tried to divide the error logs into users or infrastructure problems, i.e. binary classification. They used Term Frequency–Inverse Document Frequency (TF-IDF) as input to different classifiers such as SVC, Gradient Boosting, Random Forest [26]. Another study builds category dictionary libraries using TF-IDF and then use Levenshtein Distance [27, 28] to measure semantic similarity. Later they show that deep convolutional networks have a better classification performance than other simpler classifiers based on the given feature input [29]. Another study at Ericsson, that has a similar goal to the one in this study, use features such as the number of containers invokes (which execute the tests), number of responses, errors, trace-backs, and warnings in the log, success rate per build and overall test case success [30]. Another way is to just monitor resource usage to classify different types of errors and correlate it with the different types of error messages in the logs [31]. Yet other works use the timestamps of the log event to find patterns in failing logs [32, 33], by evaluating the timing in the order of the log events. N-grams are also very common, for both words and characters [34]. A note can be made to Word2Vec, which has two models (skip-gram, CBOW) to turn words into WEs. The output of these models can be feed to a classifier [35]. The presented used features in the papers discussed in this paragraph all present promising results, but they cannot be directly compared since they all use different data. In the mentioned papers in this paragraph, classifiers such as linear regression, random forest, gradient boosting, LSTM, Convolutional neural networks (CNN), Word2Vec are used.

2.2.2 Natural Language Processing

If we look at the field of NLP, there has been great progress within deep learning [36], where we observed the same type of progress as computer vision had a couple of years ago. The previously mentioned work DeepLog uses the LSTM model [7]. LogAnomaly use a modified variant of Word2Vec to learn WEs that provide a numerical representation of the content in logs [12]. Both are deep learning models. The benefit of the LSTM model is that it takes sequential input. The benefit of Word2Vec is that it transforms words into a meaningful representation in the space of embeddings. The simplest example, that is not related to test case logs, is constructed by using addition and subtraction to see how word representations relates: King - Man + Woman = Queen. A more in-depth of Word2Vec is present in Section 3.2.3. If more resources are available, pre-trained language models such as GPT-2 [37], GPT-3 [38], BERT [39], XL-Net [40] and ULMFiT [41] can probably be used in a similar way. They learn to model language by training on very large corpus datasets such as filtered snapshots of Wikipedia [41]. ULMFiT shows in its paper that their model can exploit pre-trained models to learn a representation of another very small new dataset with little training. The models have reached a new level in text generation [38], text classification and transfer learning (with small datasets) [41, 42], etc. The problem with these models is they require very large computational resources where, for example, GPT-3 requires a large cluster of computers to execute [38]. There are works that try to extract the essence of these large models by distilling the deep learning models, so it is possible to execute them with less resources. Distilling means that parts of the weights in the deep learning models are discarded but still performs very well on similar tasks. Such an example is distilBERT, which is a distilled version of BERT and is deployable on a single machine [43].

As the area of log analysis is expanding and it also benefit from research outside its specific area. Since all logs mostly contain written text, any model that learns to represent the meaning of the log events with word embeddings can be used to improve the analysis. Table 2.3 represents more related work in the area of log analysis and troubleshooting, where the employed method and drawback of each work is specified.

Reference	**Purpose of paper**	**Limitations**
Kc and Gu [44]	Using hybrid log analysis and clustering	Requires several predefined transition patterns between different types of messages (unsupervised learning)
Jiang et al. [45]	Using the characteristics of the customer problem	Limited to the costumer cases
Mochizuki et al. [46]	Searching for keyword file corresponding to trouble represented by entered character string	Does not provide the troubleshooting activities (just searches for and displays the related logs for troubleshooting)
Winnick [47]	Using a series of decision trees that are used to guide the user through troubleshooting.	Does not provide the troubleshooting activities (it generates questions for the user by a system diagnostic engine to determine a problem to be solved for a target system)
Debnath et al. [48]	Running a program code to generate seed patterns from the preprocessed logs.	Does not provide the troubleshooting activities (it generates final patterns by specializing a selected set of fields in each of the seed patterns to generate a final pattern set.)
Jain et al. [49]	Performing phrase extraction on the text to obtain a plurality of phrases that appear in the text	Is limited to the predefined phrases
Purushothaman et al. [50]	Using a ML computing system	Does not provide the troubleshooting activities and it just identify an associated error condition category
Jadunandan et al. [51]	Using a communication network operations center (NOC) management system.	Requires the equipment trouble history data
Vidal et al. [52]	Using unsupervised learning technique	Does not provide the troubleshooting activities and it detects just the test flake
S. Cai et al. [53]	Using NLP	Using an unsupervised learning and it does not provide the troubleshooting activities
Y. Li et al. [54]	Using NLP	Provides a sentiment analysis and it does not provide the troubleshooting.

Table 2.3 – Summary of relevant related work.

Chapter 3

Theory

This chapter gives a brief introduction to all methods and metrics used in this thesis. Just as the works presented in the beginning, we need ways to transform the content of the test case log files into a representation with meaning. We will focus on using methods for transforming the text in each log event into WEs. We then use dimensionality reduction to transform the WEs into a low-dimensional space. At last we use classifiers to perform inference on the data we have. In short, the chapter is structured in the following way. In the first half, dimensionality reduction and NLP based techniques are described. In the second half, we shortly introduce the models that we compare and evaluate.

3.1 Dimensionality Reduction

Dimensionality Reduction is used when data needs to be transformed from a high-dimensional space to a low-dimensional space. There are multiple reasons why one would want to do dimensionality reduction. Such a reason could be removing dimensions with low influence on the data, represent the data in other coordinates, etc. In ML when the data has more dimensions than data points we suffer from the curse of dimensionality. Training an algorithm to learn the representation will then lead to severe overfitting, since the model only learns to represent the data points in the data set. This leads to weak performance when performing inference. Note that there is also a possibility to remove too much information from the data when doing dimensionality reduction [55]. In this work we decided to use Uniform Manifold Approximation and Projection (UMAP), which is what we will describe next.

Uniform Manifold Approximation and Projection (UMAP) is a dimensionality reduction technique that is used for general non-linear dimensionality reduction. It relies on three assumptions: That the data is uniformly distributed on Riemannian manifold, that the Riemannian metric can be approximated as locally constant, and that manifold is locally connected. Based on these, the manifold is modeled with a fuzzy topological structure and the embedding is extracted by finding the low dimensional projection of the data that is the closest to the structure [56].

3.2 Data Representation Techniques in NLP

Employing NLP techniques in software testing has received a great deal of attention recently, since deep learning techniques have been able to create a better representation of text [57–60]. In this chapter we will go through the variant of Word2Vec, called FastText, that we use to create WEs. We will also go through the simple TF-IDF that we later use to weight the different word embeddings. To utilizing NLP techniques, we need to find a way to represent our data (a series of texts) to our systems (e.g. a text classifier).

3.2.1 Term Frequency and Inverse Document Frequency (TF-IDF)

TF-IDF is a statistical measure that is used as a type of weight mostly in text mining. It weights the number of times a word appears in the document proportionally but also includes an offset from how often the word is used in the whole corpus. One of its use cases is to find stop words. The Term Frequency for a word in a document is normalized by considering the document length. The Inverse Document Frequency considers how often words appear in the whole corpus so that words that appear in the whole corpus is scaled down, and word specific to a few documents is scaled up [61]. In more mathematical terms, we define Term Frequency to be

$$TF(t) = \frac{\textit{Number of times term t appears in a document}}{\textit{Total number of terms in the document}} \tag{3.1}$$

and Inverse Document Frequency to be

$$IDF(t) = \frac{\log \textit{(Total number of documents)}}{\textit{Number of documents with term t in it}} \tag{3.2}$$

The TF-IDF weight is the product of these values

$$\textit{TF-IDF(t)} = \textit{TF(t)} \times \textit{IDF(t)} \tag{3.3}$$

3.2.2 N-grams

N-grams in the context of NLP refers to a contiguous sequence of n items from a text. Instead of making a word a feature, the contiguous n word is the feature. This is used to get more context from each word, but the dimensions of the n-gram words increase exponentially as n increase [34]. A simple example of word 2-gram is here presented.

This is an example $\rightarrow$ *<This, is>, <is, an>, <an, example>*

N-grams are also constructible from the character in a word. Here is a simple character 2-gram example:

example $\rightarrow$ *<ex>, <xa>, <am>, <pl>, <le>*

3.2.3 Word2Vec

Word2Vec is an unsupervised predictive deep learning-based model. It is shallow since it only uses 2 layers in its NN. It generates continuous dense vector representations of words, that capture semantic and contextual similarity. Word2Vec leverage either the Continuous Bag Of Words (CBOW) model or the Skip-gram model to create the WE representations, and they are described in the subsections below. Words that are more similar in context will be closer in the WE space than words from a different context [13].

The original implementation use hierarchical SoftMax as output unit and represent the vocabulary as a Huffman binary tree. A Huffman binary tree assigns short binary codes to common words which in this case reduce the number of output units needed in the NN.

3.2.4 FastText

The *FastText* [34] model considers each word as a Bag of Character n-grams instead of word n-grams. This helps with languages that have many compositions of the same word. In the Word2Vec model where each found word is handled as it's a separate vector. With this model, more rare words have a better chance of getting a good representation since the character n-gram occurs more often than the word itself [34]. The creators of FastText [34] recommends extracting all character n-grams with $3 \leq n \leq 6$.

FastText utilize the Continuous BOW model and the Skip-Gram model creates a numerical representation of the words. The closer they are in the numerical space, the closer they are in context and meaning.

Normally when doing text analysis, lemmatization and stemming are used to reduce the number of different words. Lemmatization uses language rules to match words of the same meaning and stemming cuts off the end of the words to match similar words. The former is better if there such a model available, but that might not be the case. To instead use character n-grams, in the context of logs, is very useful since it's possible capture the meaning of log events better. Since log events contain variables, values, etc. this means that we get a representation of a never seen variable name before.

3.2.5 Continuous Bag of Words (CBOW)

The *CBOW* model is an unsupervised neural network (NN) language model that predicts the current target word (the center word) based on the surrounding words, that act as context. Compared to a NN language model, the non-linear hidden layer is removed so that the projection layer in the NN is shared for all the words it trains on. The model uses the corpus as training data by keeping out the current target word and predict and compare the result to the corpus. CBOW does not care about the order of the words (hence BOW), since it averages out the WEs of the surrounding words [13]. An example of input and output of the CBOW model is shown in Figure 3.1.

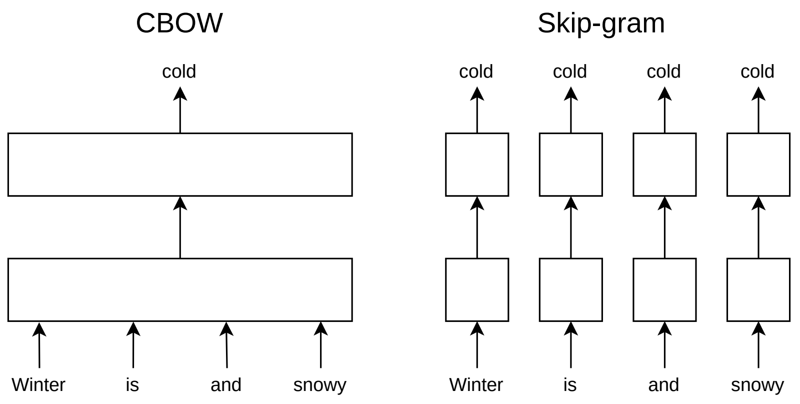


Figure 3.1 – An example of the input and output of the CBOW and Skip-gram leveraged by Word2Vec models such the FastText model. The rectangles represent layers in a ANN.

3.2.6 Skip-Gram

The *Skip-Gram* model could be described as the inverse of CBOW. The skip-gram model is an unsupervised NN language model that takes a word (input word) in the middle of a sentence and will predict the words that are most likely to be close to this word (surrounding words). The output of the model will be the probability for all the words in the vocabulary and during training these outputs are trained to represent nearby words [13] [14]. An example of input and output of the Skip-Gram model is visible in Figure 3.1. The architecture is built like auto-encoders where we train a full network but are only interested in the hidden layer weight matrix that has learned a smaller representation of the data [13].

In the model, which gives us the goal to maximize the following log-likelihood:

$$\sum_{t=1}^{T} \sum_{c \in C_t} \log(w_c | w_t) \tag{3.4}$$

where we want the WE for the words $w \in \{1, ..., W\}$. C_t is the context words for word w_T. It is the probability of observing a context word w_c given w_t. In the Word2Vec model, they frame the problem as a set of independent binary classification tasks. For word w_t the context words are framed as positive examples and random words from the dictionary as negative samples which leads to the following negative log-likelihood:

$$\log(1 + e^{-s(w_t,w_c)}) \sum_{n \in \mathcal{N}_{t,c}} (1 + e^{-s(w_t,n)}) \tag{3.5}$$

With each context position c, $\mathcal{N}_{t,c}$ is a set of negative examples sampled from the vocabulary [34].

In the FastText model each word is represented as a bag of character n-grams. This means that each word is represented by the sum of the vector representations of its n-grams. This allows representations to be shared among different words [34]. With an associated vector representation z_g, to each n-gram g, the scoring function s becomes defined as:

$$s(w, c) = \sum \mathbf{z}_g^\intercal \mathbf{v}_c \tag{3.6}$$

where v_c is the vocabulary vector [13, 34].

3.3 Machine Learning Models for Classification

As with all ML problems, we need algorithms that learns to differentiate the input data, supervised or un-supervised. In this study we focus on a supervised problem. The input to our classifiers will be WEs and the output will be the class labels that represent each category of error types. We will here introduce the classifiers we will use throughout the study: Logistic Regression, Support Vector Machine (SVM), Random Forest, Gradient Boosting, Multi-Layer Perceptron (MLP) and LSTM.

3.3.1 Logistic Regression, SVM, Random Forest, Gradient Boosting, MLP

In *linear regression*, the input and output of the model are linked using linear variables, i.e. each variable in the input of the data is multiplied with a scalar value. A common way to fit the model is to update the weights with the least-squares approach. With the use of a cost function, one can also use lasso (L^1) or ridge regression (L^2) to improve the generalization of the model.

Random Forest are an ensemble learning method for both classification and regression where the forest is made up of decision trees. Each tree is trained on a subset of the data and/or a subset of the variables. The data is divided into each level of the decision tree based on what gives the best split for the given data points. The prediction result on new data of each trained decision tree in the ensemble is combined. A low correlation between different decision trees is achieved when using different features and data points for training for each tree. It levels out the errors of each individual tree.

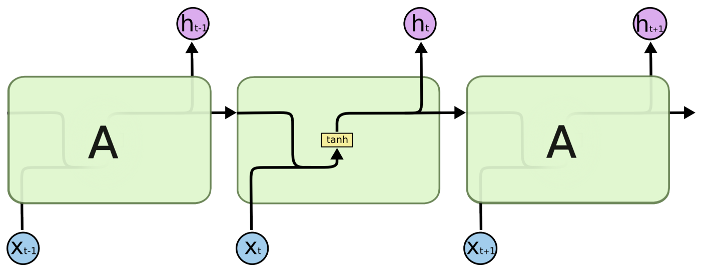
Gradient Boosting is a ML algorithm that is used for both regression and classification problems. It trains an ensemble of weak prediction models. It uses boosting, i.e. it utilizes weighted averages to make weak learners into stronger learners. Boosting helps with reducing the variance in the prediction and results in a model with higher stability. One implementation is where one weak classifier is added one at a time and are weighted relative to the weak learns accuracy. The weights are normalized after each added learner is added. The gradient part of gradient boosting refers to the use of training the ensemble using gradient descent [62].

Support Vector Machine (SVM) is a supervised algorithm for classification and regression problems and is very popular due to its ability to classify with margins between classes. It's a vector space model that finds the decision boundary between two classes that are as far as possible from the data points [63, p. 320]. The data points close to the hyperplane that splits classes are called the support vectors.

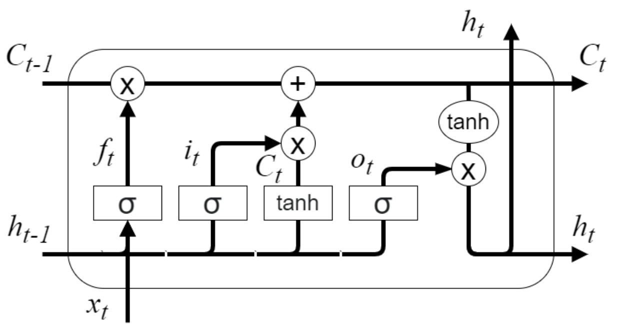
Multi-Layer Perceptron is a feedforward artificial NN (ANN), that contains at least an input layer, a hidden layer, and an output layer. MLP uses backpropagation to update its weights between all nodes. With non-linear activation functions, and with multiple layers, a non-linear mapping is learned during training.

3.3.2 Long Short-Term Memory

Long Short-Term Memory (LSTM) is a famous Recurrent Neural Network (RNN) which is used to create deep learning models [64]. The recurrent part



(a) The repeating module in a standard RNN contains a single layer [65].



(b) The inner workings of LSTM.

Figure 3.2 – The circles with an operator is a point-wise operation, the arrow means vector transfer, and arrow with two input paths is concatenation.

makes it possible for the model to process sequences of data. This is very beneficial when processing text, video, time series prediction, etc.

A simple RNN that use backpropagation to update its weights, as the one in Figure 3.2a, have the problem of vanishing/exploding gradients just as normal deep feed-forward networks have. So, while an RNN identifies the next word that only depends on the previous data points, in practice we note that it fails when the context is given in the further back in time. LSTM solves improves on this since it is better at remembering long-term dependencies [65].

Each LSTM unit contains multiple parts that define how the data flows through the cell as in Figure 3.2b: input gate (i_t, C_t), output gate (o_t), forget gate (f_t) [65]. These four cells together form a memory of the cell, by and regulate the internal state. The forget gate controls what information needs to be thrown away from the cell state. The input gate controls what values to update within the unit. The output gate controls what parts of the cell state we will let through.

3.4 Validation Metrics

We will measure the performance of the proposed solution by comparing the inferred results from the system with the labels of each test case log, given by the Subject Matter Experts (SMEs). This means that we are dealing with a supervised problem.

3.4.1 F1-score

To evaluate the classification, we use the *F1-score*, which is a combination of *recall* and *precision*. Phrased in a binary classification case, recall is the number of correctly identified positive results, divided by all results that should have been classified as positive. Precision is the number of correctly identified positive results, divided by the sum of the number of correctly identified positive results and the number of data points incorrectly classified as positive.

The equation of F1-score is

$$F_1 = \frac{2}{recall^{-1} + precision^{-1}}$$

Note that recall and precision have the same weight. The two can be weighted differently, depending on the importance of each factor. In this case, the definition of F1-score is:

- **Precision:** the number of correctly detected classes over the total number of detected classes by each method.
- **Recall:** the number of correctly detected classes over the total number of existing classes.

where the F_1 represents the harmonic mean of precision and recall. We choose to use F_1-score since the dataset was heavily imbalanced in the number of data points per class, which is viewable in Table 4.1.

To evaluate a more realistic performance of our models, we use k-fold and stratified K-fold cross-validation. When doing k-fold cross-validation, the dataset is split into K equally sized parts. One part is kept out for testing the performance of the model, and the other parts are used to train the model. This is repeated k times, one for each split. A stratified k-fold change so that the different classes in the dataset is divided evenly between the k parts. The cross-validation is presented together with the mean and standard deviation for multiple k-fold cross-validation executions for different seeds. This gives a more realistic picture of how the model would perform in production.

3.4.2 Wilcoxon signed-rank test

Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used to compare samples. It is used to compare if their population mean ranks differ. We will use it to compare if there is a statistically significant difference between the performance of the different classifiers, where the null hypothesis that the classifiers have equal F1-score. It assumes that data are paired and come from the same distribution and the pairs are chosen randomly and independently [66].

We choose a significance level of 0.05, that will decide whether we reject the null hypothesis or not. This means that we have a confidence level of 95 %. The algorithmic description is described in Wilcoxon's work [66].

3.4.3 Friedman NxN test

The Friedman test is a non-parametric statistical test [67]. We will use it to rank the results for the different classifiers. It works in the following way: Given a matrix with the dimension $\mathcal{R}^{n \times k}$ with data $\{x_{ij}\}_{n \times k}$, with n rows (blocks, or measurement) and k columns (treatments or algorithms) the ranks are calculated within each block. The matrix is then replaced with a matrix $\{r_{ij}\}_{n \times k}$ where r_{ij} is the rank of x_{ij} within block i. Then the values

$$\bar{r}_j = 1/n \sum_{i}^{n} r_{ij}$$

are found. The test statistic is

$$Q = \frac{12n}{k(k+1)} \sum_{j=1}^{k} \left(\bar{r}_j - \frac{k+1}{2} \right)^2$$

In the last step the probability distribution of Q is approximated with the chi-squared distribution when n or k are large. If the p-value is significant, post-hoc multiple comparisons test should be performed in order to check for statistically significant difference between each other [67]. The steps for this can be found in Friedman's work [68].

3.5 Summary

In this chapter we have introduced the concepts of dimensionality reduction using UMAP, techniques for text representation using FastText (Word2Vec) and shortly described a set of classifiers we use in this study. In the next chapter we will describe our pipeline and how we use these techniques to map the test case log files to the different class labels. In short we use a Word2Vec model (FastText) to transform the words in each log event into WE. We use TF-IDF to weight the influence of each word so unique words, that are specific for a class, get a higher weight. We then use UMAP to reduce the dimensionality of the word embeddings. We then use the described classifiers to evaluate if the transformed data represent the classes by training algorithms and performing inference.

Chapter 4

Methods

This chapter provides more details regarding the utilized methods for solving the log analysis problem addressed in this thesis. First, we present the pipeline in Section 4.1. Then, we describe the data collection methodology and the data preprocessing in Sections 4.3 and 4.2. Later, we give details regarding the model selection and model training.

4.1 Pipeline

Figure 4.1 shows an overview of the proposed solution for mapping a proper troubleshooting activity for a failed test case log.

The details of each step is specified in the following steps:

1. **Clean and filter the logs.** The first step is to preprocess the logs into a more straightforward format. Here unnecessary things such as IP addresses, web addresses, dates, digits, special characters, capital letters, etc. that are not needed in the analysis are replaced with identifiers such as xxip, xxdate etc. For more details, see Section 4.3.

2. **Extract log events based on a failure identifier.** The models that learn WEs train on all the defined input. If possible, the performance of the classifiers can be improved by selecting log event groups that contain key identifiers and thereby limiting the amount of input data for each test case log. In Figure 4.1 and an example is presented, with words

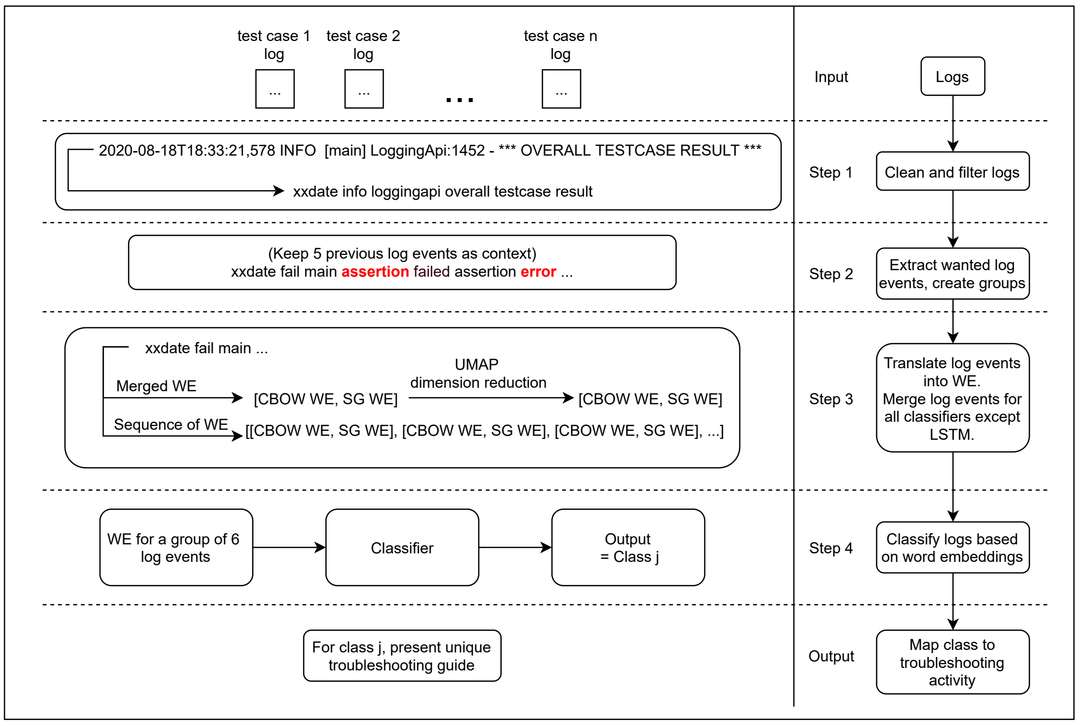

test case 1 log
test case 2 log
test case n log
Input
Logs
2020-08-18T18:33:21,578 INFO [main] LoggingApi:1452 - *** OVERALL TESTCASE RESULT ***
xxdate info loggingapi overall testcase result
Step 1
Clean and filter logs
(Keep 5 previous log events as context)
xxdate fail main assertion failed assertion error ...
Step 2
Extract wanted log events, create groups
xxdate fail main ...
Merged WE
[CBOW WE, SG WE]
UMAP dimension reduction
[CBOW WE, SG WE]
Sequence of WE
[[CBOW WE, SG WE], [CBOW WE, SG WE], [CBOW WE, SG WE], ...]
Step 3
Translate log events into WE. Merge log events for all classifiers except LSTM.
WE for a group of 6 log events
Classifier
Output = Class j
Step 4
Classify logs based on word embeddings
For class j, present unique troubleshooting guide
Output
Map class to troubleshooting activity

Figure 4.1 – The required input, steps and expected output of the proposed methodology in this thesis. In the figure.

marked in red. For each found log event, five previous log events are kept for adding context. For more details, see Section 4.3.

3. **Transform log events into WEs.** At this step, we transform the input to WEs before feeding it to the classifiers. All classifiers get both the FastText CBOW and Skip-Gram WEs representation as input in the same vector. More specifically, the CBOW part is stored in the first part of the vector and Skip-Gram in the other part. For all models, except for LSTM, the WEs are first reduced using UMAP and then merged by weighting with tf-idf. The LSTM classifier that takes the WEs directly. The transformation is described in 4.3.1.

4. **Classify the logs based on their WEs.** In this step, the WEs are sent into the classifier.

5. **Map a proper troubleshooting activity with each class.** Each class is linked to a unique troubleshooting activity. Depending on what action needs to be taken, an automated action is launched to solve the issue, or, a message with an action plan is sent to the affected SME.

Class ID	**Description**	**Number of data points**
1	Unlock/lock operation failed	78
2	Too high packet loss	30
3	Failed to power on/off unit	201
4	Node not enabled	78
-1	Unknown	109

Table 4.1 – The number of data points per class that was collected for this project.

4.2 Data Collection

The dataset used in this thesis was gathered at Ericsson. It is produced by the execution of a Jenkins job that executes tests for a set of products. The data was collected by issuing three surveys. In total, 767 failing test case logs were collected from the mentioned Jenkins job. The number of log entries produced is in the size of gigabytes for each test suit, so we limited the work to include the logs produced by the internally developed program that is called by Jenkins. These logs contain everything that is specific to the test case but leaving out the logs for each product included in the test. This means that the test case log is where the SME would look first to classify what type of fault it is.

With the help of SMEs, the data was labeled into 16 different classes that needed a unique troubleshooting activity. The dataset at hand is very unbalanced, so we settled to train on the four classes that contain most of the data points. The rest of the data was relabeled as class -1, which represents the unknown. The motivation for this is that the SMEs that would use a tool like this would like to know if the classifier recognizes the fault, or if there is any uncertainty. When using deep learning models, an alternative way would be to change so that we require the output of a final SoftMax layer to be high enough for a certain class. The unknown class proved to be very helpful in identifying unknown classes when we trained on previous surveys and made inference on a new survey. The number of data points in each class are presented in Table 4.1.

Note that we have a class of -1. This is a class where we merged the data points that we do not have enough data for. In this case, $109/496$ is in the class -1 and contains samples from 12 classes, and data points that SMEs labeled that they need more context.

Test case identifier	Log Events	Group ID	Class label

(test case 1)
Jenkins build id/
sub-system job id/
test suit id/
test case id/
	`xxdate info connection with unit initiated`
	`xxdate debug checking status of unit`
	`xxdate warn could not find ...`
	`xxdate debug message status ...`
	`xxdate fail failed to connect to unit ...`
	`xxdate assert assertion failed ...`
		1	1

(test case 2) ...	`xxdate debug ...` ...	2	3
...	...	...	...

Table 4.2 – An example of how the extracted groups of log events look like after filtering the failing test case logs.

4.3 Data Preprocessing

We start by selecting only the failing test case logs. In these logs, we replaced dates, IP addresses, URL's, file paths, citation marks, memory addresses, and digits with identifiers such as `xxdate`, `xxip`, `xxurl`, `xxfile`, etc. We also remove words from the data using a stop word list, since the number of data points in our dataset is relatively small. We extract the Java stack traces from the log events and keep them in a separate column, essentially removing them from the input data. In the log event, we replace it with `xxstack-trace`.

After that, we searched for higher-order log events such as `ASSERT`, `FAIL`, and `ERROR`. From these log events, we searched for some key identifiers given to us by the SMEs. For each highlighted log event, we kept the five previous log events to add context, and these messages could be of any type. The selection of five log events came from discussion with the SMEs, where they said that most of the relevant information is found within the five previous log events. Given the above, we have entries containing six log events with the last one being the high-level log event that we triggered collection from. These groups of log events were the input to our models. There can be multiple groups of log events with different errors within the file since each test case log file can contain multiple errors. By examining the log events we could see that most of the important information was present at the beginning of the log events, so we keep the first 100 words from each log event. When the log events contain more than 100 words, they often contain stack traces, JSON responses, memory dumps, etc., hence, do not add any valuable information to the classifier. Examples of how the data look after the preprocessing step is presented in Figure 4.1 and Table 4.2.

Log events	`xxdate assert crash` ... `xxdate debug connection` ... ...
WEs	[[0.1, 0.2], [3.2, 5.8], [-5.1, -3.2], ... , [0.1, 0.2], [-2.0, -1.5.0], [-10.1, 8.0], ...]

Table 4.3 – Concatenated WEs that are used as input in models that receive sequential input.

Log event priority	`assert`	`fail`	`error`	`info`	`debug`
Number of log events	1	1	1	2	2

Table 4.4 – The position and number of log events kept in the feature vector.

4.3.1 Word Embeddings

The WEs are created using FastTexts' CBOW and Skip-gram models by training on preprocessed data. For the LSTM model, the classifier will get the input from both CBOW and the Skip-Gram embeddings in a concatenated word vector. This means the feature vector has the form for a log event as shown in Table 4.3.

Note that in Figure 4.3 the numerical representation of the words is made up. It is an example of how it represented with 1-dimensional word vectors, and the dimension can be chosen freely. Here, the first item in the WE vector for each word represents the CBOW model representation, and the later represents the Skip-gram representation.

For all classifiers except LSTM, the WEs above are merged since these classifiers are not designed for sequential data. The WEs for each log event are averaged by using TF-IDF to weight the importance of each word. After they have been merged, the dimension of the WEs is reduced using UMAP. We use a separate UMAP model for the CBOW and Skip-Gram embeddings.

The log events priority (`assert`, `fail`, `error`, etc.) are then used to put each type of log event into a fixed position in the feature vector. This helps when log events are presented in a different order. In the input feature vector, the different types of log events will have the position that is seen in Table 4.4. As the table show, we keep the first assertion/fail/error message, and the first two info/debug messages. Depending on how many log events you want to include in each group, this needs to be changed to reflect that.

4.4 Model Training

After the preprocessed data has been converted into WEs, they are feed into a classifier. In this work, we evaluate Linear Regression with L^2 regularization, Random Forest, XGBoost, SVM, MLP, and LSTM. All but the LSTM will get WEs created by merging log events as described in Section 4.3.

4.5 Model Selection

To search for the hyperparameter space, we use a grid search. Similar performance could be achieved with most of the hyperparameters we choose for all the models except LSTM where the number of LSTM nodes has a huge effect on the number of parameters. For Random Forest, XGBoost, etc., we choose a depth of two and three, to decrease the risk of overfitting. For the LSTM model, we tried to decrease the number of nodes to the lowest possible. Note that a much more in-depth hyperparameter search could have been done, but since the distribution of word events change as tests are executed, it is not a meaningful search as the models would be likely to overfit to the data at hand.

4.6 Hardware Setup and Used Software Libraries

To test the different models, a laptop supplied from Ericsson was used. Its specifications are: Intel i5-8350U CPU @ 1.70GHz with four Cores and eight Logical Processors, 16GB DDR4 memory, Windows 10 Enterprise.

Libraries used include Python 3.7.1, Scikit-learn (random forest, linear regression, SVM, MLP, F1-score) [69], UMAP [56], FastText [34], XGBoost (Gradient Boosting) [62], and Tensorflow using Keras (LSTM) [70]. XGBoost is an open-source library that implements a gradient boosting framework for several different languages. To calculate the Friedman NxN test we use KEEL [71, 72].

Chapter 5

Results

In this section, we present the results obtained in this thesis. To compare the performance of the different machine learning algorithms, we designed an industrial case study at Ericsson AB in Sweden. To do so, we follow the guidelines proposed by Runeson and Höst [73] and Tahvili [2]. Note that the results are discussed in Chapter 7, and it is where we discuss and compare the classifiers performance.

5.1 Unit of Analysis and Procedure

The units of analysis in the case under study are all designed in the way that the result of each classifier is presented in stratified 3-fold cross-validation, together with a graph of the mean and standard deviation for multiple 3-fold executions. The case study was performed in four steps:

Step 1: During a couple of months logs from failing test cases were collected. 767 failing test case executions were selected, and their corresponding log file was extracted.

Step 2: The logs were pre-preprocessed according to the description in Section 4.3. Dates, urls, stacktraces, etc. were replaced with an identifier such as `xxdate`, `xxurl`, `xxxstacktrace`. Higher-order log events such as `assert`, `fail` and `error` are selected and we keep those containing key identifiers, together with the 5 previous log events before them. These log events form a group that needs to be classified and they were labeled by the SMEs.

Step 3: The filtered groups of log events are transformed into WEs using FastText, as described in Figure 4.1 in Chapter 4. For all the classifiers except for the LSTM, the words in each log event was added together by weighting each word embedding using TF-IDF. We then used the dimensionality reduction algorithm UMAP for all classifiers but LSTM to reduce the number of dimensions for each word event to prevent overfitting.

Step 4: The classifiers were evaluated with 100 3-fold cross-validations and a stratified 3-fold cross-validation. Note that the class -1 described as an unknown class, since we have put the classes with too little data into it. Each classifier was tested with 100 3-fold cross-validations and a stratified 3-fold cross-validation.

5.1.1 LSTM Classifier

The LSTM model has an input layer that is connected to a bidirectional LSTM layer with 32 units and 0.4 in the dropout rate. The LSTM layer is then connected to a SoftMax layer that also uses dropout with the same rate in the connections between the layers. Categorical cross-entropy is used as a loss function and Adamax as an optimizer. Adamax is chosen due to empirical results to show better performance with WEs than with Adam optimizer [74]. The learning rate is initially defined to 0.1 and is exponentially lowered by multiplying it with 0.97 after every epoch and is then constant after 200 epochs have been executed.

The input was the WEs created for sequential models described in Section 4.3.1. The results for the LSTM model is presented in Sections 5.11 and 5.12.

5.1.2 Classifiers with Merged Word Embeddings

The other classifiers used merged WEs to distinguish the different classes. A small grid search was performed to find the best hyperparameters. The choice of hyperparameters did not affect the results much for the different classifiers. Therefore, the least complex (smaller depth, fewer parameters, etc.) set of hyperparameters that was at least within 1% from the top-performing classifiers F1-Score. Note that a much bigger hyperparameter search could be done, but would not really give any better hints about the real performance since the distribution of the errors shift as more data comes in.

Class	Precision	Recall	F1-Score	Support
-1	0.37	0.42	0.39	36
1	0.74	0.77	0.75	26
2	0.92	0.98	0.95	10
3	0.70	0.57	0.63	67
4	1.00	1.00	1.00	26
Accuracy			0.78	258
Macro Average	0.75	0.75	0.74	258
Weighed Average	0.78	0.78	0.77	258
Class	**Precision**	**Recall**	**F1-Score**	**Support**
-1	0.71	0.68	0.69	37
1	1.00	1.00	1.00	26
2	0.97	0.97	0.97	10
3	0.86	0.88	0.87	67
4	1.00	1.00	1.00	26
Accuracy			0.91	258
Macro Average	0.91	0.91	0.91	258
Weighed Average	0.91	0.91	0.91	258
Class	**Precision**	**Recall**	**F1-Score**	**Support**
-1	0.30	0.36	0.33	36
1	0.83	0.73	0.78	26
2	0.93	0.64	0.76	10
3	0.78	0.88	0.83	67
4	0.56	0.96	0.70	26
Accuracy			0.70	257
Macro Average	0.68	0.71	0.68	257
Weighed Average	0.75	0.70	0.71	257

Table 5.1 – The obtained results using linear regression with L^2 regularization and stratified 3-fold cross-validation.

Class	Mean	std dev.
-1	0.713	0.056
1	0.906	0.045
2	0.946	0.015
3	0.852	0.030
4	0.949	0.032
All classes	0.87	0.036

Table 5.2 – The mean F1-Score for linear regression with L^2 regularization for each class, evaluated on 100 3-fold runs, with different seeds.

The result for the different classifiers is presented in the following tables: Linear Regression: Table 5.1 and 5.2. Random Forest: Table 5.3 and 5.4. XG-Boost: Table 5.5 and 5.6. SVM: Table 5.7 and 5.8. MLP: Table 5.9 and 5.10. Their hyperparameters is presented in Appendix A.

Class	Precision	Recall	F1-Score	Support
-1	0.34	0.75	0.47	36
1	0.55	0.46	0.50	26
2	0.92	0.93	0.93	10
3	1.00	0.37	0.54	67
4	0.79	0.85	0.81	26
Accuracy			0.71	258
Macro Average	0.72	0.67	0.65	258
Weighed Average	0.81	0.71	0.71	258
Class	**Precision**	**Recall**	**F1-Score**	**Support**
-1	0.30	0.35	0.32	37
1	0.62	0.96	0.76	26
2	0.93	0.88	0.90	10
3	0.98	0.84	0.90	67
4	0.65	0.50	0.57	26
Accuracy			0.76	258
Macro Average	0.70	0.71	0.69	258
Weighed Average	0.79	0.76	0.77	258
Class	**Precision**	**Recall**	**F1-Score**	**Support**
-1	0.62	0.36	0.46	36
1	1.00	0.23	0.38	26
2	0.90	0.75	0.82	10
3	1.00	0.84	0.91	67
4	0.29	1.00	0.45	26
Accuracy			0.69	257
Macro Average	0.76	0.63	0.60	257
Weighed Average	0.84	0.69	0.71	257

Table 5.3 – The obtained results using the Random Forest classifier and stratified 3-fold cross-validation.

Class	Mean	std dev.
-1	0.571	0.064
1	0.714	0.072
2	0.910	0.018
3	0.813	0.033
4	0.619	0.137
All classes	0.725	0.065

Table 5.4 – The mean F1-Score for the Random Forest classifier for each class, evaluated on 100 3-fold runs, with different seeds.

For the Linear Regression classifier, we see in Table 5.1 that the highest F1 score, precision and recall is achieved for classes 2 and 4. If one compares it with Table 5.2, we see that when doing 100 3-folds these classes still get the highest score.

Class	**Precision**	**Recall**	**F1-Score**	**Support**
-1	0.44	0.58	0.50	36
1	0.83	0.77	0.80	26
2	0.94	1.00	0.97	10
3	0.84	0.63	0.72	67
4	1.00	1.00	1.00	26
Accuracy			0.82	258
Macro Average	0.81	0.80	0.80	258
Weighed Average	0.84	0.82	0.82	258
Class	**Precision**	**Recall**	**F1-Score**	**Support**
-1	1.00	0.84	0.91	37
1	1.00	1.00	1.00	26
2	1.00	1.00	1.00	10
3	0.92	1.00	0.96	67
4	1.00	1.00	1.00	26
Accuracy			0.98	258
Macro Average	0.98	0.97	0.97	258
Weighed Average	0.98	0.98	0.98	258
Class	**Precision**	**Recall**	**F1-Score**	**Support**
-1	0.75	0.58	0.66	36
1	0.96	0.92	0.94	26
2	0.88	0.97	0.93	10
3	0.91	0.90	0.90	67
4	0.96	0.96	0.96	26
Accuracy			0.89	257
Macro Average	0.89	0.87	0.88	257
Weighed Average	0.89	0.89	0.89	257

Table 5.5 – The obtained results using XGBoost and stratified 3-fold cross-validation.

The Random Forest classifier get the highest F1-Score for classes 2 and 3, as shown in Table 5.3. For class 2 the precision was 0.93 and the recall was 0.88. Compared to Table 5.4, we see that when doing 100 3-folds classes the mean F1-score for class 3 is 0.813, a bit lower than in the stratified 3-fold. Also, note that the Random Forest model performs the worst of all the tested models.

The XGBoost classifier, represented in Table 5.5 and 5.6, get a mean F1-score above 0.966 for class 1, 2 and4. The unknown class -1 get a mean F1-Score of 0.782.

The MLP classifier, presented in Table 5.9 and 5.10, get an even performance between all but the unknown class and scores close to 0.9 for class 1, 3 and 4. We see a standard deviation of 0.071 for class 4, which is higher than for the unknown class.

Class	Mean	std dev.
-1	0.796	0.049
1	0.966	0.033
2	0.988	0.011
3	0.905	0.025
4	0.978	0.016
All classes	0.932	0.034

Table 5.6 – The mean F1-score for the XGBoost classifier for each class, evaluated on 100 3-fold, with different seeds.

Class	Precision	Recall	F1-Score	Support
-1	0.81	0.72	0.76	36
1	0.77	0.92	0.84	26
2	1.00	0.97	0.99	10
3	0.90	0.91	0.90	67
4	0.89	0.92	0.91	26
Accuracy			0.91	258
Macro Average	0.87	0.89	0.88	258
Weighed Average	0.91	0.91	0.91	258
Class	**Precision**	**Recall**	**F1-Score**	**Support**
-1	0.77	0.89	0.82	37
1	1.00	1.00	1.00	26
2	0.99	1.00	1.00	10
3	0.93	0.85	0.89	67
4	1.00	0.96	0.98	26
Accuracy			0.94	258
Macro Average	0.94	0.94	0.94	258
Weighed Average	0.95	0.94	0.94	258
Class	**Precision**	**Recall**	**F1-Score**	**Support**
-1	0.69	0.92	0.79	36
1	0.93	1.00	0.96	26
2	1.00	0.98	0.99	10
3	0.98	0.79	0.88	67
4	0.96	1.00	0.98	26
Accuracy			0.93	257
Macro Average	0.91	0.94	0.92	257
Weighed Average	0.94	0.93	0.93	257

Table 5.7 – The obtained results using SVM and stratified 3-fold cross-validation.

The LSTM model uses sequential WEs and by looking at Table 5.12 we see that it has a higher mean F1-Score for the unknown class than for class 3.

In Figure 5.1 we see a comparison of 100 3-fold cross-validations for each classifier. XGBoost has the highest mean F1-score (0.932) with a standard

Class	Mean	std dev.
-1	0.782	0.044
1	0.936	0.034
2	0.983	0.010
3	0.890	0.026
4	0.975	0.022
All classes	0.913	0.027

Table 5.8 – The mean F1-score for the SVM classifier for each class, evaluated on 100 3-fold runs, with different seeds.

Class	Precision	Recall	F1-Score	Support
-1	0.81	0.58	0.68	36
1	0.90	1.00	0.95	26
2	0.99	0.95	0.97	10
3	0.82	0.93	0.87	67
4	0.89	0.96	0.93	26
Accuracy			0.90	258
Macro Average	0.88	0.88	0.88	258
Weighed Average	0.90	0.90	0.90	258
Class	**Precision**	**Recall**	**F1-Score**	**Support**
-1	0.82	0.62	0.71	37
1	0.96	0.88	0.92	26
2	0.95	0.96	0.96	10
3	0.87	0.93	0.90	67
4	0.78	0.96	0.86	26
Accuracy			0.90	258
Macro Average	0.88	0.87	0.87	258
Weighed Average	0.90	0.90	0.89	258
Class	**Precision**	**Recall**	**F1-Score**	**Support**
-1	0.75	0.83	0.79	36
1	0.89	0.92	0.91	26
2	0.99	0.97	0.98	10
3	0.92	0.85	0.88	67
4	0.89	0.96	0.93	26
Accuracy			0.91	257
Macro Average	0.89	0.91	0.90	257
Weighed Average	0.92	0.91	0.92	257

Table 5.9 – The obtained results using MLP and stratified 3-fold cross-validation.

deviation of 0.034. The second best is the SVM with a mean F1-score of 0.913, and it have lowest standard deviation (0.027). The worst performer is random forest which performs 14.5 % worse than the linear regression model in absolute F1-score.

Class	**Mean**	**std dev.**
-1	0.738	0.064
1	0.900	0.062
2	0.968	0.015
3	0.899	0.028
4	0.922	0.071
All classes	0.885	0.058

Table 5.10 – The mean F1-score for the MLP classifier for each class, evaluated on 100 3-fold runs, with different seeds.

Class	**Precision**	**Recall**	**F1-Score**	**Support**
-1	0.67	0.5	0.57	36
1	0.96	0.92	0.94	26
2	0.85	0.61	0.71	103
3	0.52	0.82	0.64	67
11	0.96	1.00	0.98	26
Accuracy			0.98	258
Macro Average	0.98	0.97	0.97	258
Weighed Average	0.98	0.98	0.98	258
Class	**Precision**	**Recall**	**F1-Score**	**Support**
-1	1	0.84	0.91	37
1	1	1	1	26
2	1	1	1	102
3	0.92	1	0.96	67
11	1	1	1	26
Accuracy			0.98	258
Macro Average	0.98	0.97	0.97	258
Weighed Average	0.98	0.98	0.98	258
Class	**Precision**	**Recall**	**F1-Score**	**Support**
-1	0.93	0.69	0.79	36
1	0.81	1	0.90	26
2	0.96	0.98	0.97	102
3	0.95	0.94	0.95	67
11	0.93	1	0.96	26
Accuracy			0.93	257
Macro Average	0.92	0.92	0.91	257
Weighed Average	0.94	0.93	0.93	257

Table 5.11 – The obtained results using LSTM and Stratified 3-fold cross-validation.

5.2 Statistical Validation

On the F1-scores in this section we perform statistical validation using Wilcoxon signed-rank test and Friedman NxN test, described in Section 3.4.2 and 3.4.3.

Class	**Mean**	**std dev.**
-1	0.820	0.055
1	0.978	0.029
2	0.896	0.041
3	0.812	0.081
4	0.973	0.101
All classes	0.896	0.061

Table 5.12 – The mean F1-Score for the LSTM classifier for each class, evaluated on 100 3-fold runs, with different seeds.

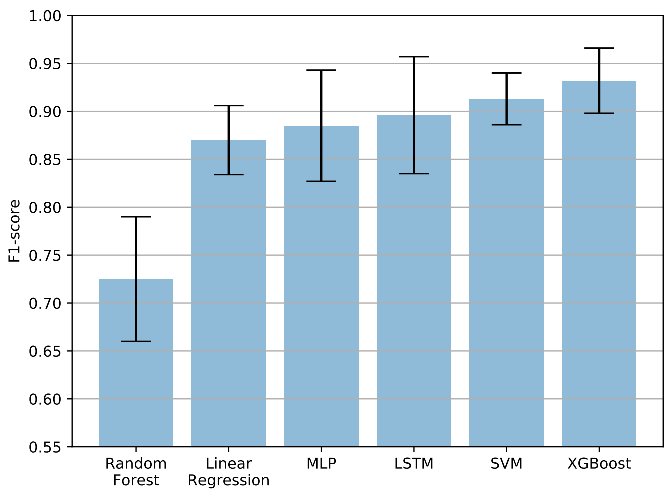


Figure 5.1 – A bar plot to compare the F1-score of the 100 3-fold cross-validations for each classifier. It contains the mean F1-score and standard deviation for each classifier.

In Table 5.13 and Table 5.15 we can see that the null hypothesis, that the mean F1-score is the same for the classifiers, can be rejected. In Table 5.14 we see the Friedman ranking, obtained by applying the Friedman procedure. We see that that the scores show the same result as the bar plot in Figure 5.1, i.e. that the XGBoost model get the best F1-score, compared to the other classifiers.

Algorithm	**Linear Regression**	**Random Forest**	**XGBoost**	**SVM**	**MLP**	**LSTM**
Linear Regression	-	7.789E-49	1.087E-35	2.149E-29	1.346E-10	1.673E-22
Random Forest	7.789E-49	-	6.159E-51	6.198E-51	6.145E-51	3.704E-02
XGBoost	1.087E-35	6.159E-51	-	1.743E-04	1.329E-03	1.520E-05
SVM	2.149E-29	6.198E-51	1.743E-04	-	5.549E-02	4.492E-08
MLP	1.346E-10	6.145E-51	1.329E-03	5.549E-02	-	1.531E-20
LSTM	1.673E-22	3.704E-02	1.520E-05	4.492E-08	1.531E-20	-

Table 5.13 – Wilcoxon test performed on the F1-scores for each 3-fold in the cross-validation.

Algorithm	Ranking
XGBoost	1.56
SVM	2.49
MLP	2.83
LSTM	3.38
Linear-Regression	4.75
Random-Forest	5.99

Table 5.14 – Average Rankings of the algorithms calculated with Friedman NxN test.

i	algorithms	$z = (R_0 - R_i)/SE$	p
15	Random-Forest vs. XGBoost	28.979336	0
14	Random-Forest vs. SVM	22.891057	0
13	Linear-Regression vs. XGBoost	20.883452	0
12	Random-Forest vs. MLP	20.665234	0
11	Random-Forest vs. LSTM	17.042817	0
10	Linear-Regression vs. SVM	14.795173	0
9	Linear-Regression vs. MLP	12.56935	0
8	XGBoost vs. LSTM	11.936519	0
7	Linear-Regression vs. LSTM	8.946933	0
6	XGBoost vs. MLP	8.314102	0
5	Linear-Regression vs. Random-Forest	8.095884	0
4	XGBoost vs. SVM	6.088279	0
3	SVM vs. LSTM	5.848239	0
2	MLP vs. LSTM	3.622417	2.29E-5
1	SVM vs. MLP	2.225822	2.60E-3

Table 5.15 – P-values Table for $\alpha = 0.05$.

Chapter 6

Threats to Validity

The threats to the validity, limitations, and the challenges faced in conducting the present study are discussed in this chapter. In the list below, different types of threats to validity are briefly described and discussed in relation to this study.

- **Construct validity** reflects on if the results executed in the study represents what the researcher is investigating, and if it represents what the research questions define [73]. The major construct validity threat in the present study is the way that the proper troubleshooting activities are mapped for each class. Utilizing the SMEs knowledge may not be attainable in other testing processes. Moreover, communication between different SMEs in the organization, in order to capture and share information, may require more time and effort than the amount of time saved by the proposed solution.
- **Internal validity** reflects on if the relationships found in a study is an actual relationship, or if it is produced by an unknown factor not known or considered by the researcher [73]. One threat to this is that the study is only performed on a dataset only available to the host company. The test case log files analyzed contain log events that are created by employees. Other datasets will probably not have the same structure. However, the method can still be applied since almost all logs contain the same type of log event priorities (i.e. ASSERT, FAIL, ERROR, DEBUG, INFO, etc.). The pipeline is testable on another dataset, but the results cannot be directly compared to this study if any changes are made.
- **External validity** reflects on the possibility of generalizing the find-

ings and how much the study can interest people outside the investigated case [73]. Since we treat the groups of log events as pre-processed text, this model could potentially be used on any log files. An optional step in the process is to select the log events with certain keywords to reduce the amount of data that needs to be processed. This may or may not be possible in other scenarios, depending on what information is extractable from the dataset.

- **Reliability** reflects on to what extent the data and analysis depend on the researchers, and if it is repeatable by another researcher [73]. One threat is that the data is initially labeled by SME at Ericsson. These labels could be influencing the quality of the data, the number of classes, etc. By executing a similar study on another dataset, created by other people, could end up being very different. The ground truth do affect the results. By examining the dataset, we see that each class resembles other data points in the same class. With the data we use, we found that the classifiers are quite stable with different hyperparameters, i.e., that the data speaks for itself. However, the accuracy of this statement needs to be evaluated while testing more datasets.

Chapter 7

Discussion and Future Work

This chapter discusses the results presented in Chapter 5. We discuss the evaluation of different classifiers and the proposed feature engineering method that involves merging WEs for each log event. We then continue with future work where improvements to the method are given. We end with a small discussion related to ethics and sustainability.

7.1 Discussion

In this study, two ways to do feature engineering is evaluated. Both translate words to WEs using FastText CBOW and Skip-gram models. All the models are feed with 6 log events per group. After transforming the words in each log event into WEs, dimension reduction is executed by using UMAP. The LSTM model, that take sequential input, is given the reduced WEs in a sequential manner. The other models are feed with the reduced merged WEs that are weighted with TF-IDF and summed into a merged WE. The best performance was achieved with XGBoost, with a mean F1-score of 0.932 and a standard deviation of 0.034 when evaluating 100 3-fold cross-validations with different seeds. The LSTM classifier got a mean F1-score of 0.896 and a standard deviation of 0.061 when evaluating using the same type of cross-validation and is the best performer on the unknown class(-1). The SVC classifier had the smallest standard deviation in F1-score (0.027) and a mean F1-score of 0.913 with different seeds and k-folds. Hence, it is more stable than the other classifiers with respect to those two parameters.

For the classifiers with merged WEs, the F1-score and its standard-deviation are similar for different data splits and seeds. However, the random forest classifier is an exception and performs much worse. This suggests that the merged WEs do capture the meaning or type of error for each group of log events, and all classifiers but random forest classify it well. The results from the random forest classifier suggest that in our setup, the classifier is prone to overfitting.

A conclusion we can make is that the suggested method for merging WEs is a viable method to reduce the dimensionality of each log event. A second one is that using the FastText CBOW and skip-gram models, that evaluate character n-grams, is a viable strategy when dealing with logs that contain merged words, variables, JSON responses, etc.

7.2 Future Work

The results of this study show the potential to form input features based on what we have described in Section 4.3.1. It opens a possibility for launch automated troubleshooting activities to solve the problem automatically when they have been classified. The focus in this study has been to aid the SMEs in the process of doing log analysis, which is a very time-consuming process so they can focus and spend their time on more important tasks. There exist many ways to extend this study, and a subsample of those is:

- *Extend the analysis on other datasets:* Currently, the analysis is done on one dataset for one of the internal products at Ericsson. At Ericsson, the study could be extended to systems that use the same type of logging tools and store them in the same format. It can also be tested on other datasets, that do not have many similarities with the dataset in this study. However, datasets consisting of logs with multiple different classes are rare. Many papers up until now mostly consist of anomaly detection, where the classification is a binary case [6].
- *Using newer types of deep learning models that can take sequential input.* The LSTM model is by today's standards in ML a quite old algorithm and many other types of architectures have shown promising results, such as distilBERT. Initial tests have been executed with distilBERT [43], but due to resource constraints, there was not enough time to evaluate the model.

- *Include more classes as data points are gathered.* Currently, more data is gathered as it is generated to increase the number of data points per class. As soon as there are enough data points for the minority classes, the classifier can be trained to classify these as well. For now, we settled with not including classes that do not have more than 10 data points, since those classes did not have enough data to represent the classes.
- *Use anomaly detection as a filter instead of key identifiers.* A lot of research has been done on anomaly detection, i.e. binary classification, where log events with an anomaly are highlighted. This could be a way to reduce the amount of logs events instead of using key identifiers, but it also requires access to labels on the log event level, which we in this case did not have, and it would be too much work for the SMEs to give. The steps given in this report would be executed on the log events highlighted using anomaly detection algorithms.
- *On other datasets, try same setup but preprocess using log parsers mentioned in Section 2.2.* The dataset in this case study was structured as free text rather than log events created from log event templates. If the log events were more structured, the input could have been filtered using log templates extraction techniques.
- *Train WE models on the whole logs.* Training the WE model on the whole log files would potentially create a better word representation of the words but would also require much more time for training.
- *Test the method on another dataset and use existing log parsers.* In this project, we did initially try to find log event templates for the whole test case log files. However, many of the templates did not make sense or were useful due to that the logs we analyzed do contain a lot of free text. This means that the log events are less structured compared to, for example, server logs. If a log with more specific patterns is analyzed, log parsers can be used to find these templates and thereby filter the log events on a higher level.

7.2.1 Ethics and Sustainability

In this section, the ethical impacts of this work, as well as its effect on sustainability, will be discussed. The impact on society needs to be analyzed, as every work can have a positive and negative impact that needs to be analyzed.

For example, NLP models such as OpenAI's GPT-3 could potentially be used to generate text that in turn is used by bots.

However, the work in this case study is not of nature since it's an analysis of test cases. Most troubleshooting of failing test cases is done manually at the department where this study was executed. Having a CI pipeline to daily evaluate the integrity of the software and hardware under development is very important. Evaluating failing test cases to find faults is a very time-consuming process, and it takes time from other tasks the developer has on their schedule. In this work, we have explored ways to process failing test case logs and classify their fault to simplify the log analysis. The work in this thesis aims at helping developers troubleshooting and giving them advice for solving a technical error. The potential with implementing the models in this study is that give developers more time to other tasks, i.e. they become more productive and saves the company resources. This type of automation presented in this paper does have a positive impact on the economy since it is a task that does not have its own profession and it helps the developers to be more productive. Also, the models presented in this thesis only analyze text generated from automated test case executions, stored in log files. This means there is no personal information stored there that could be incorrectly handled.

Chapter 8

Conclusions

In this study, we have addressed the problem of log analysis, where logs from failing test cases need to be examined to find errors and find ways to solve them. We presented a way to filter and pre-process logs and perform feature engineering to create WEs that are used to classify different types of errors. The output from the classifier was used to suggest troubleshooting activities. Since the model uses a supervised approach, SMEs were required to label groups of log events with the correct error type or troubleshooting activity. With the help of SMEs, the data used in this study was divided into 16 classes. We decided to solve the problem for a subset of those classes and relabel the rest of the classes to a label that represents an unknown class. This was done to get a more realistic performance. The proposed approach was conducted and evaluated in an industrial case study at Ericsson AB in Sweden.

Key identifiers, given to us by the SMEs, were used to reduce the size of the logs. With these, we were able to capture the important part that was needed to be able to classify the errors. A way to get around this in the future could be to use anomaly detection to filter out log events of interest. That would require an online learning system where SMEs can label the data in interesting/not interesting as well, requiring 2 labels for each log event.

In our presented approach, we train CBOW and Skip-Gram FastText models. They have the advantage of being able to transform character n-grams instead of words into WEs. This helps with the problem that logs often contain many uncommon words and includes, for example, variable names, JSON-responses, stack traces, etc. When the words in each log events are transformed into WEs, these embeddings are merged. After the merge, we have merged

WEs for CBOW and skip-gram, respectively. Each group of log events is then fed into different classifiers. Models that takes sequential input does not need the merging step since they operates directly on a sequence of WEs.

The best classifier overall was XGBoost with a mean F1-score of 0.932 and a standard deviation of 0.034 when evaluating 100 3-fold cross-validations with different seeds. For the sequential classifier LSTM, we got a mean F1-Score of 0.896 and a standard deviation of 0.061 when evaluating using the same type of cross-validation. Our empirical evaluation of the two different types of feature models, we see that the XGBoost model performs better than the LSTM model.

A conclusion we can make from this study is that to classify a group of log events, the data labeled needs to represent the true distribution. For example, if many different log events represents the same type of error, and they occur in different test case log files, enough data to represent the class needs to be gathered. This could be a trouble for rare errors but note that all these classes can be included when there is enough data, and data is generated all the time by the system for CI. In this report, we trained all classifiers with the unknown class to get a more realistic performance. Also, it is more useful to the developers since it tells them when the classifier does not know the correct class.

The approach in this study showed that the methodology used here got good potential to simplify troubleshooting in test case logs. Using character n-grams is very useful and merging log events is a successful way of making dimension reduction in logs. The merged WEs, together with the XGBoost classifier, is much faster to train compared to the LSTM model and is much easier to reproduce.

Bibliography

[1] S. Tahvili, M. Bohlin, M. Saadatmand, S. Larsson, W. Afzal, and D. Sundmark, “Cost-benefit analysis of using dependency knowledge at integration testing,” in *The 17th International Conference On Product-Focused Software Process Improvement*, 2016.

[2] S. Tahvili, “Multi-criteria optimization of system integration testing,” Ph.D. dissertation, Mälardalen University, 2018.

[3] S. Tahvili, M. Saadatmand, S. Larsson, W. Afzal, M. Bohlin, and D. Sundmark, “Dynamic integration test selection based on test case dependencies,” in *The 11th Workshop on Testing: Academia-Industry Collaboration, Practice and Research Techniques*, 2016.

[4] S. Tahvili, R. Pimentel, W. Afzal, M. Ahlberg, E. Fornander, and M. Bohlin, “sOrTES: A supportive tool for stochastic scheduling of manual integration test cases,” *IEEE Access*, vol. 6, pp. 1–19, 2019.

[5] M. Fowler. (2006, May) Continuous Integration. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html

[6] J. Candido, M. Aniche, and A. van Deursen, “Contemporary Software Monitoring: A Systematic Literature Review,” *arXiv:1912.05878 [cs]*, Dec. 2019, arXiv: 1912.05878. [Online]. Available: http://arxiv.org/abs/1912.05878

[7] M. Du, F. Li, G. Zheng, and V. Srikumar, “DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 1285–1298. [Online]. Available: https://doi.org/10.1145/3133956.3134015

[8] W. Li, "Automatic Log Analysis using Machine Learning : Awesome Automatic Log Analysis version 2.0," Master's thesis, Uppsala University, Department of Information Technology, 2013.

[9] S. Tahvili, W. Afzal, M. Saadatmand, M. Bohlin, D. Sundmark, and S. Larsson, "Towards earlier fault detection by value-driven prioritization of test cases using fuzzy topsis," in *13th International Conference on Information Technology : New Generations (ITNG 2016)*, 2016.

[10] S. Tahvili, M. Ahlberg, E. Fornander, W. Afzal, M. Saadatmand, M. Bohlin, and M. Sarabi, "Functional dependency detection for integration test cases," in *The 18th IEEE International Conference on Software Quality, Reliability and Security*, 2018.

[11] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "CloudSeer: Workflow Monitoring of Cloud Infrastructures via Interleaved Logs," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 489–502, Mar. 2016. [Online]. Available: https://doi.org/10.1145/2980024.2872407

[12] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou, "LogAnomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, Aug. 2019, pp. 4739–4745. [Online]. Available: https://www.ijcai.org/proceedings/2019/658

[13] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," *arXiv:1301.3781 [cs]*, Sep. 2013, arXiv: 1301.3781. [Online]. Available: http://arxiv.org/abs/1301.3781

[14] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality," *arXiv:1310.4546 [cs, stat]*, Oct. 2013, arXiv: 1310.4546. [Online]. Available: http://arxiv.org/abs/1310.4546

[15] A. Oprea, Z. Li, T.-F. Yen, S. Chin, and S. Alrwais, "Detection of Early-Stage Enterprise Infection by Mining Large-Scale Log Data," *arXiv:1411.5005 [cs]*, Nov. 2014, arXiv: 1411.5005. [Online]. Available: http://arxiv.org/abs/1411.5005

[16] D. Gonçalves, J. Bota, and M. Correia, “Big Data Analytics for Detecting Host Misbehavior in Large Logs,” in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, Aug. 2015, pp. 238–245.

[17] E. Chuah, A. Jhumka, S. Narasimhamurthy, J. Hammond, J. C. Browne, and B. Barth, “Linking Resource Usage Anomalies with System Failures from Cluster Log Data,” in *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, Sep. 2013, pp. 111–120, iSSN: 1060-9857.

[18] N. Gurumdimma, A. Jhumka, M. Liakata, E. Chuah, and J. Browne, “CRUDE: Combining Resource Usage Data and Error Logs for Accurate Error Detection in Large-Scale Distributed Systems,” in *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, Sep. 2016, pp. 51–60, iSSN: 1060-9857.

[19] A. Pi, W. Chen, X. Zhou, and M. Ji, “Profiling distributed systems in lightweight virtualized environments with logs and resource metrics,” in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC ’18. New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 168–179. [Online]. Available: https://doi.org/10.1145/3208040.3208044

[20] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, “Log clustering based problem identification for online service systems,” in *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE ’16*. Austin, Texas: ACM Press, 2016, pp. 102–111. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2889160.2889232

[21] B. Chen, J. Song, P. Xu, X. Hu, and Z. M. J. Jiang, “An automated approach to estimating code coverage measures via execution logs,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, Sep. 2018, pp. 305–316. [Online]. Available: https://doi.org/10.1145/3238147.3238214

[22] J.-G. Lou, Q. Fu, S. YANG, Y. XU, and J. Li, “Mining Invariants from Console Logs for System Problem Detection,” in *Annual Technical Conference (full paper)*. USENIX, Jun. 2010. [Online]. Available: https://www.microsoft.com/en-us/research/publication/mining-invariants-from-console-logs-for-system-problem-detection/

[23] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, “An Evaluation Study on Log Parsing and Its Use in Log Mining,” in *2016 46th Annual IEEE/IFIP*

International Conference on Dependable Systems and Networks (DSN), Jun. 2016, pp. 654–661, iSSN: 2158-3927.

[24] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "A lightweight algorithm for message type extraction in system application logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 11, pp. 1921–1936, 2012.

[25] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and Benchmarks for Automated Log Parsing," *arXiv:1811.03509 [cs]*, Jan. 2019, arXiv: 1811.03509. [Online]. Available: http://arxiv.org/abs/1811.03509

[26] D. Lindqvist, "Detection of Infrastructure Anomalies in Build Logs Using Machine Learning," Master's thesis, Umeå University, Department of Computing Science, 2019. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-164730

[27] C. Landin, S. Tahvili, H. Haggren, M. Längkvist, A. Muhammad, and A. Loutfi, "Cluster-based parallel testing using semantic analysis," in *The Second IEEE International Conference On Artificial Intelligence Testing*, 2020.

[28] C. Landin, L. Hatvani, S. Tahvili, H. Haggren, M. Längkvist, A. Loutfi, and A. Håkansson, "Performance comparison of two deep learning algorithms in detecting similarities between manual integration test cases," in *The Fifteenth International Conference on Software Engineering Advances*, 2020.

[29] R. Ren, J. Cheng, Y. Yin, J. Zhan, L. Wang, J. Li, and C. Luo, "Deep Convolutional Neural Networks for Log Event Classification on Distributed Cluster Systems," in *2018 IEEE International Conference on Big Data (Big Data)*. Seattle, WA, USA: IEEE, Dec. 2018, pp. 1639–1646. [Online]. Available: https://ieeexplore.ieee.org/document/8622611/

[30] J. Kahles, J. Törrönen, T. Huuhtanen, and A. Jung, "Automating Root Cause Analysis via Machine Learning in Agile Software Testing Environments," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, Apr. 2019, pp. 379–390, iSSN: 2159-4848.

[31] E. Chuah, A. Jhumka, S. Narasimhamurthy, J. Hammond, J. C. Browne, and B. Barth, "Linking Resource Usage Anomalies with System Failures

from Cluster Log Data," in *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, Sep. 2013, pp. 111–120, iSSN: 1060-9857.

[32] T. Kimura, K. Ishibashi, T. Mori, H. Sawada, T. Toyono, K. Nishimatsu, A. Watanabe, A. Shimoda, and K. Shiomoto, "Spatio-temporal factorization of log data for understanding network events," in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, Apr. 2014, pp. 610–618, iSSN: 0743-166X.

[33] C. Soto-Valero, J. Bourcier, and B. Baudry, "Detection and analysis of behavioral T-patterns in debugging activities," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR 18. New York, NY, USA: Association for Computing Machinery, May 2018, pp. 110–113. [Online]. Available: https://doi.org/10.1145/3196398.3196452

[34] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching Word Vectors with Subword Information," *arXiv:1607.04606 [cs]*, Jun. 2017, arXiv: 1607.04606. [Online]. Available: http://arxiv.org/abs/1607.04606

[35] C. Bertero, M. Roy, C. Sauvanaud, and G. Tredan, "Experience Report: Log Mining Using Natural Language Processing and Application to Anomaly Detection," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2017, pp. 351–360, iSSN: 2332-6549.

[36] S. Tahvili, L. Hatvani, E. Ramentol, R. Pimentel, W. Afzal, and F. Herrera, "A novel methodology to classify test cases using natural language processing and imbalanced learning," *Engineering Applications of Artificial Intelligence*, vol. 95, pp. 1–13, 2020.

[37] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI Blog*, vol. 1, no. 8, p. 9, 2019.

[38] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-

Shot Learners," *arXiv:2005.14165 [cs]*, Jul. 2020, arXiv: 2005.14165. [Online]. Available: http://arxiv.org/abs/2005.14165

[39] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv:1810.04805 [cs]*, May 2019, arXiv: 1810.04805. [Online]. Available: http://arxiv.org/abs/1810.04805

[40] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "XLNet: Generalized Autoregressive Pretraining for Language Understanding," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d. Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 5753–5763. [Online]. Available: http://papers.nips.cc/paper/8812-xlnet-generalized-autoregressive-pretraining-for-language-understanding.pdf

[41] J. Howard and S. Ruder, "Universal Language Model Fine-tuning for Text Classification," *arXiv:1801.06146 [cs, stat]*, May 2018, arXiv: 1801.06146. [Online]. Available: http://arxiv.org/abs/1801.06146

[42] J. M. Eisenschlos, S. Ruder, P. Czapla, M. Kardas, S. Gugger, and J. Howard, "MultiFiT: Efficient Multi-lingual Language Model Fine-tuning," *arXiv:1909.04761 [cs]*, Jun. 2020, arXiv: 1909.04761. [Online]. Available: http://arxiv.org/abs/1909.04761

[43] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter," *arXiv:1910.01108 [cs]*, Feb. 2020, arXiv: 1910.01108. [Online]. Available: http://arxiv.org/abs/1910.01108

[44] K. Kc and X. Gu, "ELT: Efficient Log-based Troubleshooting System for Cloud Computing Infrastructures," in *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, Oct. 2011, pp. 11–20, iSSN: 1060-9857.

[45] W. Jiang, C. Hu, S. Pasupathy, A. Kanevsky, Z. Li, and Y. Zhou, "Understanding customer problem troubleshooting from storage system logs," in *Proccedings of the 7th conference on File and storage technologies*, ser. FAST '09. USA: USENIX Association, Feb. 2009, pp. 43–56.

[46] H. Mochizuki and M. Mochizuki, "Troubleshooting support device, troubleshooting support method and storage medium having program

stored therein," US Patent US7 849 363B2, Dec., 2010. [Online]. Available: https://patents.google.com/patent/US7849363B2/en

[47] D. M. Winnick, "Systems and methods for automated troubleshooting," US Patent US9 984 329B2, May, 2018. [Online]. Available: https://patents.google.com/patent/US9984329B2/en

[48] B. Debnath and H. Zhang, "Field content based pattern generation for heterogeneous logs," US Patent US10 678 669B2, Jun., 2020. [Online]. Available: https://patents.google.com/patent/US10678669B2/en

[49] N. Jain and R. Potharaju, "Problem inference from support tickets," US Patent US9 229 800B2, Jan., 2016. [Online]. Available: https://patents.google.com/patent/US9229800B2/en

[50] S. Purushothaman and A. MISHRA, "Data analysis and support engine," US Patent US20 180 285 750A1, Oct., 2018. [Online]. Available: https://patents.google.com/patent/US20180285750A1/en

[51] K. V. Jadunandan, S. A. Lobo, R. D. Lumpkins, B. D. Lushear, and P. A. S. Jr, "Communication network operations management system and method," US Patent US9 753 800B1, Sep., 2017. [Online]. Available: https://patents.google.com/patent/US9753800B1/en

[52] F. Vidal, C. Bromann, B. S. Adelberg, R. Henrikson, and J. Sandberg, "Analytics for an automated application testing platform," US Patent US20 190 340 512A1, Nov., 2019. [Online]. Available: https://patents.google.com/patent/US20190340512A1/en

[53] S. Cai, L. Zhang, A. Palazoglu, and J. Hu, "Clustering analysis of process alarms using word embedding," *Journal of Process Control*, vol. 83, pp. 11–19, Nov. 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0959152418303366

[54] Y. Li, Q. Pan, T. Yang, S. Wang, J. Tang, and E. Cambria, "Learning Word Representations for Sentiment Analysis," *Cognitive Computation*, vol. 9, no. 6, pp. 843–851, Dec. 2017. [Online]. Available: https://doi.org/10.1007/s12559-017-9492-2

[55] C. M. Rosenberg and L. Moonen, "Improving Problem Identification via Automated Log Clustering using Dimensionality Reduction," *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 1–10, Oct. 2018, arXiv: 2009.03257. [Online]. Available: http://arxiv.org/abs/2009.03257

[56] L. McInnes, J. Healy, and J. Melville, "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction," *arXiv:1802.03426 [cs, stat]*, Sep. 2020, arXiv: 1802.03426. [Online]. Available: http://arxiv.org/abs/1802.03426

[57] S. Tahvili, W. Afzal, M. Saadatmand, M. Bohlin, and S. H. Ameerjan, "Espret: A tool for execution time estimation of manual test cases," *Journal of Systems and Software*, vol. 161, pp. 1–43, 2018.

[58] S. Tahvili, L. Hatvani, M. Felderer, W. Afzal, and M. Bohlin, "Automated functional dependency detection between test cases using doc2vec and clustering," in *The First IEEE International Conference On Artificial Intelligence Testing*, 2019.

[59] S. Tahvili, L. Hatvani, M. Felderer, W. Afzal, M. Saadatmand, and M. Bohlin, "Cluster-based test scheduling strategies using semantic relationships between test specifications," in *5th International Workshop on Requirements Engineering and Testing*, 2018.

[60] S. Tahvili, M. Saadatmand, M. Bohlin, W. Afzal, and S. H. Ameerjan, "Towards execution time prediction for test cases from test specification," in *43rd Euromicro Conference on Software Engineering and Advanced Applications*, 2017.

[61] Tf-idf :: A single-page tutorial - information retrieval and text mining. [Online]. Available: http://www.tfidf.com/

[62] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, Aug. 2016, pp. 785–794. [Online]. Available: https://doi.org/10.1145/2939672.2939785

[63] C. D. Manning, H. Schütze, and P. Raghavan, "Introduction to information retrieval," 2008, iSBN: 9780521865715 9780511414053 Publisher: Cambridge University Press. [Online]. Available: https://cds.cern.ch/record/2135372

[64] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, publisher: MIT Press. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[65] C. Olah. (2015, Aug.) Understanding LSTM networks – colah's blog. Online. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[66] F. Wilcoxon, "Individual Comparisons by Ranking Methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945, publisher: [International Biometric Society, Wiley]. [Online]. Available: https://www.jstor.org/stable/3001968

[67] M. Friedman, "A Comparison of Alternative Tests of Significance for the Problem of m Rankings," *Annals of Mathematical Statistics*, vol. 11, no. 1, pp. 86–92, Mar. 1940, publisher: Institute of Mathematical Statistics. [Online]. Available: https://projecteuclid.org/euclid.aoms/1177731944

[68] R. Eisinga, T. Heskes, B. Pelzer, and M. T. Grotenhuis, "Exact p-values for pairwise comparison of Friedman rank sums, with application to comparing classifiers," *BMC Bioinformatics*, vol. 18, no. 1, 2017. [Online]. Available: https://link.springer.com/epdf/10.1186/s12859-017-1486-2

[69] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[70] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[71] J. Alcalá-Fdez, A. Fernández, J. Luengo, J. Derrac, S. García, L. Sánchez, and F. Herrera, "Keel data-mining software tool: data set repository, integration of algorithms and experimental analysis framework." *Journal of Multiple-Valued Logic & Soft Computing*, vol. 17, 2011.

[72] J. Alcalá-Fdez, L. Sánchez, S. García, M. J. del Jesus, S. Ventura, J. M. Garrell, J. Otero, C. Romero, J. Bacardit, V. M. Rivas, J. C. Fernández, and F. Herrera, "KEEL: a software tool to assess evolutionary algorithms for data mining problems," *Soft Computing*, vol. 13, no. 3, pp. 307–318, Feb. 2009. [Online]. Available: https://doi.org/10.1007/s00500-008-0323-y

[73] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empirical Software Engineering*, vol. 14, no. 2, p. 131, Dec. 2008. [Online]. Available: https://doi.org/10.1007/s10664-008-9102-8

[74] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *arXiv:1412.6980 [cs]*, Jan. 2017, arXiv: 1412.6980. [Online]. Available: http://arxiv.org/abs/1412.6980

Appendix A

Hyperparameters

A.1 LSTM

Skip-gram	**word_ngrams**	**negative**	**size**	**hs**	**sg**
	3	5	20	1	1
CBOW	**word_ngrams**	**negative**	**size**	**hs**	**sg**
	3	5	20	1	0
Exponential Delay	**initial_lr**	**decay_steps**	**decay_rate**		
	0.1	100	0.96		
Layers	**type**	**output shape**	**parameters**		
	bidirectional LSTM	(, 64)	18688		
	dropout	(, 64)	0		
	dense	(, 5)	325		
	softmax	(, 5)	0		

Table A.1 – The hyperparameters related to the LSTM classifier.

A.2 Other Classifiers

Max number of log events per type per input	assert	fail	error	info	debug
	1	1	1	2	2
Skip-gram	**word_ngrams**	**negative**	**size**	**hs**	**sg**
	3	5r	20	1	1
UMAP for Skip-gram	**n_neighbors**	**n_components**	**init**		
	20	3	'random'		
CBOW	**word_ngrams**	**negative**	**size**	**hs**	**sg**
	3	5r	20	1	0
UMAP for CBOW	**n_neighbors**	**n_components**	**init**		
	20	3	'random'		
Logistic Regression	**solver**	**penalty**	**max_iter**	**class_weight**	
	lbfgs	l2	100	'balanced'	
Random Forest	**max_depth**	**max_features**	**n_estimators**	**class_weight**	
	3	0.3	100	'balanced'	
XGBoost	**max_depth**	**colsample_bytree**	**n_estimators**	**sample_weight**	
	2	0.5	100	'balanced'	
MLP	**hidden_layer_size**	**solver**	**alpha**		
	(100,100)	lbfgs	0.001		
Support Vector Machine	**kernel**	**degree**	**decision_function_shape**	**class_weight**	
	'rbf'	_	'ovo'	'balanced'	

Table A.2 – The hyperparameters related to all classifiers except the LSTM.

TRITA-EECS-EX-2020:891