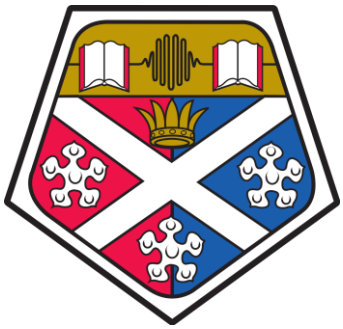

PREDICTING THE RESOLUTION TIME AND PRIORITY OF BUG REPORTS: A DEEP LEARNING APPROACH

MIHAIL MIHAYLOV

This dissertation was submitted in part fulfilment of requirements for the degree of MSc Advanced Computer Science

DEPT. OF COMPUTER AND INFORMATION SCIENCES
UNIVERSITY OF STRATHCLYDE

AUGUST 2019

DECLARATION

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself. Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive.

(please tick) Yes [x] No [ ]

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices) is: 17,611

I confirm that I wish this to be assessed as a Type 1 2 3 4 (5)

Dissertation (please circle)

Signature:

Date:

ABSTRACT

Debugging is a crucial component of the software development lifecycle. Most of the debugging done, once a system has been released, is based on bug reports. However, working on those bug reports could take developers a lot of time and resources and there are not a lot of available tools to help with that, which identifies a need to develop such tools and guidelines to analyze the information contained in bug reports.

This paper introduces the usage of neural networks when performing bug report analysis. With the help of ensemble configurations of Multi-Layer Perceptron (MLP), Convolutional Neural Network (CNN) and Long Short-Term Memory (LSTM) and using sentiment and textual analysis, an experiment has been developed to predict the resolution time of bug reports and classify the priority. Additionally, the experiment analyses the importance of both textual and numerical factors when analysing bug reports. Based on the models developed, it is shown that using both textual and numerical data improves the results of standard text only neural networks. The obtained results show that determining the bug-fix time based on the information supplied in bug reports is achievable and also show success when trying to classify the priority of a bug report.

Further evaluation and optimization of the best models, leads to the discovery of inequality in the distribution of the labels of the dataset, which leads to not very well trained models. With the use of another experiment an attempt has been made to improve the results by reducing the dataset, to contain similar counts of labels, which proves unsuccessful.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor Dr Marc Roper for guiding me through this project and providing me with the necessary support and insight to complete it.

I would also like to thank my parents who have been there for me and helped me throughout my MSc Advanced Computer Science degree.

Table of Contents

Table of Figures

List of Tables

Chapter 1: Introduction

The definition of a bug report is: a software artefact used to describe software bugs, particularly in software systems that are open-sourced (Goyal and Sardana, 2017). Fixing bugs in a code is a significant part of the software development lifecycle (SDLC). From the beginning of a software project a great amount of the work involves fixing bugs in the source code. Around 45% of all expenses of a software project is used on maintenance activities (Zhang et al., 2015). Maintaining open bug repositories helps developers to collect, organize and resolve bug reports. However, for a software bug to be fixed in a timely manner, it is necessary that the bug report is a source of sufficient information to the developer.

In order for a software bug to be resolved quickly and easily, there are many guidelines and tools to assist developers in debugging. Guidelines are used by reporters when submitting bug reports to ensure that the reporter provides sufficient information necessary for the developer to be able to reproduce the bug. There have also been tools developed that use the information in the bug report in order to try and triage and localize the bug as well as detect duplicates, estimate the bug-fix time and predict the priority. Most of those tools use machine learning in order to automate those processes and alleviate the workload of developers.

This project would mainly focus on exploring bug report features supplied by reporters and use them to predict the fix-time and the priority of the bug. Developing a good model which could predict those would help developers better manage their time and increase their productivity as well as improve the guidelines on fixing bug reports giving the reporters an estimate on the quality of their reports.

The next chapter of this report would focus on exploring the available tools and guidelines and provide background information on different kinds of bug report analysis, textual and sentiment analysis as well as machine learning and neural networks. Chapter 3 is the research chapter, which would focus on the methodology, defining the hypotheses and designing the experiment, as well as the implementation details and Chapter 4 will present the results from the experiment and analyse those results, attempting to validate

the hypotheses. The final chapters will then go on to describe future work and conclude this paper respectively.

Chapter 2: Background and Literature Review

This section will provide the reader with the necessary background regarding this project in order to ensure full understanding of the methodology, implementation and analysis sections of this report. It will begin with explaining what bugs and bug reports are, the dataset that is being used as well as background research into types of bug report analysis, machine learning and neural networks. It would also inform about similar works in the field of bug report analysis.

2.1 Bugs and Bug Reports

A software bug is a problem which causes the software to crash or work incorrectly. A bug can be an error, mistake or defect of fault, which may cause the software to act in unexpected ways and may cause failure or deviation from expected results (Techopedia.com, 2019).

Bug reports are used to inform developers of software bugs in a system by users of the software. Bug reports contain the information necessary to reproduce and fix problems, such as version of the software, component, steps to reproduce, stack traces, code, description, operating system and other relevant information.

Over the last few years a considerable amount of research has been carried out on bug report analysis. A survey on bug reports conducted by Bettenburg et al. focuses on asking developers about the most important features of bug reports. The survey was responded to by 156 developers from Apache, Eclipse and Mozilla. They state that the most widely used bug report features are steps to reproduce, observed and expected behaviour, stack traces and test cases, with the most important one being steps to reproduce. Based on the survey conducted, the researchers have developed a tool called Cuezilla, which can be used to measure the quality of bug reports based on the contents. The tool uses linear regression and the data collected from the developers' survey responses and is evaluated on a dataset of bug reports which were evaluated, by hand,

from developers. The highest testing accuracy achieved by the Cuezilla tool is 87% (Zimmermann et al, 2008). The same survey was performed with just the eclipse platform and using the same tool which in this case the developers evaluation differed within one interval for 90% of the bugs (Bettenburg et al., 2007). Another research involving statistical analysis on over 27,000 publicly available bug reports in the Mozilla Firefox project was performed by Hooimeijer et al. They try to estimate the time it takes to triage a bug report using a basic linear regression model and try to categorize bug reports depending on whether it would be cheap or expensive to triage. Their model performed better than random in terms of precision and recall (Hooimeijer and Weimer, 2007). These research papers all show the necessity for better tools to help both developers and bug reporters in dealing with bug reports.

When it comes to bug reports there are different types of analysis that can be carried out on them. Such analysis involve bug triaging, bug localization, bug-fix time prediction, duplicate bug identification and priority prediction. The next few subsections would include information about the different types of bug report analysis as well as already available tools and research papers.

2.1.1 Dataset

The dataset that this project has been conducted with has been obtained from the Github repository of the LogPai Team (logpai, 2019). It consists of 85,156 bug reports. The reports available in the dataset come from the Eclipse bug repository. One of the reasons this dataset has been chosen for this experiment is because of the size of the dataset. A dataset of 85,156 instances with textual and numeric values is neither too small to achieve accurate models on it, nor too big to make the training time too long and require a lot of computational power. Another reason is that the bug reports are all collected in a file with comma separated values (extension .csv), which is a common file type for datasets and is suitable to work with in an anaconda environment with tensorflow, pandas and the other necessary packages. An example of an instance (bug report), as is in the dataset, can be seen in Table 1 below.

Issue ID: 105008	Priority: P3	Component: UI	Duplicate: N/A
Title:	TVT3.1: Eclipse doesnt honor flipped icons		
Description:	OS : Windows XP ; Must fix ; Build date:22/07 ; Blocking: NO ; Language: ARA ; Bitmap Location: Z:\defects\TPTP_Icon1 ; Tester Name: Mohamed Esmat ; ; Problem Description: ; The Icons in the figure attached is already identified to be flipped ; but ; they are not flipped at the build. I need them to be flipped.; - Arabic tester-; ; The Fragment will be sent to development in a separate note.		

Status: Verified	Resolution: Fixed	Version: 3.1	Created Time: 25/07/2005 09:45:00	Resolved Time: 26/09/2005 15:30:58

Table 1. Sample bug report from the Eclipse dataset.

The third reason to choose this dataset is because of the features available in it. The created time and resolved time values can be used to calculate the bug-fix time, which is going to be used as a label. It also contains the resolution of the bug report, which is really helpful because the only bug reports of interest are the ones with resolution values “Fixed”. Another feature available which can be used as a label is the priority column. In this case it is in a format which can easily be transformed to numerical values. Title and description are also available for each bug report as well as the version of the software that the reporter is using and the component, which points to the part of the software that the problem occurred at. Those features will be processed using the suitable tools and used to train neural network models on them.

2.1.2 Bug Triage

Bug triage is a process of screening and prioritising bug reports and feature requests. Triage is the first step after a bug report has been submitted but it is possible for this process to be repeated in certain circumstances, such as if the assumptions about the bug report were wrong, the issue was resolved in a different way and others.

The process of bug triaging involves initial screening, confirming the issue, following up on the issue as well as revisiting older reports. The initial screening part of the process involves answering a set of questions in order to determine if the report is a genuine report which should be further investigated or it should be closed. The next part,

confirming the issue, starts by investigating the report. It needs to be verified that the bug reported does not present a security issue. Such issues get a higher security level in order for them to be resolved as soon as possible. Another step is determining the issue type, confirm the assigned priority and check the versions that are affected by the bug. The bug triage process does not end once it has been sent to the developer to resolve. Checking up on the progress of solving the report is also part of the process, which is necessary in the case that the process needs to be restarted. Another part of the bug triage is revisiting an old issue. If an older report is encountered that hasn't been solved yet, it should be re-evaluated to ensure that the report is still valid and needs to be investigated. To sum up, the process of bug triage is an important step in bug report analysis, which confirms the validity of bug reports and ensures they are assigned to the right developers.

Automating the process of bug triage is a complicated issue. On the other hand, there has been much research done in optimizing and automating different parts of the process. For example, one problem in bug triage is that when a bug report has been assigned to a developer by accident or if another developers' expertise is necessary to resolve the bug, the report gets reassigned. This happens in the case of between 37%-44% of bug reports in Mozilla and Eclipse and leads to increased resolution time (Jeong, Kim and Zimmermann, 2009). A research done by Jeong et al. introduces a model based on graphs, which focuses on assigning the bug reports to developers with the necessary qualities to resolve them. Their model reduces reassignment of bug reports by up to 72% (Jeong, Kim and Zimmermann, 2009).

Another research focuses on automating bug triage by using text categorization. The researchers identify that the amount of developer resources necessary to triage a bug report grows larger with the size of the open-source projects and propose the use of machine learning techniques to assist in the process by using text categorization to predict the developer that should be working on the bug from the bug's description. Their prototype uses supervised Bayesian learning and achieves an accuracy of 30% (Murphy and Cubranic, 2004). The same problem has been tackled by Jifeng et al. who propose a semi-supervised approach, which would avoid the problem of not having enough labeled data. The approach uses naive Bayes classifier and expectation maximization and trains the model twice by firstly training only on labeled data in order to classify the unlabeled

data and then trains again on the fully labeled dataset. The accuracy of their models reaches up to around 48% (Jifeng et al., 2017). Other research papers also tackle the problem of who can fix a certain bug report. Ahsan et al. propose the use of Support Vector Machine (SVM) combined with feature selection and latent semantic indexing in order to categorize bug reports based on the developer they get assigned to and trains a model with 44.4% testing accuracy (Ahsan, Ferzund and Wotawa, 2009). Jifeng et al. on the other hand, extract attributes from historical bug data sets to determine the order of applying instance selection and feature selection algorithms in order to achieve data reduction, which they input in three algorithms (Support Vector Machine, K-Nearest Neighbor and Naive Bayes) in order to triage the bug reports. With that they achieve up to 60.4% testing accuracy on a test set of around 2100 Eclipse bug reports (Xuan et al., 2015). Another research focuses on training set reduction when it comes to bug triage. Just like the previously described research it uses feature selection and instance selection algorithm, which in the previous research are applied sequentially while in this one they are combined and again a Naive Bayes model is trained to categorize the bug reports. Their model reaches an accuracy of 66.95% on the testing set of an Eclipse collection of bug reports when it is set to provide a list of 10 recommended developers (Zou et al., 2011). A research done by Hu et al. leads to developing a software tool called BugFixer, which gathers information about developers on a team and instead of trying to classify a bug report to one developer, helps a triager by providing them with a list of suitable developers to tackle on a bug. This tool is then evaluated on three large-scale projects and two smaller industrial projects and it proves that even though it is as good as conventional methods which use SVMs on small projects, it performs a lot better than them on large-scale projects (Hu et al., 2014).

A significantly different approach has been proposed by Park et al. who unlike other experiments based on content-based recommendation it also takes into account a content-boosted collaborative filtering in order to reduce overloading more experienced developers. Doing this they reduce the cost of bug triage efficiently by 30% without taking a serious hit on the accuracy of the automation (Jin-woo et al., 2011).

2.1.3 Bug Localization

A considerable issue when it comes to fixing software bugs is finding the location of the code that produces the bug. The process of doing that is called bug localization. When it comes to dealing with smaller software systems that is a hard and time consuming task and when a bug is reported in a large-scale system it is a lot harder and requires a lot more time and effort. Finding the location of a bug requires in-depth knowledge of the software system by the developer. Another reason why this process is so complicated is due to the fact that bug reports do not have a specific structure. Vague and unclear descriptions of bugs could lead the developer to look for the bug in a very different location. The process of finding the location of a bug is not only hard but very expensive as well and could sometimes lead to much longer resolution times. This identifies a need for the automation and development of tools to expedite this process.

Fortunately there has been a lot of research done in that field. A statistical model-based bug localization approach, called SOBER, is used to localize bugs without any prior knowledge of the program semantics. The way it works is by firstly evaluating patterns of predicates in correct and incorrect runs and then classify a predicate as bug-relevant if its pattern in incorrect runs differs from the one in correct. The proposed approach was evaluated by running it on the Siemens suite, SOBER managed to locate 68 out of 130 bugs (Liu et al., 2005), which shows an accuracy of 52.3%. Another approach called BugLocator has been proposed by Zhou et al. The approach is based on an information retrieval method and attempts to locate the files relevant to the bug. It does so by ranking all the files in a system by the similarity they have to the bug report, with the use of revised Vector Space Model. BugLocator would also take into consideration information about bugs of similar nature that have been fixed before. The best achieved accuracy when using this approach is 62.6% on Eclipse 3.1 where the relevant files to locate the bug were ranked in the top ten from 12,863 files (Zhou, Zhang and Lo, 2012). There are many other tools for locating bug reports, one such tool is called BLUiR, which, just like BugLocator, is also based on information retrieval. This tool is based on the idea that “structured information retrieval based on code constructs, enables more accurate bug localization.”(Saha et al., 2013) BLUiR takes as an input the source code of the software system and bug reports and finds the locations of the bugs based on similarity data. Using

BLUiR on the Eclipse bug report dataset with 3075 bugs from 12,863, it managed to find 2010 bugs in the list of top 10 probable location (Saha et al., 2013), which shows an accuracy of 65.4%.

There is also an approach presented by Dallmeier et al., called iBUGS, which can be used to evaluate different tools for bug localization. They create a benchmark dataset, where 369 bugs were extracted from ASPECTJ to be used in the evaluation. 201 of the bugs in the dataset were local, requiring modifications to only a few methods. Also, 10% of bug fixes require corrections to a single line of code and 44% of all required required changes to 10 lines or less. The iBUGS approach was applied to the AMPLE tool. AMPLE is a tool which tries to capture the control flow of a program as sequences of method calls issued by the classes in the program (Dallmeier and Zimmermann, 2007).

A newer method for bug localization has been developed by Malhotra et al. which uses a text mining approach and a multi-objective NSGA-II algorithm to recommend a list of classes by lexically comparing bug reports and API descriptions. The proposed method was validated on three applications and shows better results than traditional methods of locating software bugs (Malhotra et al., 2018).

Bug localization has also been done using deep learning methods. One such research focuses on developing a combination of deep learning and an information retrieval technique, called revised Vector Space Model (rVSM) to try and determine the location of the bugs. The deep neural network is used to create relations between terms in bug report to different tokens in the code and terms in the source files, while the rVSM technique is used to compare the textual similarities between bug reports and source files. The research done by Lam et al. shows that the previously described combination works well in an attempt to locate the buggy files. Also when the model was combined with the history of previously fixed bugs it manages to predict the correct file in 50% of the cases on the first recommendation, 66% in the top 3 and almost 70% in the top 5 recommendations (Lam et al., 2017).

2.1.4 Bug-fix Time Estimation

Estimating how long it will take to fix a bug is a complex task. Predicting the time it will take to fix a bug is much harder than other software development tasks, such as

developing new features for example. The reason for that is because developing new features is a construction task and fixing a bug is mainly a search task. Sometimes in order for bug to be found it takes a long time and requires going through a lot of code, running the software or just parts of it, following states and even going through previous versions of the software. Once the issue has been found, it is a matter of modifying the code to fix it which is again a construction task that can be more easily estimated. Prediction of the bug fixing time is another thing that helps the bug triaging process. If the triager knows that a particular bug with a high priority is going to take a longer time to fix, they can assign more developers to it. Bug resolution time can also help with scheduling future software events such as scheduling testing activities or estimating release times of new versions. That is why anticipating the time it will take to fix a bug is an important part of bug report analysis.

There has been numerous papers and experiments conducted on the problem at hand. One research paper published by Zhang et al. presents an empirical study performed on predicting bug fixing time in commercial software projects. They perform their research on a company called CA Technology (Ca.com, 2019), which is a multinational company that provides IT management software and solutions. The paper analyses the way bugs have been handled in three of the company's projects and based on the research proposes a Markov-based method to predicting how many software bugs will be fixed in the future. For some of the bugs a method is proposed to predict the amount of time it will take to fix them. Another method proposed in the same paper involves classifying bug reports into two categories based on the bug-fix time (fast or slow). The models, evaluated on the same 3 projects from CA Technology, confirmed the validity of the three methodologies. For the first one the best results were achieved to be an MRE value of 0.015%. For the second methodology the average MRE achieved was 6.45% and for the third methodology the accuracy was in the range of 65% to 85%. Based on the scale of the data the MRE values achieved for the first and second methodology showed very promising results (Zhang, Gong and Versteeg, 2013). Another research paper, also performs an empirical study, but with the goal of achieving an understanding of delays between bug assignment and the start of bug fixing in mind. The empirical research involved examining factors which would affect the bug resolution time and

compare them with the use of descriptive logistic regression models. They found that high severity of bug reports leads to a reduced delay time and the size of the code to be examined could lead to longer delays (Zhang et al., 2012). Predicting the time and effort has also been attempted with the help of the Lucene framework, which was used to find similar, previous reports and use their mean time as a prediction. The proposed method was evaluated on effort data collected from JBoss and obtains predictions which differ from the actual effort by an hour (Weiss et al., 2007).

Other papers that involve predicting the time to fix software bugs propose methodologies using a bit more complex machine learning algorithms. One such paper uses a decision tree algorithm to classify the resolution time in fast and slow. The evaluation of their method on the Eclipse Platform predicted the fast fixed bugs correctly in 65.4% of the cases (Giger, Pinzger and Gall, 2010). Another research focuses on recognizing bugs that would be very fast or very slow to fix. Using data from Mozilla, Eclipse and Gnome and then keeping only the bugs in the 25th percentile as very fast and the 100th percentile as very slow. That data is then used to train a Naive Bayes classifier and achieves a precision accuracy rate of 39% for very fast bug fix times and 49% for very slow on the EclipseJDT project (Abdelmoez, Kholief and Elsalmy, 2012).

Some research has also been done on using neural networks as well. A methodology proposed by Zeng et al. focuses on estimating the fix effort, rather than the fix-time of software defect using self-organizing neural networks. They propose the use of the Kohonen network, which is an unsupervised network with the ability to self-organize. The model proposed was evaluated on 70 defects in a dataset called KC3 by NASA, and achieved a maximum mean root error ranging from 23% to 83%, which points to the conclusion that the proposed model is less than excellent in effort estimation (Zeng and Rine, 2004).

2.1.5 Duplicate Bug Report Detection

Duplicate bug reports are a very common occurrence in open bug repositories. Very often reporters tend to report bugs without checking if that software bug has already been reported before. Other times reporters, due to different technical knowledge report the same bugs in different ways. That could lead to a lot of resources from developers

wasted on bug reports which have already been reported, fixed or in the process of fixing. On the other hand, those duplicate bug reports if identified early could lead to helping the developers assigned to the original reports by providing additional information, if the bug has not been fixed yet. Due to the importance of detecting duplicate bug reports, in order to save costs of triage and improve resolution time, there has been a lot of research carried out in that field.

One such research developed an approach using sparse vector computation to detect duplicate bug reports which achieved a precision rate of 29% and recall - 50% on the Firefox project. Even though the precision and recall are relatively low, the approach has been used to increase the accuracy of duplicate detection of novice bug triagers (Hiew, 2006). Another research proposes a retrieval function based not only on the textual content in the summary and description but also using other features such as product, component version and so on. It then uses a similarity formula called BM25F and a stochastic gradient descent algorithm on a labeled training set. When applied to an Eclipse dataset consisting of 209,058 bug reports, the recall achieved was between 37% - 71% and the mean average precision was 47% (Sun et al, 2011). Alipour et al. on the other hand propose adding contextual information to state of the art systems for duplicate detection. Their proposed system shows that context should not be ignored when using information retrieval tools (Alipour, Hindle and Stroulia, 2013). It has also been shown that using discriminative models such as Support Vector Machines for information retrieval could increase the accuracy of duplicate bug report detection. A research using that model shows a highest result of up to 65% recall rate for the Eclipse dataset and compares it to a model proposed by Wang et al., which uses natural language and execution information and achieves a recall rate of 49% on the same dataset (Wang et al, 2008, Sun et al., 2010). Another paper also focussing on information retrieval introduces an approach which tries to learn different descriptive terms using historical data consisting of duplicate reports in order to make connections. The empirical evaluation of the proposed approach shows improvement of other advanced approaches by up to 20% in accuracy (Nguyen et al., 2012).

Based on the papers described before and the results they achieved duplicate bug detection remains a challenging task for simpler algorithms. However with the increasing

popularity of neural networks in the past few years there have been some new research papers published that involve using deep learning techniques to improve the automation of duplicate bug report detection. An experiment conducted by Budhiraja et al. proposes a deep word embedding network for duplicate detection, in 3 main steps. The first one is training a dataset of bug reports on a word embedding neural network, then transforming only pairs of duplicate bug reports into document vectors and then feeding it through a deep neural network which when trained on a Firefox project bug report dataset reaches up to 70% recall rate (Budhiraja et al., 2018). Another research, carried out by Deshmukh et al. involves a retrieval and classification models using Siamese Convolutional Neural Networks and Long Short-Term Memory in order to accurately detect duplicate reports. The proposed model is a complicated ensemble of neural network models involving a single layered neural network for the structured information, a bidirectional LSTM for the short description and CNN for the long description of each report which are then combined in order to detect their similarity. This proposed model performed significantly better than the previously described approaches by achieving an accuracy of nearly 90% and a recall rate close to 80% on a dataset consisting of Eclipse bug reports (Deshmukh et al., 2017).

2.1.6 Priority Detection

When it comes to the bug triage process there is another component that needs to be considered except determining which developer a bug report gets assigned to. This other component is determining the priority of a bug report. Prioritizing a bug report happens on 5 levels, where it starts at level 5 - the lowest priority, which indicates software bugs which do not affect the execution of the software and may not be fixed for a long time, if they are fixed at all and goes down to level 1 which is the highest priority, denoting software bugs that need to be fixed immediately before the release of the next version of the software. Determining the priority of a bug report is performed by the bug triager manually and may take a long time. It is done by analyzing the information in the bug report and comparing it to similar bug reports in order to decide the appropriate priority level. This identifies a need for developing a system to automate the process of predicting a bug report priority.

A number of such systems have already been developed using different machine learning methods. One experiment shows that the automation of this process can be done using Naive Bayes and Support Vector Machine classifiers. It also focuses on determining which features from a bug report are most suitable for this process. They use a collection of bug reports from the Bugzilla bug repository to evaluate different combinations of textual and numerical features using both Naive Bayes and Support Vector Machine. Their experiment shows that the highest accuracy is achieved by an SVM when both numerical and text features are combined for training with a precision of 55% (Kanwal and Maqbool, 2012). Another experiment tries a similar approach using multi-factor analysis, taking into consideration the different features available in a bug report. The researchers extract the features and train a model based on linear regression with enhanced threshold, where instead of classifying into categorical values they use ordinal values, where they order the values from 1 to 5. Same as the previously described experiment, the model in this one is trained on a collection of bug reports from the Bugzilla bug repository and the average precision accuracy achieved was 29.73% (Tian, Lo and Sun, 2013). Another research using a similar approach has been done by Alenezi et al. In their paper an experiment is performed to categorize bug reports using Naive Bayes, Decision Tree and Random Forest algorithms. However, unlike the other 2 previously described experiments, this one tries to categorize the bug reports in 3 classes based on the priority (high, medium and low). The models are then evaluated on 4 datasets, two from the Eclipse both with different features and the other two from the Firefox project also with different features. The best results were obtained using the Random Forest algorithm, achieving an accuracy of 61.2% precision when classifying bug reports. The paper concludes that Random Forest and Decision Tree outperform Naive Bayes in all the cases (Alenezi and Banitaan, 2013).

Another similar research paper covers the use of Support Vector Machine, Naive Bayes, K-Nearest Neighbors and Neural Network, but unlike the other papers discussed in this chapter it uses cross project validation to evaluate the performance of those methods. The paper tries to answer which of the machine learning techniques is more appropriate when it comes to predicting priority as well as if cross project validation works for this problem. It was concluded that Support Vector Machine and Neural Network give the overall highest accuracy from the previously mentioned techniques and also that

using cross project validation works for priority prediction with more than 72% accuracy (Sharma et al., 2012).

2.2 Sentiment and Textual Analysis

This subsection of the background will focus on exploring the different types of textual analysis relevant to this project. For sentiment analysis it will provide a description and a detailed explanation about its methods and features as well as applications. It will also focus on the software used to obtain the sentiments in this project and some examples of the results obtained will be shown. The textual processing subsection will explain different relevant text processing tools and will contain a description of the tools used in this project.

2.2.1 Sentiment analysis

Sentiment analysis also known as Opinion Mining is a part of Natural Language Processing (NLP) that builds systems which try to identify and extract opinions within text. In addition to that these systems extract expression attributes, such as subject, polarity and opinion holder (MonkeyLearn, 2019).

Extrapolating the polarity of a text is a descendant of the General Inquirer method (Stone, Dunphy and Smith, 1966) which provides hints towards quantifying patterns in text as well as a research done to measure the psychological states through the content of verbal behaviour (Gottschalk, 1969). Later on a method was developed, patented by Volcani and Fogel, which would determine and control the impact of text (Volcani and Fogel, 2001). It indicates lexical impact of words in a text and provides the user with various statistics relating to the lexical impact of the text, in addition to providing suggestions for increasing the sentiment by replacing certain words within the text with similar words.

Research has also been done on using sentiment in bug reports. One research investigates the sentiment of bug-introducing and bug-fixing commit messages on GitHub and finds that the commits are mostly positive rather than negative (Islam and Zibran, 2018). However it is unclear if there is a connection between the sentiment of the bug report and the way it was handled by the developers. Another research paper on

“Sentiment Based Model for Predicting the Fixability of Non-Reproducible Bugs” finds that including sentiments into the analysis increases the prediction accuracy from 2 to 5% for various classifies (Goyal and Sardana, 2017).

Some researches show that there are different ways to represent the polarity of texts. One way would be to represent the polarity with three values, one for positive, negative and neutral sentiment. Some experiments use a single value with a range from minus five for example to plus five. This would indicate that a text with severe negative polarity would have a value of minus five where neutral would have a value of zero and severely positive would have a value of plus five. However, text can have both positive and negative polarity, since they are not mutually exclusive. In this project there will be two value for sentiment. One for positive ranging from 1 to 5 where 5 is highly positive sentiment and one for negative ranging from -1 to -5 where -5 is highly negative sentiment.

There are different software programs that can be used to extrapolate the polarity of a text. One such program is HubSpot’s ServiceHub. It is a tool which breaks down qualitative survey responses and analyses them for positive or negative sentiment. Another such program is Quick Search, which is a sentiment analysis tool that is a part of a larger platform called Talkwalker. This tool is best suited for use with social media channels. Other programs also exist for analysing sentiment of text such as Repustate, Lexalytics, Sentiment Analyzer, SentiStrength and others. Most of those software programs would require payment to use their services, others are not suitable for use with bug reports. For this project the chosen software used is SentiStrength (Sentistrength.wlv.ac.uk, 2019). It is fast and free, has the option to use two scores one for negative and one for positive sentiment and is also suitable for informal language, which is the case with most bug reports. In order to show how it works, a sample bug report description and text is taken and input to the sentiment software and detailed results from SentiStrength are provided in table 2 below.

Title: Internationalization (NSL)
Description: "vcm is not nsl ready! need resource bundles. policy has bind() methods ready for use. notes: painfully all six vcm plugins will have to have a policy class and duplicate the code for generating strings from bundles. isnt core supposed to be providing support and a story for this? internationalization complete. each plugin currently has getresourcestring(string) and getresourcestring(string object[]). core has no support for us. in the future we should shorten the names (e.g. bind) because theyre used so much. moving to inactive for future consideration."
Title Sentiment: Internationalization[0] (NSL)[0] [sentence: 1,-1]
Title Sentiment Result: 1, -1
Description Sentiment: vcm[0] is[0] not[0] nsl[0] ready[0] [[Sentence=-1,1=word max, 1-5]] need[0] resource[0] bundles[0] [[Sentence=-1,1]] policy[0] has[0] bind[0] methods[0] ready[0] for[0] use[0] [[Sentence=-1,1]] notes[0] painfully[-3] all[0] six[0] vcm[0] plugins[0] will[0] have[0] to[0] have[0] a[0] policy[0] class[0] and[0] duplicate[0] the[0] code[0] for[0] generating[0] strings[0] from[0] bundles[0] [[Sentence=-4,1]] isnt[0] core[0] supposed[0] to[0] be[0] providing[0] support[1] and[0] a[0] story[0] for[0] this[0] [[Sentence=-1,2]] internationalization[0] complete[0] [[Sentence=-1,1]] each[0] plugin[0] currently[0] has[0] getresourcestring[0] string[0] and[0] getresourcestring[0] string[0] object[][0] [[Sentence=-1,1]] core[0] has[0] no[0] support[1] for[0] us[0] [[Sentence=-1,2]] in[0] the[0] future[0] we[0] should[0] shorten[0] the[0] names[0] e[0] [[Sentence=-1,1]] g[0] [[Sentence=-1,1]] bind[0] because[0] theyre[0] used[0] so[0] much[0] [[Sentence=-1,1]] moving[0] to[0] inactive[0] for[0] future[0] consideration[0] [[Sentence=-1,1]][[[2,-4 max of sentences]]]
Description Sentiment Result: 2, -4

Table 2. Example of the results of sentiment extraction from SentiStrength.

As it can be seen from Table 2, the SentiStrength software takes as an input text and splits it into words and sentences, where it gives a sentiment value for each word and then sums it up to give a positive and negative value from each sentence. Once that is done it takes the maximum values for positive and negative from the sentences as the final result. Due to the title being so short the resulting sentiment is neutral since both the positive and negative sentiments are the same values. As for the description the sentiment is mostly negative due to certain words which show negativity in the text.

The way SentiStrength works is by taking a lexical approach. It contains “1,125 words and 1,364 word stems” (Khan, 2019), where each word contains a score for positive or negative sentiment. When these match to a word in the text supplied it indicates the presence of sentiment and its strength. For example in table 2 above it can be seen that the word painfully has a negative sentiment of -3. Another useful feature in the SentiStrength software is negation. That is when a positive term succeeds a negating word

it sentiment is flipped and when negative terms have a positive word before them they are neutralized (Khan, 2019).

2.2.2 Natural Language Processing

Natural language processing (NLP) is the technology used to help computers understand human language. The main objective of NLP is to read, decipher, understand and make sense of the human languages in a manner that is valuable (Garbade, 2018). Natural language processing entails applying a set of algorithms to extract useful information from text, which could then be used to train machine learning models on it. Some techniques used to extract useful information from text include but are not limited to: lemmatization, word segmentation, parsing, sentence breaking and stemming.

Stemming and lemmatization are both used to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form (Manning, Raghavan and Schütze, 2018). However, stemming refers to a heuristic process which focuses on removing the endings of the words in order to reduce them to their base words (stems) and lemmatization involves using a vocabulary and morphological analysis of words to reduce them to their base words. Due to the fact that this experiment involves descriptions of bugs and they contain different error messages and variable names which are not suitable for use with stemming or lemmatization.

Word segmentation involves dividing a string into its component words, which in English and many other languages can be done by dividing the sentences using a word divider which in this case is the space character. That is one of the processes that is going to be applied to the textual features to prepare them for machine learning models.

Parsing refers to the formal analysis by a computer of a sentence into its constituents, resulting in a syntax tree which represents the syntactic structure of a string. Parsing is preceded by lexical analysis, also known as tokenization, which is the process of replacing sensitive data with unique identification symbols that retain all the essential information about the data (Rouse, n.d.). Tokenization separates tokens using simple heuristics and following a few steps. Firstly, words are separated by whitespace, punctuation marks or line breaks. Then all characters from a continuous string (word) make up one token.

As mentioned earlier, another part of natural language processing is sentence breaking. Sentence breaking is a technique to separate sentences. The standard approach to sentence breaking involves 3 rules. Those are that a sentence ends if there is a period and if the next token is capitalized and it doesn't end a sentence if the preceding token is in the list of abbreviations.

2.3 Machine Learning and Neural Networks

Machine learning is a method of data analysis that automates analytical model building. It is a branch of artificial intelligence based on the idea that systems can learn from data, identify patterns and make decisions with minimal human intervention (Sas.com, 2019). There are many different types of machine learning systems which split into different categories based on whether they are trained with human supervision, whether they can learn incrementally or on the fly and whether they are instance-based or model-based (Géron, 2017). In the case of bug report analysis based on the dataset obtained it can be concluded that the necessary machine learning model would have to use supervised learning since the labels for each instance are available, also the model would have to use batch learning and be model-based, because it would have to detect patterns in the training data and build a predictive model. There are many challenges which arise when training a machine learning model, such as insufficient quantity or unrepresentative data, poor quality, irrelevant features, overfitting and underfitting. Those challenges could all be overcome in the case that they occur by taking different measures against it. In the case of bug report analysis those measures involve changing the complexity of the models and increasing the quantity of data.

There are also different types of machine learning systems depending on whether the goal of the model is to classify the instances into different categories based on a set of changing variables or try to determine the relationship between a target variable and a set of changing variables. Those are called classification problems and regression problems respectively. This project will involve both classification and regression problems. When training a machine learning model it is necessary to compare it to a baseline model. The goal of the machine learning model trained would then be to perform better than that baseline model. The baseline models that are going to be examined here

are linear regression and decision tree. Linear regression is a machine learning algorithm which performs a regression task based on supervised learning. Decision trees same as linear regression is one of the simpler machine learning systems. It splits the data according to certain parameters and can perform both classification and regression tasks. A part of machine learning is called deep learning. It is based on artificial neural networks. Example artificial networks are multi-layer perceptron, convolutional neural network and recurrent neural network which will be covered in the next subsections.

2.3.1 Multi-Layer Perceptron

A Multi-Layer Perceptron (MLP) consists of at least three layers, where the first layer is an input layer and the last layer is the output layer. The layers in between the first and last layers are called hidden layers. Every layer in an MLP except for the output layer has a bias neuron and is fully connected to the next layer. When a neural network has more than 2 hidden layers it is called a deep neural network (DNN). The way the MLP works is by taking in a set number of inputs passing it through each layer and changing the values based on a set of weights and an activation function. When it reaches the final layer the result is the prediction. It then compares the prediction to the actual value and calculates the error rate. Once that is done, it passes the information backwards calculating the error gradient across the connected weights in the network. Figure 1 below shows an example of a simple MLP.

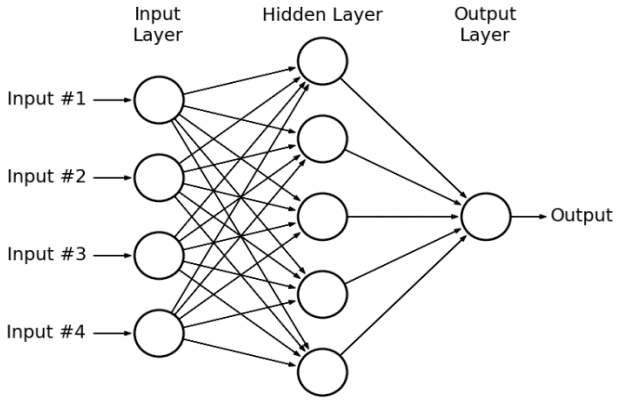

Figure 1. A simple Multi-Layer Perceptron.

It has been shown that MLPs can be trained to approximate almost any smooth, measurable function, provided with enough hidden units are available (Hornik,

Stinchcombe and White, 1989). Unlike other techniques the multi-layer perceptron makes no prior assumptions about the distribution of the data. It can be trained to generalise very accurately over, future unseen data (Gardner and Dorling, 1998).

2.3.2 Convolutional Neural Network

Convolutional Neural Network (CNN) is a type of deep neural network. They are in general regularized version of MLPs. Unlike multi-layer perceptrons, not all neurons in CNNs are fully connected which makes less prone to overfitting data. Like the name suggests those networks use an operation called convolution. Those networks are simply neural networks with at least one convolutional layer. CNNs are particularly useful in image classification problems but they also perform successfully in other tasks like natural language processing and voice recognition.

Using convolutional layers allows the network to concentrate on low-level features in the first hidden layer, which are then assembled into higher level feature. The way a convolutional layer works is by using receptive fields with different sizes. Each receptive field is considered one filter. Usually a CNN would have many filters since each filter can only learn one feature. Convolutional layers could also have different dimensions based on the input data. For natural language processing the perceptive field will only have 1 dimension, for images it is 2 dimensions and for video - 3 dimensions.

Another characteristic of CNNs is that they use pooling layers as well. Once features have been extracted using a convolutional layer, a pooling layer is used to shrink the input and reduce overfitting. There are different kinds of pooling layers but the most common ones use a mean or a max aggregation function. Those layers do not have weights associated with them. Figure 2 below shows a simple CNN used for natural language processing.

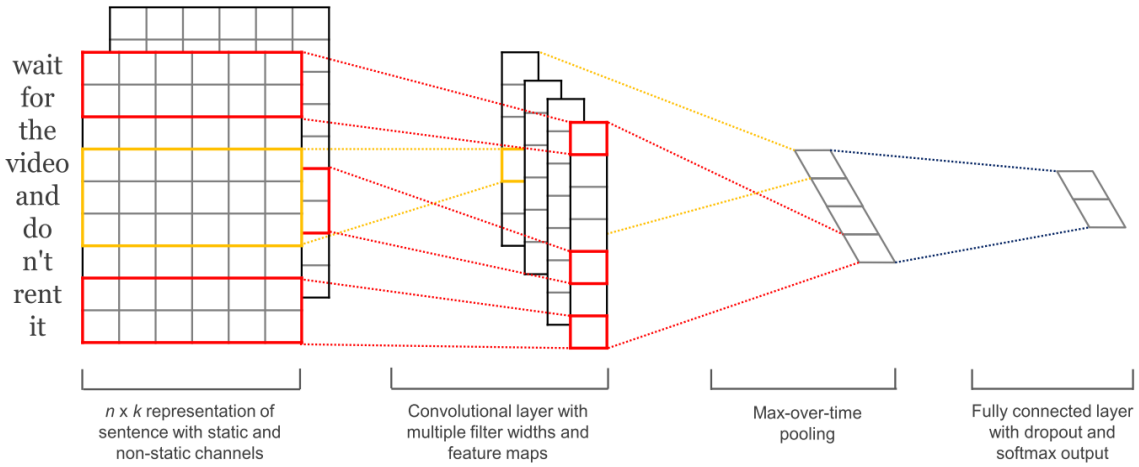

Figure 2. A Convolutional Neural Network for Natural Language Processing.

CNNs often achieve excellent results when used for NLP. A research done on deep convolutional neural networks for sentiment analysis of short texts achieves a sentiment prediction accuracy of 86.4% for the Stanford Twitter Sentiment corpus, which contains Twitter messages (dos Santos and Gatti, 2014). Other works done on sentiment analysis and question classification using NLP show that a simple CNN with little hyperparameter tuning and static vectors achieves excellent results. It achieves an 81.5% accuracy on a movie review dataset with one sentence where the goal is to detect if the review is positive or negative. It also achieves an accuracy of 89.6% for the opinion polarity detection subtask of the MPQA dataset (Kim, 2014). A research carried out by Alexis Conneau et al. presents a new architecture for a very deep convolutional network for the task of text processing which operates directly on the character level and uses only small convolutions and pooling operations. The architecture presented by them contains 9 convolutional layers with small parameters and 3 pooling layers. Their work shows that increasing the number of layers and not the size leads to better performance of the CNNs (Conneau et al., 2016). There has also been some research done on designing convolutional neural networks with recurrent layers. A research carried out by Siewi Lai et al. at the National Laboratory of Pattern Recognition show that applying a recurrent structure to capture contextual information as much as possible when learning word representations, may result in less noise compared to traditional neural networks. The recurrent CNN designed achieves an accuracy of 96.49% on the 20Newsgroups dataset with around 2% improvement over a CNN (Lai et al., 2015).

Another research on using CNNs for text classification suggests a semi-supervised method. The idea is that integrating a supervised CNN with learned embeddings of small text regions from unlabeled data would perform better in a text classification problem. The results obtained on the RCV1 (news) dataset with 55 single label classes show an error rate of 7.71% compared to a regular CNN which achieved an error rate of 9.17% (Johnson and Zhang, 2015).

Based on the excellent results achieved by convolutional neural networks in NLP tasks that was one of the architectures chosen for this project.

2.3.3 Recurrent Neural Network

A recurrent neural network (RNN) is a feedforward neural network where the network also has connections backwards. RNNs can exhibit temporal dynamic behaviour. They can also use their internal state to process sequences of input, which makes them useful for many different tasks such as natural language processing for automatic translation, speech-to-text or sentiment analysis as well as analyze time series data such as stock prices, anticipate car trajectories and so on. There are different kinds of RNNs, such as fully recurrent, Hopfield, recursive, gated recurrent unit (GRU), long short-term memory (LSTM) and others. A fully recurrent neural network is a simple RNN where all the connections between the nodes form a directed graph along a temporal sequence. Hopfield is an RNN, in which all the connections between the nodes are symmetric and requires stationary inputs. Another kind of RNN is a recursive neural network which has the same weights applied to it recursively over a differentiable structure in topological order. Gated recurrent units are similar to LSTM networks but have a forget gate and lacks an output gate. The last RNN mentioned earlier is the LSTM. Long short-term memory neural networks were created to deal with the vanishing gradient problem, that can be encountered with the other RNNs. LSTMs work just like RNNs but can retain information for longer periods of time. They have a chain structure that consists of four neural networks and different memory blocks. Information in LSTMs is retained by the cells and memory manipulations are performed by three gates, which are the forget gate, used to remove information which is no longer useful, the input gate, used to add useful information to the cell state and output gate, used to extract useful information from the

current cell state and present it as an output (GeeksforGeeks, n.d.). Figure 3 below shows a basic architecture of an LSTM.

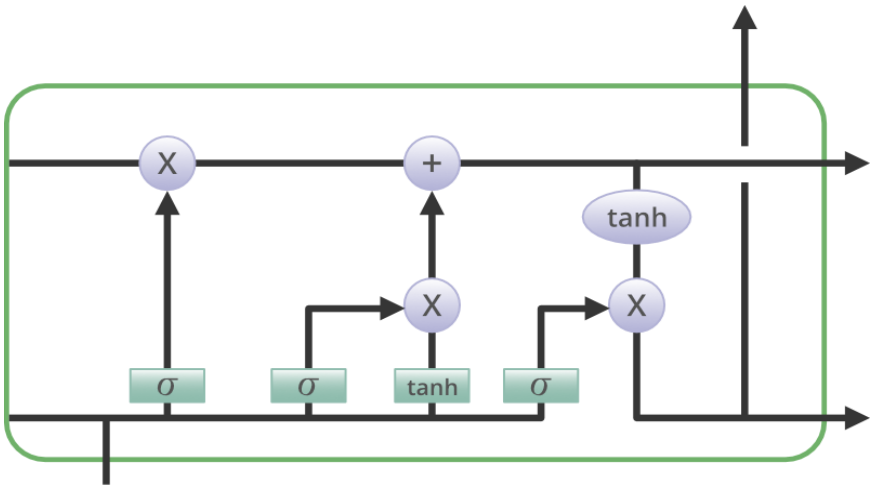

Figure 3. A Long Short-Term Memory Network (GeeksforGeeks, n.d.).

LSTMs have a wide range of usage when it comes to working with sequences and there are many research papers published using LSTM models. One such paper presents an improved semantic representation from tree-structure LSTMs. Kai Sheng Tai et al. introduces a generalization of LSTM to tree-structured network topologies, which outperforms existing systems and basic LSTMs on predicting semantic relatedness of two sentences and sentiment classification. One of the models they propose achieves an accuracy of 88% over a binary sentiment classification on the Stanford Sentiment Treebank dataset. That is 3.1% improvement over a regular LSTM and 0.5% higher than a Bidirectional LSTM (Tai, Socher and Manning, 2015). Another NLP related research paper involves using and LSTM to perform word segmentation on texts in Chinese. The paper proposes 4 LSTM network models which achieve a 97.5% test accuracy on the MSRA dataset, which consists of images with texts in chines on them (Chen et al., 2015).

Other uses of LSTM involve training them for relation classification. Relation classification as the name suggests involves making relations between entities of sentences, which could help natural language processing in tasks such as sentence interpretations, knowledge graph construction and so on (Ren et al., 2018). A research done on using LSTM for relation classification suggests that a bidirectional long short-term memory network can achieve an F1 value of 84.3 compared to other research papers using CNNs for the same task achieving values of 84.1 (Zhang et al., 2015). The F1 value is

a measure of a test's accuracy represented by multiplying the precision and the recall of a neural network and then dividing it by the sum of it.

Another research shows that cached LSTMs can be used for sentiment classification on long texts such as documents. The research done by Xu et al. proposes an LSTM with a caching mechanism, which divides memory into several groups, which enables it to keep sentiment information better. The proposed network seems to outperform other state of the art models, including models based on CNNs (Xu et al., 2016). Similarly, an experiment to predict the polarities of twitter messages has been performed by Wang et al. They use an LSTM recurrent network with constant error carousels in the memory block structure and perform their experiment on noisy data achieving sentiment prediction accuracies better than other feature-engineering approaches. An evaluation on a negation phrase test set, shows that the architecture developed by them using LSTMs doubles the performance of non-neural models (Wang et al., 2015).

According to all these research papers based on using LSTMs for NLP it suggests that those kinds of neural networks can perform very well when training on textual data.

Chapter 3: Research

The previous chapter of this report focused on providing the relevant background to this experimental project. Based on the literature review and background information it was determined that the focus of this project is to conduct an experiment which focuses on bug-fix time and priority. This chapter will go into more specifics about the exact goals of this experiment, defining the hypotheses and provide details about the design of the experiment.

The background section included information about the different kinds of bug reports analysis as well as the available tools to perform these analyses, while this section would provide more specifically defined and innovative aims of this experiment. The information on sentiment and textual analysis and the available information in the bug reports would be used to describe how the dataset of bug reports will be used to design

the experiments which would determine the success of this experiment and the experiment design subsection of this chapter will show how exactly with the use of the machine learning models described those hypotheses will be answered.

3.1 Methodology

After determining the specific focus of this project, which is the analysis of bug report fix-time and priority, it was necessary to define a set of hypotheses in the next subsection, which validity will be then supported or refuted by the conduction of the experiment designed in the subsection afterwards.

3.1.1 Hypotheses

One of the fundamental parts of conducting an experimental research project is to build a set of hypotheses in order to design the experiment in a way to prove or disprove their statement. Hypothesis is a statement in which the result of an experiment is speculated upon (Shuttleworth and Wilson, n.d.). The hypotheses in this case have been generated based on the research questions and the background research conducted in the previous section. Observations upon the analysis on bug reports and textual and sentiment analysis has led to the conclusion that it may be possible to predict the resolution time of a software bug based on the information provided in the bug report. This has led to formulating the following two hypotheses:

Hypothesis 1: The title and description of a bug report can be used to determine the time it would take to fix a bug.

Hypothesis 2: Adding the numerical features of a bug report (including sentiment of the description) to the title and description can improve the results of determining the resolution time of a bug report.

Also by researching into classification and regression problems it is thought that the bug-fix time prediction might give better results if the resolution time is split into time frames which would turn the problem into a classification problem. This lead to the formulation of the 3rd hypothesis which is:

Hypothesis 3: The results produced from predicting the resolution time would be more accurate if they were transformed into categories for classification.

Taking another look at the data available and necessary to predict the resolution time of a bug report it was determined that the same information could also be used to determine the priority of the bug report by adjusting the experiments slightly. This lead to the belief that the following two hypotheses could also be supported by the experiment:

Hypothesis 4: The title and description of a bug report can be used to determine the priority of a bug report.

Hypothesis 5: Adding the numerical features of a bug report (including sentiment of the description) can be used to improve the accuracy of the priority prediction.

3.1.2 Experiment design

In order to validate the defined hypotheses it was necessary to design an experiment. This experiment involves creating a set of ensemble neural networks which would take as an input a bug report dataset and depending on the hypothesis output either the resolution time or the category. The first step is to ensure the data is prepared for a neural network to be fitted to it. The initial dataset consists of 85,156 bug reports which were described earlier in the background section.

Data Preparation

In order to prepare the dataset for fitting onto a neural network it is necessary to do pre-processing of the dataset. Starting with the initial dataset, the first step would be to filter the noise from the data and deal with null values. Then the remaining data would be converted to numerical values except for the title and description. Where necessary one hot encoding will be performed. Once that is done, it is important to extract some additional information from the title and description, such as length of texts and sentiments. Once all the numerical data is prepared the next step would be to pre-process the textual data by performing some natural language processing methods on it. The first thing would be to remove all punctuation and digits, as well as stopwords. Once that is done the labels would need to be prepared in different ways depending on the goal of the

neural network model. For the bug-fix time prediction it is essential to calculate the labels and if necessary transform them to categorical data or scale them. Once all that is done a train-test split will need to be performed. The data will be split into 80% training data and 20% testing data and the textual features will need to be separated from the numerical as they will be fed to different neural network models.

Experiments

To prove hypotheses 1 and 3, 2 neural networks will be trained on the training data. The first one will take as input the title and description combined and feed them to a Convolutional Neural Network (CNN) and the second one will do the same with a Long Short-Term Memory Network (LSTM). Depending on the evaluation of those models on the testing set it will be determined if the information contained in the title and the description of the bug reports is enough to calculate the bug-fix time.

For hypothesis 2 there will be 4 additional models trained. They would be different ensemble neural networks which would take as an input both textual and numerical data. The results from evaluating the models on the test set will then be compared to the 2 neural networks built to prove hypotheses 1 and 3 and if the results are better it would determine the validity of hypothesis 2. Performing those 4 model trainings as both classification and regression problems which would further determine the validity of hypothesis 3.

The experiment conducted to prove hypotheses 4 and 5 would be very similar to the one performed to prove the first three hypotheses. The main difference being that in this case the goal of the neural networks would be to determine the priority of the bug reports. This can be done by performing very little modifications to the previously explained neural network models or the dataset. The main change in the experiment would be to set the goal of the neural network to classify the priority of the bug report. Again training a model solely on the title and description of the bug report would prove hypothesis 4 and training the other four models and comparing the results to the first two would prove hypothesis 5.

3.2 Implementation

The next stage after defining the hypotheses and the design for this experiment is the implementation stage. The implementation stage involved creating an anaconda environment in an amazon web services ES2 instance. The instance uses a GPU processor and was used to increase the computation power and speed up the training of the models.

3.2.1 Data Preparation

After determining the methodology of the project, this part of the report will focus on the implementation details of the project. The initial dataset contains 85,156 bug reports with each report having 11 features, both textual and numerical, which need to be prepared for fitting to a machine learning model. The initial features and their types can be seen in Table 3 below.

Data Column	# of non-null values	Data type
Issue_id	85156 non-null	int64
Priority	85156 non-null	object
Component	85156 non-null	object
Duplicated_issue	14404 non-null	float64
Title	85156 non-null	object
Description	85027 non-null	object
Status	85156 non-null	object
Resolution	85156 non-null	object
Version	85156 non-null	object
Created_time	85156 non-null	object
Resolved_time	85156 non-null	object

Table 3. Columns and data types of the initial dataset.

The first step is to remove the unnecessary columns. Those were determined to be the Issue_id which was used for indexing and has no connection to determining the resolution time or the priority of a bug report and the second one was Duplicated_issue which contains indexes to point towards duplicate reports. The next step is to remove the instances with null values. The only one column containing null values left was the Description one. It contains 129 empty values which were identified and removed from the dataset leaving 85,027 bug reports. Since the main focus of this project is to calculate the resolution time of a bug report it is necessary to filter out the bugs which have not

been resolved. In order to do that the Resolution column was inspected next which contained the following values shown in Table 4 below.

FIXED	42327
DUPLICATE	14398
WONTFIX	9666
WORKSFORME	8414
INVALID	7090
NDUPLICATE	2330
NOT_ECLIPSE	801
MOVED	1

Table 4. Value counts for the Resolution feature.

After keeping only the values marked as fix there are 42327 bug reports left in the dataset for further inspection. Once that has been done the Resolution column was dropped from the dataset as it contains only FIXED values and would not contribute to the neural network model. The next column inspected was the Status column which contained 3 unique values: “RESOLVED”, “VERIFIED”, “CLOSED”, which all indicate that the bug has been fixed and the column is unnecessary so it was removed from the dataset. After removing the unnecessary columns it was time to work on the remaining columns and add new ones.

The first part of that was to change the priority from an object to an integer. The priority column was written in the format P#, which required removing the letter P from before the number and converting the column. From there the additional columns added were Title and Description Length and Positive and Negative Sentiments. The first two were obtained by counting the number of words for each of them but the second one required the use of the external software SentiStrength described in the background section of this report. The way this was done by saving each description into a separate line of a text file which was then uploaded into the SentiStrength software, which output another text file with each description as well as two values. The first one for positive and the second one for negative. The positive sentiment ranged from 1 to 5 and the negative from -1 to -5. After that the text file with the sentiment values was loaded back into Spyder and the two numbers were saved as additional columns to the dataset. The same process was repeated for the Title column as well but the results had very little variance

due to the shortness of the titles. The values for the positive and negative sentiment for the description column are shown in Table 5 below:

+ Sentiment	Value Counts	- Sentiment	Value Counts
1	24986	-1	15605
2	14358	-2	21645
3	2876	-3	4551
4	107	-4	496
5	0	-5	30

Table 5. Value counts for positive and negative sentiment.

The next step in the dataset preparation is to prepare the text columns. There are 4 text columns which need to be transformed: Title, Description, Version and Component. In order to fit them to a neural network the Title and Description columns were combined into one column, the punctuation and numbers were removed from the text as well as stop words and words shorter than 4 characters which could potentially lead to the model not generalizing well over the dataset. The Version and Component column were then prepared for one hot encoding by checking the value counts and removing the instances which contained values less than 30 for Version and a 100 for Component. The initial number of unique values was 43 for Version leaving 40 after removing the values with low value count and for component there were 21 unique values initially leaving 16 after. Once that was done they were one hot encoded and added to the dataset which produced a dataset of 42204 rows and 64 columns for each feature.

Once the text features were prepared the next preparation was to prepare the labels for fitting to a neural network. In order to determine how long it took to resolve a bug report the time of creation was subtracted by the time of resolution for each instance and then converted to a float number of days. Once that was done the Created_time and Resolved_time columns were dropped from the dataset as they were no longer necessary. After the number of days were calculated the next step was to inspect them and look at the distribution of the labels. The values of the labels ranged from less than a day to 4345.94 days with a mean value of 170.65 days. Due to the maximum value being so far from the mean value it was necessary to clean up the labels in order for the trained models to generalise well over the whole dataset. For that purpose the bug reports which had higher resolution time than 365 days were removed and the histogram for the remaining

bug reports can be seen below. At this point there were 36,622 bur reports left in the dataset. Once that was done the labels were scaled between 0 and 1 to further increase the chances of good generalisation of the models.

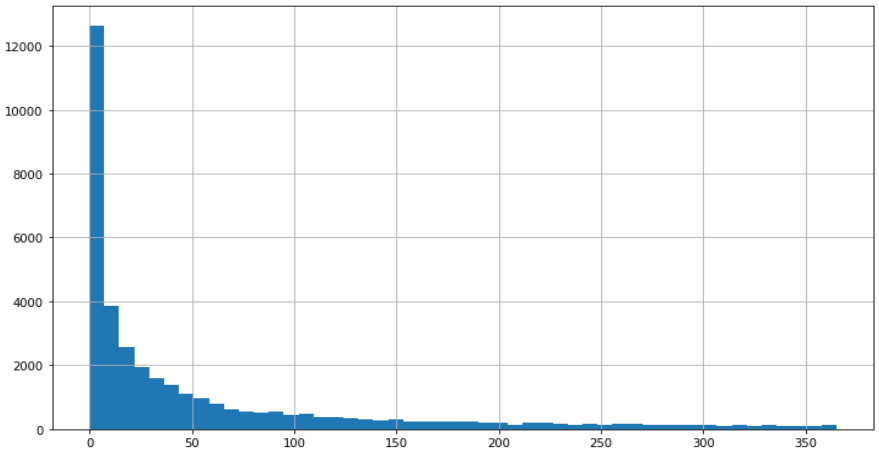

Figure 4. Histogram of the data in the Resolution Time column.

In the case of regression for a resolution time that was the preparation of the labels done. In the case of classification the resolution time was split into 52 equal categories, which equated the number of weeks it has taken to fix a bug report, and then changed to categorical data, in order to use them for softmax, which was explained in the Background section of this report. Additionally when the goal of the neural network was to predict the priority of the bug report, that column was the one converted to categorical instead.

The final steps of the data preparation was to split the dataset and tokenize the textual attributes. For the regression model the data was split randomly into 80% training set and 20% testing set. In the case of classification the data was again split into 80% training and 20% testing set, but this time using Stratified Shuffle Split, which was explained in the Background section, in order to preserve the distribution of the labels. Once the data was split into training and testing set it was necessary to separate textual features, numerical features and labels into 3 different objects for both sets. Once that was done the final step before fitting a model was to tokenize the combined Title/Description of the bug reports to a set length of 512.

This concludes the data preparation and the next process was to find the proper models to answer the hypothesis described earlier in the report.

3.2.2 Models

In order to determine the validity of the hypotheses there were 12 ensembles of neural network configurations in total developed which used the prepared dataset, described in the previous section. 6 of the ensemble models were used to answer the first 3 hypotheses regarding the resolution time of a bug report and the other 6 to answer the remaining 2 hypotheses regarding the priority of the bug reports. The first 6 configurations were both trained as a classification and regression problem in order to answer hypothesis 2. The difference between them is that the regression problem had a final fully connected layer of 1 neuron for the output. All of the ensemble configurations were designed as different combinations between multi-layered perceptron, convolutional neural network and long short-term memory neural network. Those combinations were made using predefined models. In the next part of this report they will be described and then the combinations made between them would be explained.

One of the models is a multi-layered perceptron (MLP), which was explained previously in the background section. The CNN consists of 3 fully connected layers (Dense) with a dropout layer. The first layer has 128 neurons and takes as an input the numerical features from the dataset. It then fully connects to a second layer with 64 neurons, connected to a dropout layer with a rate of 0.4 which is finally fully connect to another layer with 32 neurons. A graph of the model can be seen in Figure 5 below.

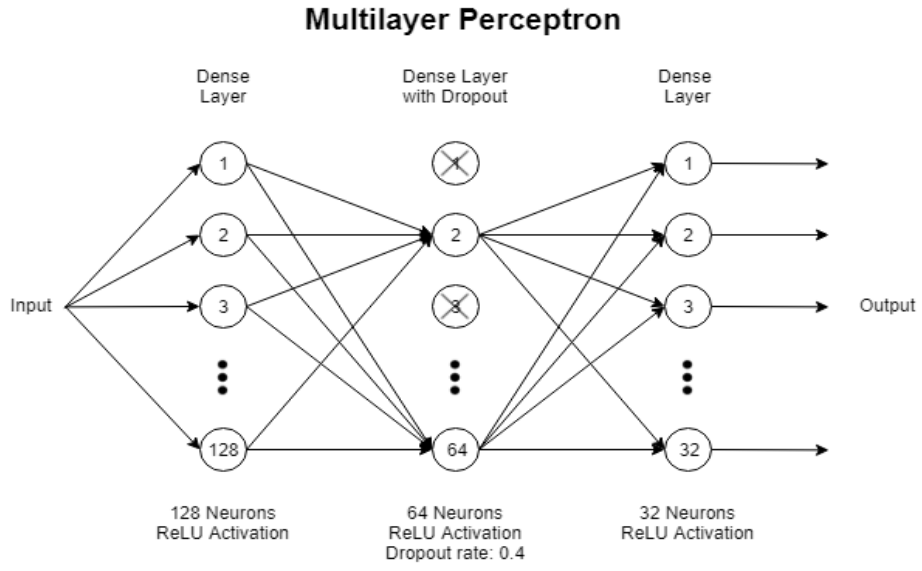

Figure 5. Diagram of the Multi-Layer Perceptron developed for this experiment.

Another model is a convolutional neural network (CNN), which was also briefly described in the background section. The CNN starts with an embedding layer which takes as an input the tokenized textual feature from the prepared dataset and connects to a dropout layer with a rate of 0.2, which is then connected to the first 1-dimension convolutional layer with 128 filters and a kernel size of 6, which then goes through a batch normalization layer and a 1-dimension average pooling layer with a pool size of 5. This is then repeated by going into another dropout layer with a rate of 0.3 and another 1-dimension convolutional layer with 64 filters and a kernel size of 12, as well as an l2 kernel regularizer, in order to reduce possible overfitting. The convolutional layer is connected to another batch normalization layer which goes into a 1-dimensional max pooling layer with a pool size of 3 before the output of it is flattened and goes into a 3rd dropout layer with rate 0.4 which then goes on to connect to two fully connected layers with 64 and 32 neurons respectively. A graph representation of the CNN described can be seen in Figure 6 below

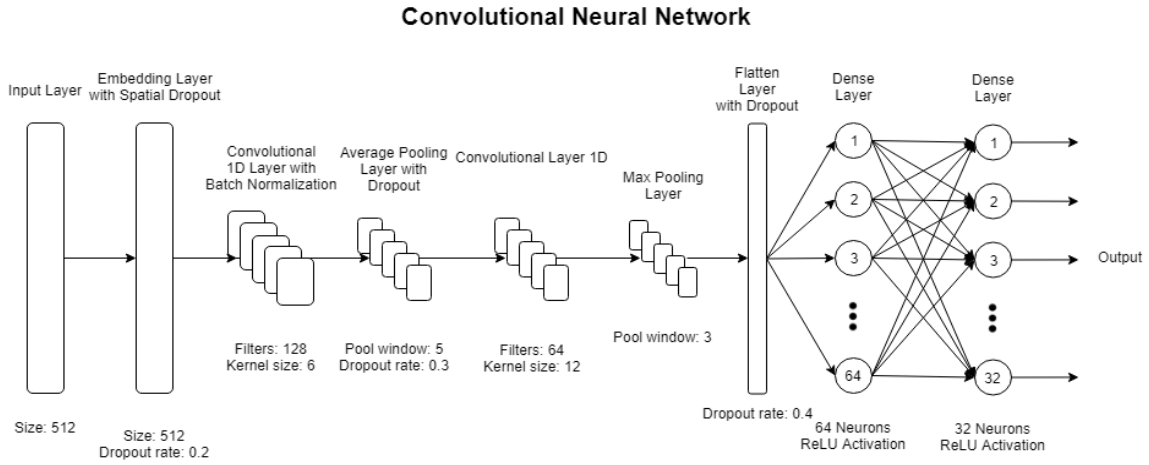

Figure 6. Diagram of the Convolutional Neural Network developed for this experiment.

The final model created was a long short-term memory network (LSTM). This model just like the CNN starts with an embedding layer taking in the textual feature as an input but this time it connects to an LSTM layer with 128 neurons which has a dropout rate of 0.2 and a recurrent dropout rate of 0.2. The output of this layer is connected to 3 fully connected layers with 128, 64 and 32 neurons respectively with a dropout layer with a rate of 0.4 between the first and the second dense layers. A graph of the LSTM model can be seen in Figure 7 below.

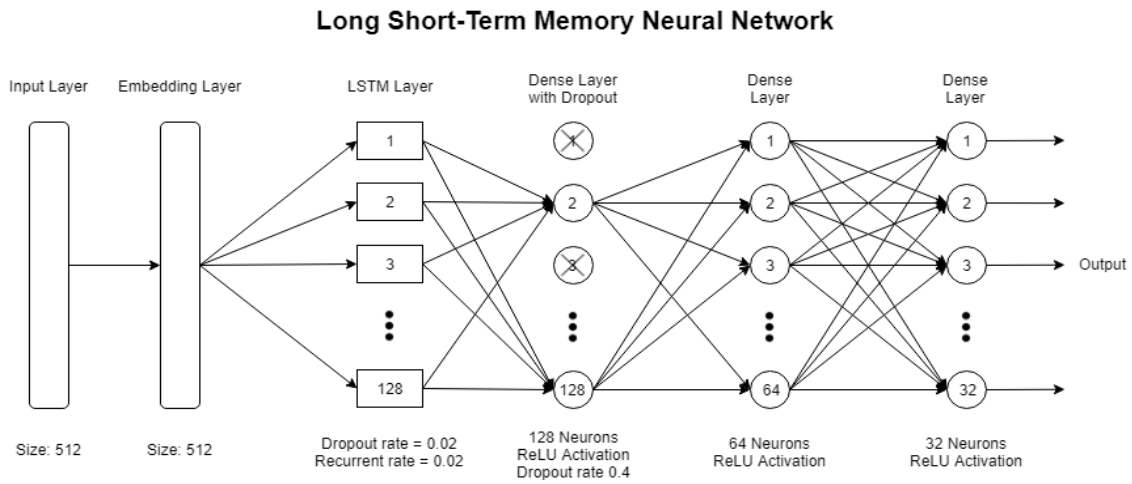

Figure 7. Diagram of the Long Short-Term Memory Network developed for this experiment.

The main 12 ensemble configurations made were a combination of the 3 models explained above and were used to determine the validity of the 5 hypotheses, also explained before. The first configuration was a combination of a CNN and an MLP which would take as an input the text feature and all the numerical features including the one-hot encoded version and component features. After combining the CNN and MLP there were 2 other fully connected layers added with 64 and 16 neurons respectively and finally an output fully connected layer which had, in the case of classification, the same number of neurons as categories and for regression one neuron for the output. The second configuration created was very similar to the first one with the only difference that it did not consider the version and component features thus reducing the number of numerical inputs from 61 to 5. The third configuration was used to answer hypothesis 2 and only involved a CNN, as the one described above, with an additional fully-connected layer of 16 neurons and a final output layer of either 1 neuron, for regression, or the number of categories, for classification.

The next 3 configurations all involved using an LSTM model, where the first one was a combination of LSTM and MLP with an additional dense layer of 16 neurons and a final layer with either 1 neuron or the number of neurons the same as the number of categories. This configuration would take in as an input to the LSTM the text feature and for the MLP the numerical features including the version and component features. The next configuration was the same as the previously described one but without having the version and component features as an input. And the final of the 3 configurations was a

simple LSTM model which would take as an input only the text feature representing a tokenized combination of the title and description of a bug report. Those were the 6 configuration with regards to the resolution time and used to confirm the first 3 hypotheses.

The other 6 configurations were developed to attempt to classify the priority of a bug report. They were created in the same way as the previously described 6 configurations with the only difference that the goal of the neural networks was to classify the priority. Unlike the first 6 configurations those did not have the option to be used as a regression model and the number of categories for classification was 5.

Figure 8 below shows an example of configuration 1 which takes as an input 61 numerical features and 1 textual feature with a size of 512. It also depicts both regression and classification. This Figure is just an example of what a configuration looks like. This concludes explaining the models created to confirm the hypotheses and now this report is going to proceed to presenting the results in the next section and then analyse and discuss them in the following one.

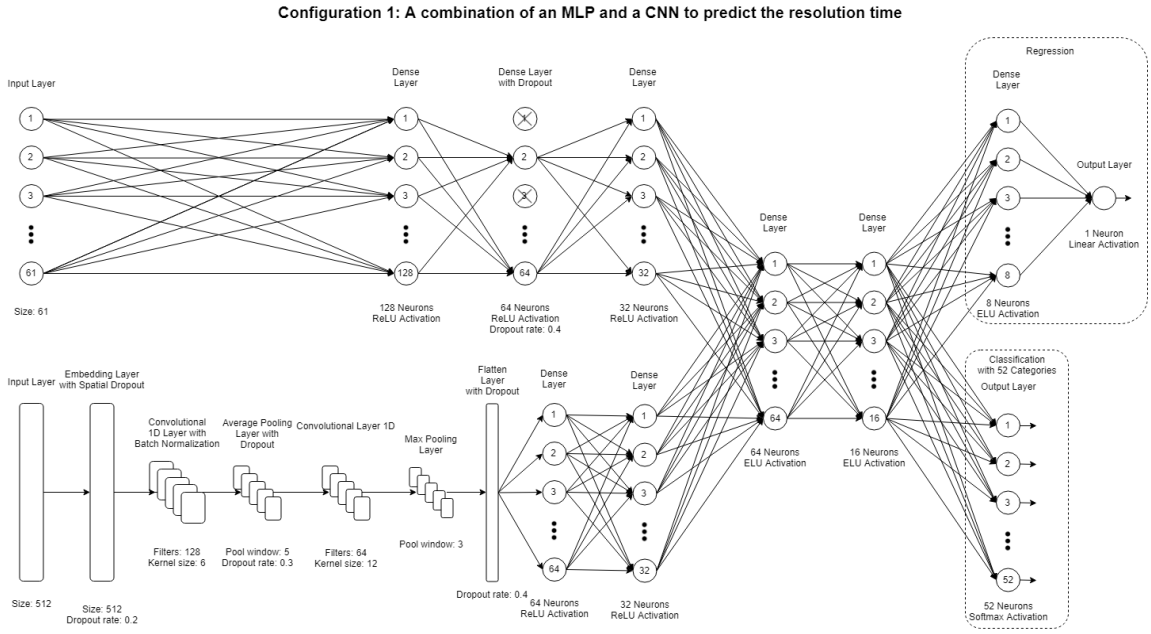

Figure 8. Diagram of a sample Ensemble Neural Network - a combination of an MLP and a CNN.

All of the previously described models have been compared to a baseline metric to assess their performance. In the case of regression problem a linear regression model was used to compare to the neural networks developed, as for classification the

comparison was done to a decision tree classifier. In all of the cases the neural networks performed better than the baseline models.

Chapter 4: Analysis

This chapter would focus on determining the validity of the hypotheses. After carrying out the designed experiments and implementing them in the previous section, the next step would be to run the implementation and collect results. As described in the background section of this report the neural networks have multiple hyperparameters to tune to ensure that the highest accuracy rate is obtained. After running the different configurations described in the implementation section there was a large set of results obtained.

4.1 Results

Due to the large number of models designed, Table 6 was created to describe each model, their target and index them for future reference.

Model Number	Description	Target
1	A combination of CNN and MLP which take as an input both the textual and numerical features (including the One-hot encoded version and component features)	Resolution time
2	A combination of CNN and MLP which take as an input both the textual and numerical features (excluding the version and component features)	Resolution time
3	A CNN which takes as an input the tokenized combination of the title and description features	Resolution time
4	A combination of LSTM and MLP which take as an input both the textual and numerical features (including the version and component features)	Resolution time
5	A combination of LSTM and MLP which take as an input both the textual and numerical features (excluding the version and component features)	Resolution time
6	An LSTM which takes as an input the tokenized combination of the title and description features	Resolution time
7	A combination of CNN and MLP which take as an input both the textual and numerical features (including the One-hot encoded version and component features)	Priority
8	A combination of CNN and MLP which take as an input both the	Priority

	textual and numerical features (excluding the version and component features)	
9	A CNN which takes as an input the tokenized combination of the title and description features	Priority
10	A combination of LSTM and MLP which take as an input both the textual and numerical features (including the version and component features)	Priority
11	A combination of LSTM and MLP which take as an input both the textual and numerical features (excluding the version and component features)	Priority
12	An LSTM which takes as an input the tokenized combination of the title and description features	Priority

Table 6. Indexed short description of the different models created and evaluated.

Having described the models and assigning them a model number these numbers will be used throughout the rest of this section to refer to the corresponding models. All of the configurations described above were trained multiple times in order to try and obtain the best results possible. However, due to the complexity of the ensemble neural networks, it would take a long time and resources unavailable for this project to run all of them more than twice. Table 7 below contains a list of all the model configurations trained with different hyperparameters. The table also includes the learning rate of the neural networks, number of times the model has iterated through the whole training set, size of batches, which represents the number of instances that were supplied to the model at every turn, whether the configuration was a classification or regression as well as testing and training accuracy.

Model Number	Learn Rate	Iterations	Batch size	Classification/ Regression	Training Result	Testing Result
1	0.0005	20	200	Regression	0.0996 RMSE	0.2569 RMSE
1	**0.0001**	**25**	**500**	**Regression**	**0.2471 RMSE**	**0.2528 RMSE**
1	**0.0001**	**15**	**300**	**Classification**	**34.5%**	**29.47%**
1	0.0005	40	400	Classification	87.3%	13.52%
2	**0.0002**	**30**	**100**	**Regression**	**0.0964 RMSE**	**0.2504 RMSE**
2	0.001	35	300	Regression	0.0888 RMSE	0.2559 RMSE
2	0.0005	50	500	Classification	88.5%	13.09%
2	**0.002**	**10**	**150**	**Classification**	**57.63%**	**17.26%**
3	0.0005	20	300	Regression	0.1220 RMSE	0.2547 RMSE
3	**0.002**	**15**	**200**	**Regression**	**0.1002 RMSE**	**0.2530 RMSE**
3	**0.0001**	**25**	**400**	**Classification**	**40.35%**	**18.78%**
3	0.0002	35	100	Classification	86.02%	13.67%
4	**0.0003**	**10**	**100**	**Regression**	**0.2226 RMSE**	**0.2233 RMSE**
4	0.0005	25	400	Regression	0.223 RMSE	0.2237 RMSE

4	**0.0002**	**15**	**300**	**Classification**	**33.42%**	**33.42%**
5	**0.0002**	**30**	**200**	**Regression**	**0.2231 RMSE**	**0.2234 RMSE**
5	0.0001	40	200	Classification	33.45%	33.46%
5	**0.0003**	**10**	**500**	**Classification**	**33.46%**	**33.46%**
6	**0.001**	**12**	**200**	**Regression**	**0.2226 RMSE**	**0.2235 RMSE**
6	0.0005	25	400	Regression	0.2227 RMSE	0.2236 RMSE
6	**0.002**	**12**	**100**	**Classification**	**33.45%**	**33.46%**
7	0.0002	15	150	Classification	95.84%	74.62%
7	**0.003**	**10**	**50**	**Classification**	**94.62%**	**78.48%**
8	0.0005	20	300	Classification	97.15%	72.11%
8	**0.0001**	**12**	**200**	**Classification**	**83.63%**	**80.13%**
9	**0.003**	**20**	**300**	**Classification**	**97.54%**	**76.4%**
9	0.001	30	200	Classification	97.17%	72.63%
10	**0.0005**	**15**	**300**	**Classification**	**85.44%**	**85.43%**
10	0.0008	25	400	Classification	85.49%	85.39%
11	0.0002	18	100	Classification	85.46%	85.46%
11	**0.001**	**10**	**300**	**Classification**	**85.46%**	**85.46%**
12	0.0001	50	200	Classification	85.46%	85.46%
12	**0.0001**	**25**	**100**	**Classification**	**85.49%**	**85.48%**

Table 7. Results for each model configuration.

In the next subsection those results will be explained in detail and the subsection after will focus on further evaluation.

4.2 Discussion

This section will not only contain detailed explanations of the results presented above but also focus on determining the validity of the hypotheses and suggestions for improvement on the results. Even though Table 7 above, contains the results from all the models trained with all the different hyperparameters, this section would only refer to the best results from each model configuration (i.e. the ones in bold). As can be seen in the table above the results for the classification problems is presented in percentages and the results of regression problem is evaluated using RMSE.

For classification it is used a percentage value which represents the percentage of correctly classified instances, thus the accuracy. As for regression - the root mean squared error evaluation (RMSE) represents the root mean squared error difference between the actual values and the predicted values. As such, the closer that value is to 0 the better the predictions are. The results obtained are based on labels ranging from 0 to 1. Another

thing worth mentioning is the difference between the results from the training set and the testing set. That occurs when the model that has been trained is overfitting.

Hypothesis 1

As explained in the hypotheses and experiments sections the results from 2 of the models will be used to determine the validity of hypothesis 1. That is models 3 and 6 which only take as an input the textual features to determine if the resolution time can be predicted based on them alone. From the results for models 3 and 6, as it can be seen from the table 2 above, model 3 achieves better results for the training set compared to the testing set which means that the model is overfitting and would not work well on future unseen data, and for model 6, the training and testing results are quite similar, which would point to the model generalizing well over the full dataset, but the predictions are not very accurate thus refuting hypothesis 1.

Hypothesis 2

As for hypothesis 2 it was necessary to compare the results from the models which have both textual and numerical inputs to the ones that only have textual inputs. The models developed representing both textual and numerical inputs are models: 1, 2, 4 and 5 and as mentioned earlier the ones with textual only inputs are 3 and 6. In this case to support or refute this hypothesis, the results from models 1 and 2 were compared to model 3 and the results from 4 and 5 - to 6. This was done as such because models 1, 2 and 3 involve a CNN and the other models involve an LSTM. The results for the six models described, are available from the results table in section 4.1. Based on the results in the table it can be seen that, for models 1 and 2, the training and testing results have a significant difference between them which shows overfitting just like model 3 and models 4 and 5, having similar results on both the training and testing data, seem to generalize well over the whole data set same as model 6. By comparing the values it can be seen that the models which include numerical data perform slightly better than the textual only ones. Even though the difference between the compared models is not significant it supports the validity of hypothesis 2 but would require further research to prove it with certainty.

Hypothesis 3

Hypothesis 3 tries to show that transforming the prediction of the bug-fix time problem into a classification one, rather than regression, would yield better results. This can be proved by comparing the classification models to the regression models from 1 to 6. This time, it is not necessary to compare all models, but the best model from regression can be compared to the best model from classification. The best classification model in this case is model 5 with average accuracy of 33.45% over the training and testing set and the best regression model is model 4, which gives an average RMSE value of 0.22 for the training and testing set. Even though it is not possible to compare accuracy values with RMSE values, in this case the regression results can be multiplied by 100, which would give the percentage that the predictions will be off and that is on average around 22%. In the case of bug reports fix-time prediction regression that is off by 22% seems preferable to classification which is accurate in only 33.46% of the time. The regression model would be better suited to be used in a guideline for writing bug reports or a tool which supplies the developer with information about the bug reports. Thus refuting hypothesis 3.

Hypothesis 4

Hypotheses 4 and 5 focus on predicting the priority of bug reports and the models trained to prove their validity is models 6 to 12. For hypothesis 4 in order to prove its validity it is necessary to train models which only involve textual features and that is models 9 and 12. As it can be seen from the results table model 9 achieves a significantly higher accuracy for the training set than the testing set and model 12 achieves very similar results over both sets. The results from model 9 show overfitting of the model which means that the model would not perform well on future unseen data, however the results from model 12 generalize almost identically over the whole data set and is able to classify the priority of bug reports correctly in approximately 85.5% of the cases. This means that the trained models perform well in the attempt to classify priority and thus support the validity of hypothesis 4.

Hypothesis 5

Just like hypothesis 2 in order to prove hypothesis 5 it was necessary to compare the 4 trained models on both textual and numerical data with the 2 trained models on

textual data input only. The models trained on the mixed data of text and numeric features are: 7, 8, 10 and 11 and the ones only on text data are models 9 and 12 which were mentioned in the previous hypothesis already. Same as hypothesis 2, the comparison will be performed on models 7 and 8 to model 9 and 10 and 11 to 12 since they use the same deep learning neural networks. The results for models 7, 8, 10 and 11, as can be seen from the table in section 4.1, show that the accuracy of model 7 over the training data is higher than the testing data accuracy, model 8, even though there is still a difference between training and testing accuracy, it is not significant, and models 10 and 11 show similar results for the training and testing data. From the comparison of models 7 and 8 to model 9 it can be seen that both models 7 and 9 are overfitting since the testing accuracy is significantly below the training accuracy, but model 8 generalizes well which supports the hypothesis, and as for models 10, 11 and 12 they all generalize well over the whole dataset, however model 12 which takes only textual data as input performs slightly better than the other 2 LSTM models. From the CNN models one of them, with both numerical and textual features, performs significantly better than the other 2 and for the LSTM models the only text input one performs slightly better than the other 2. Thus it can be concluded that even though it would require further research hypothesis 5 is supported.

This finalises the discussion chapter which shows that hypotheses 2, 4 and 5 are supported by the experiment conducted and hypotheses 1 and 3 are refuted.

4.3 Further Evaluation and Optimization

Even though all the hypotheses were answered using the already trained models, in this subsection some further evaluation will be performed on optimizing the best models for resolution time regression and priority classification.

Further Evaluation Methodology

For the resolution time, the data will be split into the 4 main percentiles and based on their values a decision will be made to establish a boundary between slow, medium and fast resolution times. Once that is done predictions will be made using the best model for resolution time regression to determine the RMSE for the predictions of each category.

The reason that is done is that, for example the best resolution time regression model is model 4 with an RMSE value of 0.2233 RMSE on the testing data with a scaled range of 0 to 1. This means that the results on average will be off by 22.33%. Based on the unscaled data used the resolution time ranges from 0 (less than a day) to 365 (1 year). The idea is that a 22.33% deviation is a lot more significant on bug reports which have been fixed for a day than ones fixed for 11 months, for example. Based on that if the results are unsatisfying, the model will be further optimized by training it on only a sample of the previously trained data, which would contain a certain percentage of instances from each quadrant.

For the priority classification, a deeper look will be taken on the number of instances for each category for both the training and test set. First a comparison will be made between the predicted values and the actual values for each category and then a detailed confusion matrix will be created and the precision and recall for the model will be calculated. Based on the research done before (See section 2.1.6), it is known that bug reports with priority category 1 are crucial to be identified, since they need to be fixed as soon as possible, while the bugs with priority 5 may not be fixed at all. In this case if a bug of priority 1 is incorrectly classified as a bug of priority 3 could lead to some serious problems, while if a bug of priority 5 is classified as 3, may only lead to a bug of low importance to be fixed sooner than necessary. Thus, based on the results from the confusion matrix, the best model, which is model 12 with testing accuracy of 85.48%, will be retrained on a sample of the previously trained data, which would contain a certain percentage of instances from each category in order to guide the model towards the more important ones if necessary.

Evaluation and Optimization

The first step in the implementation of the resolution time regression problem evaluation is to inspect the label values in order to make an informed decision. Based on the test set the data distribution is that in the 25^{th} percentile the values are below 3.96 days, in the 50^{th} percentile it is under 19.83 days and the 75^{th} percentile is below 71.87 days. Another value to take note of when making the split decision is the mean of the labels which is 56.73 days. Based on those values it was decided that the fast resolved bug reports will be considered the ones below 19.83 days, the medium resolution time will be

between 19.83 and 71.87 days and the slow will be above 71.87 days to resolve. Once the test data was split it resulted into 3663 fast resolved, 1832 medium resolved and 1830 slow resolved bug reports. After that each of those splits of data was evaluated on model 4 and the results can be seen in Table 8 below.

Data	# of Instances	Result
Test Set	7325	0.2233 RMSE
Fast Resolved	3663	0.1343 RMSE
Medium Resolved	1830	0.0594 RMSE
Slow Resolved	1832	0.3999 RMSE

Table 8. Results from the evaluation of the best resolution time model on the test set split.

As the table above shows the predictions closest to the actual value come from the bug reports with medium resolution time and the worst results was achieved by the slowly resolved bugs. In order to try and train a better model, using this information the next step was to look at the training data labels. Using the same values as separators as before the training data showed that from 29,297 training instances, 14,493 belonged to the group with fast resolution time, 7,362 were part of the medium and 7442 the slow resolution time. The fact that the bug reports with fast resolution time were almost twice as much as the slow or the medium, shows that they have the most influence when training the model. For that reason in order to even out the numbers there were 2 options. One is to add more data and the other one is to reduce the data. Adding more data would increase the cost and training time of the models, so the option to reduce the data was chosen. Doing that, the fast resolution time instances were reduced to 7402. After the data was evened out, the model was retrained with the same hyperparameters and the results obtained were recorded in Table 9 below.

Data	# of Instances	Result
Test Set	7325	0.2322 RMSE
Fast Resolved	3663	0.206 RMSE
Medium Resolved	1830	0.1219 RMSE
Slow Resolved	1832	0.3405 RMSE

Table 9. Results from the evaluation of the optimized resolution time model on the test set split.

Comparing the results from the model being trained with both datasets, shows that even though the model with equal distribution of data, performs slightly worse than the original on the full testing set, the results from the separated test set demonstrate

that the prediction of bugs with slow resolution is more accurate at the expense of the prediction error for both fast and medium resolution times. Even though reducing the data did not lead to a better model in general, it still goes to show that modifying the distribution of labels in the dataset can guide the neural network to focus on particular group of instances. Figure 9 below can be used to compare the training loss of both models.

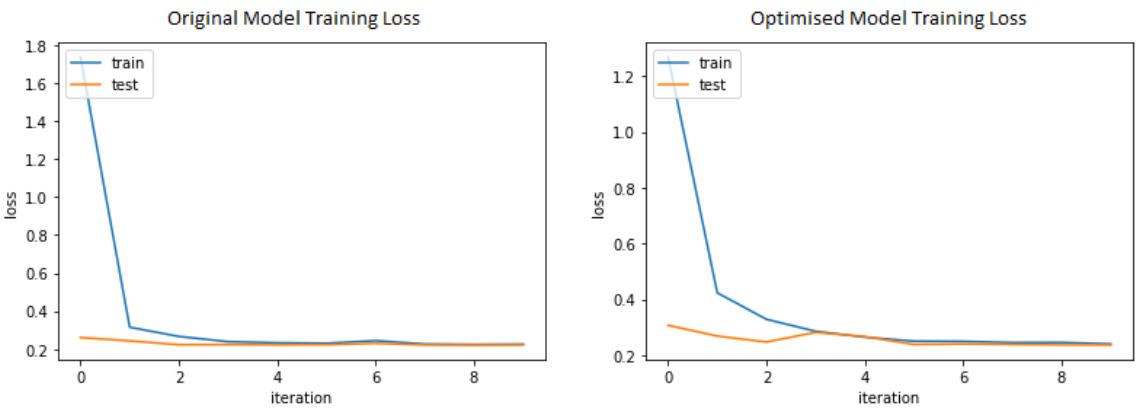

Figure 9. Training loss of both models.

The next part was to perform the evaluation of the best priority classification model. It started with examining the number of values for each category for both the training and testing set, which is shown in Table 10 below.

Priority	Train Set	Test Set
1	1208	302
2	2673	668
3	25113	6279
4	238	60
5	153	38

Table 10. Value counts for the Priority column.

Based on the values in the table, the bug reports with priority 3 are significantly more than the rest of them. More specifically for the test set based on 7,347 instance there are 6,279 with priority 3, which when calculated as percentage they represent 85.46% of the test set, which as it happens is very similar to the accuracy achieved for the best model trained to classify priorities. The testing accuracy is 85.48% for that model. After that it was time to look at the actual predictions, which are presented in Table 11 in the form of a confusion matrix.

Actual	Predicted					Precision
	1	**2**	**3**	**4**	**5**	
1	**0**	0	302	0	0	0%
2	0	**1**	667	0	0	0.15%
3	0	0	**6279**	0	0	100%
4	0	0	60	**0**	0	0%
5	0	0	38	0	**0**	0%
Recall	0%	100%	85.5%	0%	0%	

Table 11. Confusion matrix for the predictions from the test set.

From the confusion matrix it can be seen that the model trained, due to the very large number of instances with priority value 3, predicts every instance as that same class. Based on the values in Table 11, the calculated F1 score is 0.185. This would mean that the model is unreliable and would need some optimization. In order to optimize the classifier, just like for the resolution time regression problem, there are again 2 options for improvement, which is to add more data or to reduce the data in order to even it out. Again the chosen option was to reduce the data, due to the lack of time and resources and to avoid very high training times. Based on the distribution of the data, it was decided that only 10% of the instances classified as priority 3, would be included in training the optimized model. That would mean that the dataset will be reduced to 6,783 instance, with only 2,511 priority class 3 bug reports. Even though that is a big reduction and the model would not perform as well as before, the number of correctly classified instances with priority 1 and 2 should increase. Again the same model was retrained with the reduced data, although due to the reduced size of the dataset model 11 was also retrained on the reduced training set and the number of iterations was increased to 100. However a callback method called EarlyStopping was added to both models to stop the training if the validation accuracy does not increase over 10 iterations. The original accuracy results from models 11 and 12 as well as the accuracy results from training on the reduced dataset for both models can be seen in Table 12 below.

Model Number	Learn Rate	Iterations	Batch size	Classification/ Regression	Training Result	Testing Result
11	0.001	10	300	Classification	85.46%	85.46%
12	0.0001	25	100	Classification	85.49%	85.48%
11	0.0001	100 (23)	100	Classification	26.6%	26.94%
12	0.0001	100 (11)	100	Classification	4.17%	4.14%

Table 12. Results from the evaluation of the models on the original and reduced datasets.

Based on the accuracy results from training the models on the dataset, it can be concluded that the models did not have enough data to converge to a decent model. Another reason to believe that is that the new models were both set to a 100 iterations and stopped on 23 for model 11 and 11 for model 12, which would show that there was no improvement in the validation accuracy. In Tables 13 and 14 and Figures 10 and 11 are also shown the confusion matrices and the training data plots for models 11 and 12 respectively.

Actual	**Predicted**					**Precision**
	1	**2**	**3**	**4**	**5**	
1	**0**	256	46	0	0	0%
2	0	**554**	114	0	0	9.63%
3	0	4854	**1425**	0	0	89.29%
4	0	52	8	**0**	0	0%
5	0	32	3	0	**0**	0%
Recall	0%	82.9%	22.7%	0%	0%	

Table 13. Confusion matrix for the predictions from model 11 on the test set.

Actual	**Predicted**					**Precision**
	1	**2**	**3**	**4**	**5**	
1	**301**	0	1	0	0	4.1%
2	665	**0**	1	1	1	0%
3	6265	2	**3**	8	1	60%
4	60	0	0	**0**	0	0%
5	38	0	0	0	**0**	0%
Recall	99.7%	0%	0.04%	0%	0%	

Table 14. Confusion matrix for the predictions from model 12 on the test set.

Based on the 2 tables above, the models trained on reduced data have indeed performed worse. However the fact that most of the predictions for model 11 are for category priority 2 and 3 further confirms the theory that the data is not enough to train a good model on it. Which is also supported by the predictions from model 12, which have all predicted priority 1.

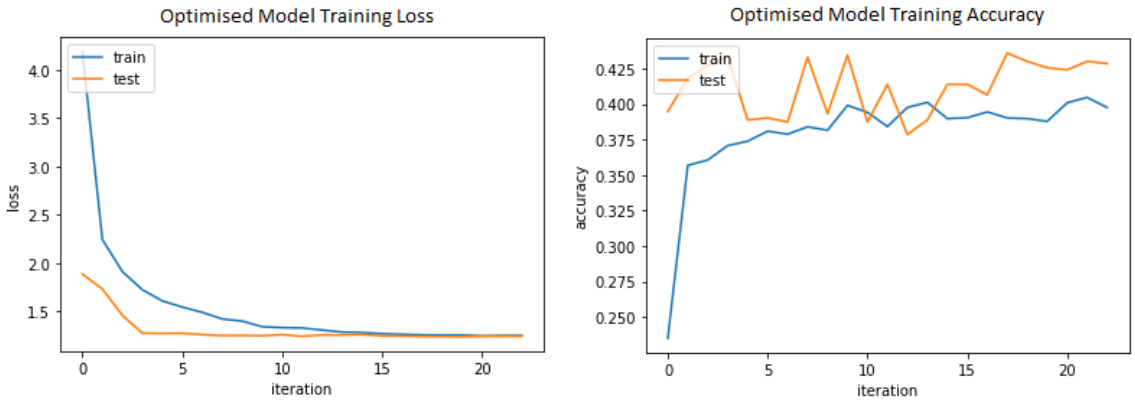

Figure 10. Model 11 training accuracy and loss.

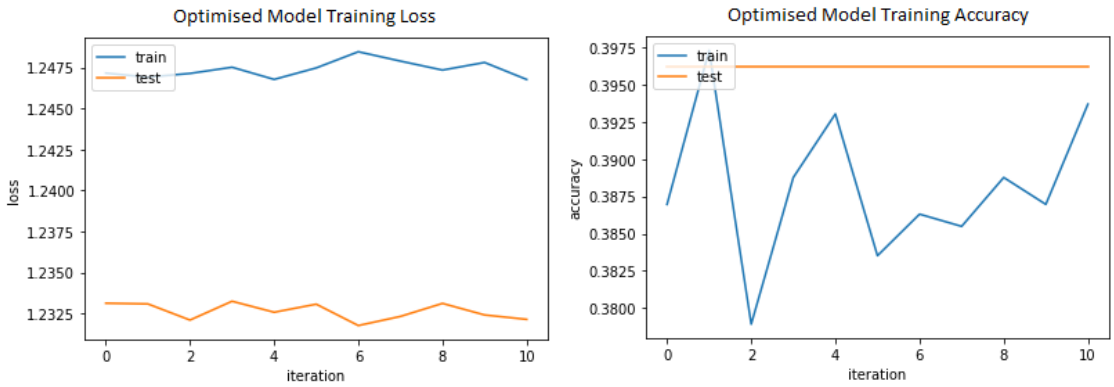

Figure 11. Model 12 training accuracy and loss.

Figure 10 shows model 11's accuracy for both the train and test data fluctuating through iterations, while the accuracy for model 12 on the test set does not improve nor decline, which could mean that the model is not complex enough to find the patterns in the text data. Another metric computed to further evaluate the newly trained models is the F1 metric, which for model 11 shows a value of 0.1069 and for model 12 shows a value of 0.0159. Compared to the F1 value of 0.1849 for the original model 12, only further solidifies the conclusion that the results from training on a reduced dataset are not satisfactory.

Chapter 5: Future Work and Usage

This section will focus on providing suggestions for future work which can be done to provide better results as well as describe some usages for the models built.

5.1 Future Work

There are 3 main ways that could possibly improve the results. One would be to add more data, the other one to add more features and the third one is to explore other types of neural networks. In order to reduce the RMSE for regression or increase the accuracy percentage for classification a bigger dataset could be used. The current one has 85,156 instances which after cleaning and splitting the data leaves 29,297 instances to train the 12 original models on. Increasing the number of instances could possibly improve the models and provide better results. Another way to improve the results would be to add extra features. For example developing a method to recognize different kinds of text in the description, such as stack traces and code. Additionally, some information could be added to each bug report regarding the developer that has resolved the problem. Since the main goal of this project is to predict the bug-fix time, and developers have a lot of influence regarding the time it is going to take to resolve a bug. Increasing the complexity of the neural networks (for example, adding more convolutional layers to a CNN, or more LSTM layers to an LSTM neural network) or implementing a different model (such as Gated Recurrent Unit or Hopfield Network), could possibly improve the results.

Unfortunately working with a bigger dataset or a more complex network would require much greater resources, such as more time and computational power, which are unavailable for this project.

5.2 Usage

Using the best model trained for prediction the resolution time of a bug report can be helpful for both developers and reporters. The best model in this case would be the model which is a combination of an LSTM and an MLP and takes as an input both text and numerical features including version and component. That model achieved a result of 0.2233 RMSE on the testing set. This means that the predictions would be off by 22.33% on average. Developing a system that would predict the resolution time of a bug report together with the priority could really help developers manage their time better and resolve more bug reports. On the contrary if a reporter knows on average how long it

would take to fix the bug they are reporting, could lead them to providing more information in order to improve the resolution time.

With regards to priority classification, as it can be seen from the results section the highest accuracy is 85.48% accuracy on the testing set, which means that the prediction is correct in nearly 85.5% of the cases. With accuracy that high, this model could be used to develop tools which automatically assign the priority of new bug reports, which would make the job of the developers working on those bug reports easier.

Chapter 6: Conclusion

The goal of this project was to examine the behaviour of neural networks when applied to bug report analysis. In particular to try and predict the resolution time and the priority of bug reports, as this has not yet been done using deep neural networks. This dissertation was also done to analyze the significance of adding numerical features to textual features by combining different kinds of neural networks.

After conducting multiple experiments, using different kinds of neural networks and number of features it was concluded that adding numerical features to textual features can increase the accuracy of the predictions for both the resolution time prediction and the priority classification. It was also verified that using the text features from a bug report alone is enough to classify the priority in about 85% of the cases.

Even though the assumptions that the prediction of the bug-fix time can be accurately obtained using only the textual features were refuted, there are some suggestions that those hypotheses could also be supported given that additional data was obtained and trained on more complex models, which are inaccessible, given the available resources.

The overall experience from this experiment would also suggest that neural networks are very useful when working with textual data and to combine different types of features. Running the experiments also suggests that adding the positive and negative sentiment of a bug report as a feature may make a difference even if it is a small one.

Other observations from this dissertation are that in the case of textual processing, RNNs like the LSTM used generalize better than CNNs.

Bibliography

Abdelmoez, W., Kholief, M. and Elsalmy, F. (2012). Bug fix-time prediction model using naive Bayes classifier. *2012 22nd International Conference on Computer Theory and Applications (ICCTA)*.

Ahsan, S., Ferzund, J. and Wotawa, F. (2009). Automatic Software Bug Triage System (BTS) Based on Latent Semantic Indexing and Support Vector Machine. *2009 Fourth International Conference on Software Engineering Advances*.

Alenezi, M. and Banitaan, S. (2013). Bug Reports Prioritization: Which Features and Classifier to Use? *2013 12th International Conference on Machine Learning and Applications*.

Alipour, A., Hindle, A. and Stroulia, E. (2013). A contextual approach towards more accurate duplicate bug report detection. *2013 10th Working Conference on Mining Software Repositories (MSR)*.

Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R. and Zimmermann, T. (2007). Quality of bug reports in Eclipse.

Budhiraja, A., Dutta, K., Reddy, R. and Shrivastava, M. (2018). DWEN: deep word embedding network for duplicate bug report detection in software repositories. *Proceedings of the 40th International Conference on Software Engineering Companion Proceeedings - ICSE '18*.

Ca.com. (2019). *Company - CA Technologies - EMEA*. [online] Available at: https://www.ca.com/gb/company.html [Accessed 15 Aug. 2019].

Chen, X., Qui, X., Zhu, C., Liu, P. and Huang, X. (2015). Long short-term memory neural networks for chinese word segmentation. *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp.1197-1206.

Conneau, A., Schwenk, H., Le Cun, Y. and Barreau, L. (2016). Very deep convolutional networks for natural language processing.

Dallmeier, V. and Zimmermann, T. (2007). Extraction of bug localization benchmarks from history. *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*.

Deshmukh, J., M, A., Podder, S., Sengupta, S. and Dubash, N. (2017). Towards Accurate Duplicate Bug Retrieval Using Deep Learning Techniques. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

dos Santos, C. and Gatti, M. (2014). Deep convolutional neural networks for sentiment analysis of short texts. *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pp.69-78.

Garbade, M. (2018). *A Simple Introduction to Natural Language Processing*. [online] Medium. Available at: https://becominghuman.ai/a-simple-introduction-to-natural-language-processing-ea66a1747b32 [Accessed 16 Aug. 2019].

Gardner, M. and Dorling, S. (1998). Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric Environment*, 32(14-15), pp.2627-2636.

GeeksforGeeks. (n.d.). *Deep Learning | Introduction to Long Short Term Memory - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/ [Accessed 16 Aug. 2019].

Géron, A. (2017). *Hands-on machine learning with Scikit-Learn and TensorFlow*. 1st ed. O'Reilly Media, Inc., p.3.

Giger, E., Pinzger, M. and Gall, H. (2010). Predicting the fix time of bugs. *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering - RSSE '10*.

Gottschalk, L. (1969). *The measurement of psychological states through the content analysis of verbal behaviour*.

Goyal, A. and Sardana, N. (2017). *NRFixer: Sentiment Based Model for Predicting the Fixability of Non-Reproducible Bugs*. [online] Available at: https://www.semanticscholar.org/paper/NRFixer%3A-Sentiment-Based-Model-for-Predicting-the-Goyal-Sardana/6d0ab0406b43deb8c6256a1a7fb0e93535cfb46e [Accessed 15 Aug. 2019].

Hiew, L. (2006). *Assisted Detection of Duplicate Bug Reports*. MSc. The University of British Columbia.

Hooimeijer, P. and Weimer, W. (2007). Modeling bug report quality. *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*.

Hornik, K., Stinchcombe, M. and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5), pp.359-366.

Hu, H., Zhang, H., Xuan, J. and Sun, W. (2014). Effective Bug Triage Based on Historical Bug-Fix Information. *2014 IEEE 25th International Symposium on Software Reliability Engineering*.

Islam, R. and Zibran, M. (2018). Sentiment analysis of software bug related commit messages.

Jeong, G., Kim, S. and Zimmermann, T. (2009). Improving bug triage with bug tossing graphs. *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - ESEC/FSE '09*.

Jifeng, X., He, J., Zhilei, R., Jun, Y. and Zhongxuan, L. (2017). Automatic Bug Triage using Semi-Supervised Text Classification.

Jin-woo, P., Mu-Woong, L., Jinhan, K., Seung-won, H. and Sunghun, K. (2011). CosTriage: A Cost-Aware Triage Algorithm for Bug Reporting Systems.

Johnson, R. and Zhang, T. (2015). Semi-supervised convolutional neural networks for text categorization via region embedding. *Advances in neural information processing systems*, pp.919-927.

Kanwal, J. and Maqbool, O. (2012). Bug Prioritization to Facilitate Bug Report Triage. *Journal of Computer Science and Technology*, 27(2), pp.397-412.

Khan, L. (2019). *Sentiment Analysis with SentiStrength - Laeeq Khan, Ph.D.*. [online] Laeeq Khan, Ph.D. Available at: http://professorkhan.com/sentiment-analysis-with-sentistrength/ [Accessed 16 Aug. 2019].

Kim, Y. (2014). Convolutional neural networks for sentence classification.

Lai, S., Xu, L., Liu, K. and Zhao, J. (2015). Recurrent convolutional neural networks for text classification. *Twenty-ninth AAAI conference on artificial intelligence*.

Lam, A., Nguyen, A., Nguyen, H. and Nguyen, T. (2017). Bug Localization with Combination of Deep Learning and Information Retrieval. *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*.

Liu, C., Yan, X., Fei, L., Han, J. and Midkiff, S. (2005). SOBER: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 30(5), p.286.

Logpai (2019). *BugRepo*. [online] GitHub. Available at: https://github.com/logpai/bugrepo [Accessed 16 Aug. 2019].

Malhotra, R., Aggarwal, S., Girdhar, R. and Chugh, R. (2018). Bug localization in software using NSGA-II. *2018 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*.

Manning, C., Raghavan, P. and Schütze, H. (2018). *Introduction to information retrieval*. Cambridge: Cambridge University Press, pp.32 - 36.

MonkeyLearn. (2019). *Sentiment Analysis: Nearly Everything You Need to Know | MonkeyLearn*. [online] Available at: https://monkeylearn.com/sentiment-analysis/ [Accessed 16 Aug. 2019].

Murphy, G. and Cubranic, D. (2004). Automatic bug triage using text categorization.

Nguyen, A., Nguyen, T., Nguyen, T., Lo, D. and Sun, C. (2012). Duplicate bug report detection with a combination of information retrieval and topic modeling. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*.

Ren, F., Zhou, D., Liu, Z., Li, Y., Zhao, R., Liu, Y. and Liang, X. (2018). Neural relation classification with text descriptions. *Proceedings of the 27th International Conference on Computational Linguistics*, pp.1167-1177.

Rouse, M. (n.d.). *What is tokenization?*. [online] SearchSecurity. Available at: https://searchsecurity.techtarget.com/definition/tokenization [Accessed 16 Aug. 2019].

Saha, R., Lease, M., Khurshid, S. and Perry, D. (2013). Improving bug localization using structured information retrieval. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Sas.com. (2019). *Machine Learning: What it is and why it matters*. [online] Available at: https://www.sas.com/en_gb/insights/analytics/machine-learning.html [Accessed 16 Aug. 2019].

Sentistrength.wlv.ac.uk. (2019). *SentiStrength - sentiment strength detection in short texts - sentiment analysis, opinion mining*. [online] Available at: http://sentistrength.wlv.ac.uk/ [Accessed 16 Aug. 2019].

Sharma, M., Bedi, P., Chaturvedi, K. and Singh, V. (2012). Predicting the priority of a reported bug using machine learning techniques and cross project validation. *2012 12th International Conference on Intelligent Systems Design and Applications (ISDA)*.

Shuttleworth, M. and Wilson, L. (n.d.). *Research Hypothesis*. [online] Explorable.com. Available at: https://explorable.com/research-hypothesis [Accessed 16 Aug. 2019].

Stone, P., Dunphy, D. and Smith, M. (1966). The General Inquirer: A computer approach to content analysis. *MIT Press*.

Sun, C., Lo, D., Khoo, S. and Jiang, J. (2011). Towards more accurate retrieval of duplicate bug reports. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*.

Sun, C., Lo, D., Wang, X., Jiang, J. and Khoo, S. (2010). A discriminative model approach for accurate duplicate bug report retrieval. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*.

Tai, K., Socher, R. and Manning, C. (2015). Improved semantic representations from tree-structured long short-term memory networks.

Techopedia.com. (2019). *What is Software Bug? - Definition from Techopedia*. [online] Available at: https://www.techopedia.com/definition/24864/software-bug- [Accessed 15 Aug. 2019].

Techopedia.com. (n.d.). *What is Tokenization? - Definition from Techopedia*. [online] Available at: https://www.techopedia.com/definition/13698/tokenization [Accessed 16 Aug. 2019].

Tian, Y., Lo, D. and Sun, C. (2013). DRONE: Predicting Priority of Reported Bugs by Multi-factor Analysis. *2013 IEEE International Conference on Software Maintenance*.

Volcani, Y. and Fogel, D. (2001). *System and method for determining and controlling the impact of text*. US7136877B2.

Wang, X., Liu, Y., Sun, C., Wang, B. and Wang, X. (2015). Predicting polarities of tweets by composing word embeddings with long short-term memory. *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, 1, pp.1343-1353.

Wang, X., Zhang, L., Xie, T., Anvik, J. and Sun, J. (2008). An approach to detecting duplicate bug reports using natural language and execution information. *Proceedings of the 13th international conference on Software engineering - ICSE '08*.

Weiss, C., Premraj, R., Zimmermann, T. and Zeller, A. (2007). How Long Will It Take to Fix This Bug?. *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*.

Xu, J., Chen, D., Qui, X. and Huang, X. (2016). Cached long short-term memory neural networks for document-level sentiment classification.

Xuan, J., Jiang, H., Hu, Y., Ren, Z., Zou, W., Luo, Z. and Wu, X. (2015). Towards Effective Bug Triage with Software Data Reduction Techniques. *IEEE Transactions on Knowledge and Data Engineering*, 27(1), pp.264-280.

Zeng, H. and Rine, D. (2004). Estimation of software defects fix effort using neural networks. *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*.

Zhang, F., Khomh, F., Zou, Y. and Hassan, A. (2012). An Empirical Study on Factors Impacting Bug Fixing Time. *2012 19th Working Conference on Reverse Engineering*.

Zhang, H., Gong, L. and Versteeg, S. (2013). Predicting Bug-Fixing Time: An Empirical Study of Commercial Software Projects.

Zhang, J., Wang, X., Hao, D., Xie, B., Zhang, L. and Mei, H. (2015). *A survey on bug-report analysis*.

Zhang, S., Zheng, D., Hu, X. and Yang, M. (2015). Bidirectional long short-term memory networks for relation classification. *Proceedings of the 29th Pacific Asia conference on language, information and computation*, pp.73-78.

Zhou, J., Zhang, H. and Lo, D. (2012). Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. *2012 34th International Conference on Software Engineering (ICSE)*.

Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schroter, A. and Weiss, C. (2008). What Makes a Good Bug Report?. *IEEE Transactions on Software Engineering*, 36(5), pp.618-643.

Zou, W., Hu, Y., Xuan, J. and Jiang, H. (2011). Towards Training Set Reduction for Bug Triage. *2011 IEEE 35th Annual Computer Software and Applications Conference*.