UNIVERSITY OF GHANA

COLLEGE OF BASIC AND APPLIED SCIENCES

MULTIPLE BUG DETECTION AND EFFORT ESTIMATION FRAMEWORK FOR OPEN-SOURCE PROJECTS

BY

ANAS HARA

(10804091)

A THESIS SUBMITTED TO THE SCHOOL OF GRADUATE STUDIES IN PARTIAL FULFILLMENT OF THE AWARD OF DEGREE OF MASTER OF PHILOSOPHY IN COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

SEPTEMBER, 2021

DECLARATION

I Anas Hara, author of this thesis, do now declare that the work presented in this thesis/dissertation; "**Multiple Bug Detection and Effort Estimation Framework for Open-source Projects**" is my work, produced from research undertaken at the Department of Computer Science, University of Ghana, Legon under the supervision of my thesis supervisors. Other people's work has been duly acknowledged and cited.

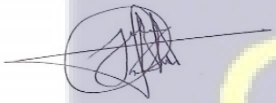
STUDENT

Name: Hara Anas

Signature:  Date: September 20, 2021

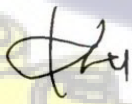
SUPERVISOR

Name: Dr. Solomon Mensah

Signature:  Date: September 22, 2021

CO-SUPERVISOR

Name: Dr. Ebenezer Owusu

Signature:  Date: September 23, 2021

ABSTRACT

Bug reports are essential in the development and maintenance of software. Bug tracking systems allow testers to submit bug reports which allow for report analysis and assignment of reports to fixers to address them. A given bug *x* is described as *multiple bug* when it is reported by more than two bug reporters. It is described as a *duplicate bug* when it was reported by two reporters. In a given pool of bug reports from a tracking system, estimating the effort required to identify *multiple bug* is a challenge, and hence the need to conduct this study. Thus, a plausible solution based on an effort estimation framework to detect *multiple bugs* will reduce the effort software testers spend in analyzing bug reports and also improve software reliability and productivity. Although several studies are attempting to solve the problem, there is the need to introduce an effort estimation framework to detect *multiple bugs* in software projects, specifically open-source projects. However, the following constraints exist when detecting *multiple bugs* in open-source projects: - (1) a large number of existing bug reports, and (2) much effort is required when detecting and analyzing *multiple bug* reports. This study seeks to develop a framework to detect *multiple bugs* and estimate the effort required in identifying such bugs in open-source projects. This study implements the *bugdetector* tool, which uses bug information and code features to find similar bugs. It will first extract features from bug information in a bug tracking system, next it locates bug methods in source code and extracts bug method code features. It calculates similarities between each overridden and overload method, and finally, it determines which method may cause potentially related or similar bugs. Empirical analysis was conducted on bug reports from two open-source projects, namely Mozilla Firefox and Eclipse. Thus, empirical analysis was conducted on the extracted bug reports by the *bugdetector* tool. The analysis was conducted using Deep learning algorithms (LSTM, Bidirectional LSTM and CNN) and conventional machine learning algorithms (SVM and Random Forest). Accuracy, Precision, Recall, and F1-score metrics were used to evaluate

the models' performance. Estimating the required effort for identifying *multiple bugs* was done using a proposed effort estimation metric. Empirical result shows that the deep learning method, namely the Bidirectional LSTM algorithm yielded improved performance for *multiple bug* detection across the two-studied datasets. Thus, for Mozilla Firefox, the Bidirectional LSTM yielded the best performance accuracy (71.09%), precision (68.30%), and recall (45.7%). For Eclipse, Bidirectional LSTM dominated the best performance about accuracy (82.6%) and F1-score (50.9%). The effort required for detecting multiple bugs on average ranges from 1255.7 to 1383.2 days for the studied Eclipse bug repository, and 1049.8 to 1139.2 days for the studied Mozilla Firefox bug repository. The study concluded that the deep learning method has a better tendency in detecting multiple bugs in open-source projects as compared to the conventional machine learning approach. An effort estimation metric is introduced to compute the effort required to detect *multiple bugs* in open-source projects. This will assist software testers/fixers to differentiate between severity levels of the detected bugs based on the respective efforts computed.

Keywords: Duplicate bugs, Effort estimation, Bug detection, Deep learning, Open-source projects.

DEDICATION

To my family especially my mum (Mrs. Salamatu Hara).

ACKNOWLEDGEMENT

I want to show my sincere appreciation to my supervisors Dr Solomon Mensah and Dr Ebenezer Owusu for the care and support given throughout this study. Also, my gratitude to my lovely family, my brothers and friends that cheered me on to this end. May God bless you all.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AST	Abstract syntax trees
BFC	Base functional component
EA	Exact-accuracy
GloVe	Global Vectors for Word Representation
GUI	Graphical user interface
IR	Information retrieval
ML	Machine learning
MMRE	Mean magnitude of relative error
MAE	Mean absolute error
NLP	Natural language processing
SEE	Software effort estimation
SVM	Support vector machine
LSTM	Long-short term memory
LDA	Latent dirichlet allocation
PLSA	Probabilistic latent semantic analyse
TF-IDF	Term frequency Inverse Document Frequency

CHAPTER ONE

INTRODUCTION

1.1 Background and Motivation

Estimating the effort required to identify bugs in a pool of bug reports from a tracking system is a challenge for software maintenance researchers and practitioners [1]. Users can submit bug reports through a bug tracking system that allows for report analysis and the assignment of reports to fixers to address them [2]. However, it has been observed that bug reports are multiple bugs when more than two testers report the same bug, normally from the source code. A bug is regarded as a duplicated bug when it is reported by two testers or reporters [2]. Bug reports can then be used to guide software maintenance behavior and help in the development of more secure software systems. Recognizing software bug reports can aid in the triaging process, allowing developers to prioritize and address the most critical complaints first [3]. Developers frequently get a large number of bug reports and may be unable to handle them due to various constraints such as effort and time limits. Prioritization of bug reports is a time-consuming and difficult process [3].

Thus, a plausible solution for an effort estimation framework to detect multiple bugs will reduce the effort the developer and tester spend on analyzing bug reports while also improving software reliability and productivity. Advaita bug detection has been proven to be a good approach to assist developers in finding issues early in the software development process, saving time and effort [4]. However, they are limited in their ability to detect flaws that use numerous approaches and have a high percentage of false positives [5]. Regrettably, it appears that the research community has overlooked this crucial aspect of software engineering. Furthermore, the available databases

are massive and difficult to study manually [5]. As a result, an effort estimation framework model is required to anticipate the effort required in identifying and resolving all reported bugs [6]. Given all the advantages, someone would expect a well-founded theory and several applications in the area of effort estimation framework from available data [6]. Although rapid developments in computing technology have resulted in powerful multi-gigahertz CPUs, software stability has not caught up. Despite the rising need for software to be trustworthy, software program bugs continue to be common. System requirements must meet strict uptime requirements and must continue to operate even if hardware or software failures occur [7]. They may also be required to be monitored and debugged. Programs, like anything else created by humans, contain thoughtless errors or misconceptions, which we refer to as bugs or flaws [8]. Some bugs, such as null pointer dereference and double-free, cause system failures; others, such as memory leaks, degrade system performance; and yet others, such as memory leaks, do not cause failure but do not produce the expected results.

Testing, human code review, static analysis, dynamic analysis, and other methods have all been developed to find bugs and repair them [9]. This study was motivated by the fact that many overridden or overloaded methods can be found in large software projects written in programming languages [10]. Because all overridden or overloaded methods follow the same logic, it is evident that they could produce related or comparable errors [10]. Meanwhile, open-source software project developers frequently keep open bug mines. Users can report issues they find while using software in a "bug mine," and developers can confirm and solve the reported bugs in the next version.

1.2 Statement of Problem

Previous studies [1] [11] [12] have shown that in a given pool of bug reports from a tracking system, estimating the effort required to identify multiple bugs is a challenge because bug fixers normally go through a series of challenges identifying and estimating highly prioritized bugs from a given bug report repository to fix or resolve. As a result, a series of techniques have been introduced. One of the techniques is *triage* [11], which attempts to identify if a given report is of high priority and meaningful. This study seeks to prioritize bugs that have been reported by more than two reporters, hence the name "*multiple bugs*." Due to the vast nature of the existing bug reports, it is problematic for the *triage* [11] to evaluate all previous bug reports to discover *multiple bugs*. Only a small number of similar bug reports are obtained by the *triage* [12]. which then compares the new bug report to each retrieved bug report to check if it has been multiplied. Otherwise, the *triage* [12] must conclude that no multiplication has occurred. The difficulty of finding the main bug report for each new report inside the matching proposed bug list may then be considered as multiple-bug-report detection [13]. The potential problems or risks may be discovered while managing the projects that may use some shared resources. Therefore, the risk may be shared across multiple software projects.

Less effort estimation techniques have been developed to accommodate the detection of multiple bugs in a pool of bug reports by software testers [14]. If traditional techniques are used for estimating the effort required for detecting multiple bugs in software projects, the findings will undoubtedly be erroneous [14]. On the other hand, current practise in software project work estimating is based on a single iteration [15]. In the case of open-source projects, some features, such as persistent changes in levels of scope or product, are not emphasised [15]. Software complexity and human resource issues, such as technical and analytical skills

[16]. As a result, to anticipate the development work of software projects, an effort estimation model for detecting multiple bugs is being theoretically and empirically investigated to assist fixers to identify and resolve such priority bugs [17].

1.3 Scope of the Study

This study focuses on the detection of multiple bugs and estimates the effort required to detect multiple bugs. Established bug report data from Mozilla Firefox and Eclipse open-source projects archived in the Github repository is analyzed for this study.

1.4 Research Objectives

1.4.1 Global Objective

The main objective of the study is to develop a framework to detect multiple bugs and estimate the effort required to detect multiple bugs in open-source projects.

1.4.2 Specific objectives

The study addresses the following objectives:

1. To review existing techniques from the literature for detecting bugs in software projects.
2. To develop a framework to detect instances of multiple bugs in open-source projects.
3. To define an effort estimation metric for detecting multiple bugs in open-source projects.

1.5 Research Input and Contribution

1.5.1 Research Input

According to Li and Reformat [18] the knowledge input of a thesis can be a strategic technique, prototypical, construct, method, or instantiation. This study adds to the pool of information by considering the prospect of multiple bug detection techniques in a software project that is lacking. The research covers the application of a known tool, *triage*, in the software management system. Thus, this is an exaptation study according to Li and Reformat [18].

1.5.2 Academic Input

The study will ascertain whether the deep learning model has a better tendency than the conventional machine learning model at detecting multiple bugs in open-source projects.

1.5.3 Practical Contribution

The study makes the following applicable contributions:

1. A framework to detect multiple bugs in open-source projects.
2. An effort estimation metric for estimating the time required to identify multiple bugs.

1.6 Organization of the Thesis

Chapter two presents a literature review on bug detection in software projects.

Chapter three discusses the framework and methodology employed for the study. The dataset used is discussed, with data pre-processing and experimental setup explained.

Chapter four presents the experimental results and discussions from the empirical analysis conducted in this study.

Chapter five presents the conclusion and future work of the study.

CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

As mentioned in the previous chapter, the aim of this thesis is to develop a framework to estimate the effort required to detect multiple bugs in open-source projects by comparing its performance with machine learning techniques. Therefore, this chapter presents the literature review. The literature review focuses on machine learning techniques in bug detection and the need for this study. The chapter is divided into seven sections. The first section presents the introduction. The second section presents an overview of a framework for finding bugs in software projects. The third section presents a review of bug finders in a software project. The fourth section identifies the duplicate bug tracking system. The fifth section presents techniques for finding duplicate bugs. The sixth section uses a machine learning technique to describe approaches for estimating effort in software development. The seventh section discusses software project estimation methods.

2.2 Overview

This section introduces the DeepBugs framework [19] for constructing name-based bug detectors using machine learning. The fundamental idea is to teach a classifier to distinguish between codes with and without name-related bugs [19]. A "bug pattern" is a series of programming errors that break the same routines [20]. Bug patterns include accidentally switching a role's arguments, using the wrong API calls, and using the wrong binary operator [19]. Hand-built bug checkers like FindBugs and Error-Prone use bug patterns that require distinct analysis [21]. Using DeepBugs to create a bug detector needs several processes [19].

(1) Create training data from the corpus. This stage selects good instances from a corpus and produces negative examples [17]. Inferred positive code examples are unlikely to be affected by the bug pattern because we assume the bulk of the codes are accurate [22]. DeepBugs uses basic code alterations to build negative training examples [19].

(2) Develop a model that can distinguish between correct and incorrect data.

Given two sets of codes containing positive and negative examples, this stage trains a classifier to distinguish between them. The classifier is a feedforward neural network.

(3) Detects previously unknown code flaws. This phase employs the previous classifier to determine if a new piece of code is affected by the bug pattern [17]. The approach warns the developer if the trained model thinks the code is wrong. An example of this is a bug detector that looks for incorrectly ordered function arguments [21]. A possible problem in Step 6 requires swapping the dim and x-dim variables. As a result, the approach may detect such inaccuracies because the learnt classifier generalizes beyond the training data. That width and x dim are pairwise semantically similar in the case that allows DeepBugs to find the bugs [19].

2.3 Find Duplicate Bugs from Program Inconsistency

The find duplicate bugs from programme inconsistency technique is used by prominent and successful commercial static analysers, such as Coverity [23] . One of the most difficult aspects of discovering duplicate defects is determining which rule a piece of programme code should obey[24]. Allowing the programmer to specify it, either by saving it to an extra file or using a special comment type, is one option [25]. Program inconsistency identification, the approach presented in this section, may automatically extract such rules from the source code rather than the

programmer [26]. Program "beliefs" are the key to this technique [27]. A dereference of a pointer p indicates that p is non-null, and a function call unlock (l) implies that the lock l was locked, which are examples of programme "beliefs." They are divided into two categories for "beliefs," one for "must believe" and the other for "may believe." The phrase "must believe" is pulled from code, and there is no doubt that the programmer believes it. When a pointer is dereferenced, the programmer assumes that the pointer is not null [27]. "May believe" refers to when we see a pattern in the code that suggests a fact, but it could just be a coincidence [27]. A call to the function "adds" followed by a call to "dec" could indicate that the programmer believes they must be called painstakingly, but it could merely be a coincidence [28] . While this "may belief" is highly likely to be true if this pairing occurs 999 times out of 1000 [22].

2.4 Techniques for Finding Duplicate Bugs

Automatic systems for finding software faults need to know the rules that a program must follow, or "specifications" [29]. Finding duplicate bugs in a software project can be done in a variety of ways [30]. Code inspections can be quite effective in detecting duplicate bugs, but they come with the apparent drawback of needing a lot of manual labor [31]. Human observers are also techniques, but they are susceptible to being influenced by what a piece of code is supposed to perform [32]. The advantage of automatic approaches is their relative objectivity [33]. Static analysis evaluates code without running it and can uncover potential security violations (e.g., SQL injection), as well as runtime faults (e.g., dereferencing a null pointer), as well as logical contradictions [21]. While there is a wealth of literature on the algorithms and analytical frameworks employed by such tools, publications describing real-world experiences with such tools are harder to come by in terms of the sophistication of their analysis methodologies. FindBugs does not push the envelope; rather, it is intended to determine what types of faults may be effectively discovered using relatively simple

procedures, as well as to assist us in understanding how such tools might be integrated into the software development process [21] . FindBugs has been downloaded over 580,000 times and is used by several large corporations and software projects [21].

Testing and assertions are dynamic procedures that rely on a program's runtime behavior [34]. This can be both a benefit and a drawback. The benefit is that dynamic techniques do not consider infeasible routes [34]. The problem is that they are usually confined to discovering duplicate bugs in the executed program pathways [34]. Furthermore, the time necessary to execute a full suite of tests can be prohibitively long. Static approaches, in contrast to dynamic techniques, can examine abstractions of all potential program behaviors and are therefore not constrained by the quality of test cases to be effective [33]. The complexity of static approaches varies, as does their ability to find or eradicate duplicate bugs. A formal proof of correctness is the most effective (and difficult) static technique for removing duplicate bugs [35]. Having a correctness proof is probably the best way to ensure that software is free from duplicate bugs [35]. These strategies demonstrate that a program's intended property holds for all conceivable executions. These techniques may be complete or incomplete; if incomplete, the analysis may be unable to verify that the required attribute exists for some right program [36]. Finally, while ineffective approaches can detect "possible" duplicate bugs, they can also miss true duplicate bugs and inaccurate warnings [37]. There are several attempts to find duplicate bugs from available data reported in the literature.

Sharma et al. [38] introduced a priority prediction strategy based on SVM, neural networks, NB, and KNN. The proposed approach enables the prioritization of bug reports. The results indicated that, with the exception of the NB technique, the accuracy of the machine learning techniques utilized in estimating the importance of bug reports inside the project was greater than 70%.

Badashian et al. [39] developed a new strategy for bug triage that involves the utilization of Q&A community sites such as GitHub, which is a source of developer expertise. This approach extracts keywords from bug reports by utilizing keywords (from metadata), project languages, tags from "GitHub" or "Stack overflow," as well as title and description fields. These terms are synonymous with social expertise. They examined this strategy using bug reports from 20 GitHub projects and discovered that it had an accuracy of 89.43 %.

Mani et al. [40] suggested a deep bidirectional recurrent neural network (DBRNN-A) model for the approach. The model is for categorizing appropriate developers based on the title and features of individual software bug reports, utilizing Naive Bayes, cosine distance, SVM, and SoftMax. Experiments with software project bug reports (e.g., Google Chromium, Mozilla Core, and Mozilla Firefox). It demonstrated a precision of 47% for Google Chromium, 43% for Mozilla Core, and 56% for Mozilla Firefox.

Lyubinets et al. [3] describe an RNN-based model for labelling bug reports. Accuracy was achieved in the 56–88 percent range.

Nguyen et al. [41] introduced DBTM, a method for detecting duplicate bug reports that takes into account not only IR-based but also topic-based aspects. Each bug report is modelled by DBTM as a textual document explaining one or even more technical difficulties, and duplicate bug reports are modelled as those reporting the same technical issue. With historical data, including duplicate reports that have been detected, DBTM may learn sets of phrases expressing the same technical concerns and detect further not-yet-identified duplicates. Our empirical examination of real-world systems demonstrates that DBTM can increase the accuracy of state-of-the-art techniques by up to 20%.

Shokripour et al. [2] presented a two-phased approach based on allocation recommendations on the bug's expected location. This approach makes use of the source code as well as the title and description fields, from which meaningful terms are retrieved. They used bug reports from Eclipse and Firefox to reach an assessment accuracy of 89.41 percent and 59.76 percent, respectively. However, the quantity of bug complaints is significantly lower than in previous studies.

Shokripour et al. [42] introduced an ABA-time-TF-IDF approach, a strategy for automatic assignment based on TF-IDF time metadata. A corpus is built using words, and developers are identified and recommended for specific expertise. Shokripour compared the ABA-time-TF-IDF to the ABA-time-TF-IDF, the NB, the VSM, the SUM, and the SVM in Eclipse, NetBean , and ArgoUML projects. As a result, accuracy and mean reciprocal rank (MBR) improved by up to 11.8 and 8.94%, respectively.

Bhattacharya and Neamtiu [43] suggested an approach based on enhanced classification to increase the accuracy of bug report triage and shorten the tossing pathways. It utilizes the NB and a tossing graph to extract features such as TF-IDF or bag-of-words (BOW). It takes advantage of meta data from various sorts of bug reports, title, description, keywords, product, component, and the most recent developer actions. They evaluated this technique using bug reports from Eclipse and Mozilla, achieving 77.43 percent accuracy for Eclipse and 77.87 percent accuracy for Mozilla.

Bashar et al. [3] used a 5-layer deep learning RNN-LSTM neural network to estimate the priority of bug reports, comparing the results to those from Support Vector Machine (SVM) and K-nearest neighbors (KNN). The suggested RNN-LSTM model's performance was evaluated using the JIRA dataset, which contains over 2000 bug reports. The proposed model was found to be 90% accurate when compared to KNN (74% accuracy) and SVM (87 percent).

Sawarka et al. [44] introduced a new technique for estimating the time required to fix a newly reported bug. The developed algorithm is validated against a publicly available database. A comparative analysis is performed in which the output of the algorithm is compared to the output of the algorithm using various machine learning approaches. A highest accuracy of 64% is achieved, which is superior to any previously proposed algorithm in this domain.

2.5 Duplicate Bugs Fix Effort Estimation using Machine Learning

There are several attempts at estimating efforts from available data reported in the literature. A study by Mahmood et al. [45] used NASA's KC1 dataset to present a solution for estimating software defect fix effort using self-organizing neural networks. The basic idea behind their work was to identify the most similar earlier-issued duplicate bug and use the reported normal time as an estimate. To investigate the effort for new issue reports or duplicate bug reports, they used the nearest neighbour approach [46]. They found out that the association rule mining technique is better at predicting the effort compared with other machine learning techniques like PART, C4.5, and Naive Bayes. The empirical evaluation is based on the Eclipse bug database [46]. The proposed model can correctly predict up to 34.9% of the bugs into a discretized log scaled lifetime class. Although the authors introduced a method for developing an effort estimation model, they did not present such a model.

A study by Ahsan et al. [47] concluded that the effort estimation using programmer involvement is lower than the estimation from product metrics. He obtained effort data from a commercial company and also collected a set of product metrics like lines of code, Halsted vocabulary, object and complexity metrics. He designed a duplicate set of neural networks based on a different combination of hidden layers and the number of perceptron per layer. He used different

combinations of product metrics to train the model. He performed the experiments using 33,000 different models of neural networks and collected the best data [47]. His result shows that neural networks can be used for an effort estimation model. A huge number of experiments are necessary to obtain the best framework. The majority of effort estimation work is associated with a commercial or closed software system [47]. Very few research papers for OSS are available, especially in the area of duplicate bug fix effort estimation [48]. Most of the mentioned work used machine learning techniques to train the model using available effort data, whereas in our case we do not have effort data. Therefore, we have to extract the effort data from OSS to perform the experiments [48]. A study by Sharma et al.[49] showed that SVM and neural networks yielded improved performance as compared to Naive Bayes and KNN in predicting the priority of reported bugs.

A study by Chaturvedi and Singh [50] classified bug severity and found that neural networks yielded an improved performance as compared to KNN, SVM, Naive Bayes Multinomial, RIPPER, and J48 in defining the class of bug severity.

A recent study by Sawarkar et al. [51] recommended SVM with sigmoid functions as the best classifier, followed by Random Forest for predicting bug estimation times for newly reported bugs.

Malhotra et al. [52] showed that the single-layer perceptron (neural networks) is the best approach among all the techniques used in this study for the development of a defect prediction model.

A study by Baarah et al. [53] determined the class of bug severity and found that Logistic Model Trees yielded improved performance as compared to Naive Bayes, Naive, SVM, KNN, and Random Forest in predicting the severity level of software bug reports in closed-source projects.

A study by Ahsan et al. [54] used NASA's SEL bug dataset and applied association rule mining to the dataset to detect "duplicate bugs" using intervals. They discovered that association rule mining outperforms other machine learning algorithms such as PART, C4.5, and Naive Bayes in detecting duplicate bugs [54]. Although the authors introduced a method for developing duplicate bug detection models, they did not present such a model. The problems of programmer engagement and effort modelling for OSS were discussed[7]. He worked with data from the ECLIPSE project and calculated the effort based on programmer participation and product metrics [55]. He concluded that detecting duplicate bugs based on programmer engagement is less accurate than detecting duplicate bugs based on product metrics [56]. They used Microsoft Project Visual Studio data for their research work [56]. Their statistical research revealed that the real detection error is positively connected with the featured size, as well as the process metrics and the detection error. They developed a duplicate detection model using a neural network [57]. He designed a duplicate set of neural networks based on a different combination of veiled layers and the sum of perceptron per layer [57]. He performed the experiments using different models of a neural network and collected the best data. His result shows that a neural network can be used to find duplicate bugs in software projects [57]. A huge number of experiments are necessary to obtain the best framework. Most of the duplicate bug detection work is related to commercial or closed software systems [29]. Very few research papers for OSS are available, especially in the area of duplicate bug fixing.

2.6 Estimation Approach in Software project

The expedition for better software systems, value, and consistency is not an ending point. Several techniques have been presented to help testers or developers detect and resolve duplicate bugs

[24]. To test the application starting with static techniques (database analysis, duplicate bugs, duplicate bug prediction, classical inspection, and authentication). Bug detection, for example, assists testers or developers in detecting several bugs early by analyzing source code directly and determining whether or not a specific source code is buggy [58]. It has been demonstrated that detecting duplicate bugs improves software quality and dependability [59]. The existing state-of-the-art bug detection techniques are divided into the following phases or categories:

- Duplicate bug detection based on rules: Several programming criteria are set in this method to statically detect common programming errors or problems. FindBugs is a common example of this kind of model [21]. Although this kind of method is quite active, novel directions for detecting new forms of duplicate bugs.

- Bug identification based on data mining: Mining-based techniques depend on removal of current source code to get around pre-defined rules [60]. Typically, exploitation data mining techniques generate implicit software design instructions after database source codes and detect violations of the retrieved rules as probable duplicate bugs [60]. These mining-based techniques still have a significant flaw in that they cannot distinguish between wrong codes and infrequent or rare codes, resulting in a high false-positive rate.

- Duplicate bug identification with machine learning [46]: Through the advancement of machine learning (ML), particularly deep learning models, numerous ways to learn from previously known and reported duplicate defects and detect bugs in new code have been presented [61]. While ML-based duplicate bug detection models depend on feature selection techniques, duplicate bug detection models use the capacity to learn essential features from training data.

This technique is shown to have advantages over existing ML-based duplicate bug identification algorithms. Deep learning techniques are still limited to detecting duplicate defects in single operations without taking their interdependencies into account. In practice, there are various instances when duplicate problems occur in multiple methods [62]. To determine whether a method is flawed or not, a model must take into account other methods that share data or control with the method under review. Thus, deep learning-based algorithms have high false-positive rates, making their regular use by software engineers impracticable [23]. That is, approximately one in three reported duplicate problems is a false positive, wasting engineers' time [63] . Duplicate bug reports can be bundled and processed sequentially[12]. Techniques for decreasing duplicate bug reports include NLP, IR, and similarity-based classification. Artificial Intelligence (AI) is used to find duplicate bug reports [46]. Because bug reports are written in human language, distinct terminology is commonly used [64]. Methods (including stemming, lemmatization, and stopword processing) are used to improve accuracy. Topic modelling is the most often used NLP technique, processing words from bug reports or summary text and selecting criterion words. The topic picked represents a few features of the problem report and specifies a duplicate bug report. The most common subject modelling technique is the LDA, NB, and NB polynomial [65]. We discovered duplicate bug reports using the LDA, NB, and NB polynomial [2]. This was done using Eclipse bug reports to compare the results [66]. As a result, nearly 80% of duplicate bug reports were discovered. They also showed substantial differences in employing statistical approaches, including the LDA and N-gram (LNG) technology [2]. A linearly connected weight-based N-gram similarity and topic modelling using the existing LDA approach. The N-gram (LNG) approach has better recall, precision, and EA rates than existing algorithms in detecting bug report duplication [2].

When DTBM, a cutting-edge approach for detecting bug report duplication, was used, the recall rate increased by 2.96 percent to 10.53 percent [2]. Meta data such as bug type, operating system, priority, and severity aid in the identification of similar bug reports [37]. Approaches to identifying duplicate bug reports based on information retrieval are frequently used in concert with these other methods focusing on contextual information to deduplicate bug reports [63] . They extracted context from bug reports using BM25F techniques, a scoring algorithm commonly used in information retrieval [63]. The word list was constructed in response to a non-functional requirement relating to software quality as well as the subject of the bug report. As a result, the number of duplicate bug reports was cut in half. The LDA, NB, and NB polynomials were used to find duplicate bug reports [2]. Using eclipse bug reports, they compared the results to conventional machine learning [66]. As a result, nearly 80% of duplicate bug reports were discovered. They also found substantial variations in the statistical approaches offered for the LDA and N-gram (LNG) technologies [2]. Metadata like bug type, OS, priority, and severity assist in identifying related bug reports [37]. Even in the same setting, bug reports can have diverse solutions. For bug report deduplication, information retrieval techniques are commonly employed in concert with other methods focused on context [67]. They employed BM25F methods, a scoring algorithm used in IR fields, to extract context from bug reports [68]. According to their research, non-functional needs linked to software quality and the nature of bug reports influenced the terms list. As a result, the number of duplicate bug reports fell by 11.55 percent.

2.7 Effort Estimation Accuracy

To determine whether one effort estimating model produces better results than another. In the literature, a variety of different prediction accuracy statistics have been utilized. The Mean

Magnitude Relative Error (MMRE) is an average error that indicates how much the predictions overestimate or underestimate the model's true value [45]. The magnitude of relative error (MRE) is derived by multiplying the difference between the actual and estimated values by the actual value [45]. As a result, the mean MRE (MMRE) is the average value of this indicator across all observations in the dataset. In general, a lower MMRE score suggests a more accurate model. On the other hand, the coefficient of duplicate determinations indicates how much of the variation can be explained by the independent variables. When R2 approaches 1, it means that the independent and dependent variables have a strong relationship. The quality of the predictions is represented by the PRED (k) prediction level parameter.

2.8 Chapter Summary

The chapter reviewed related literature for duplicate bugs and effort estimation. The review acknowledged that there has been much improvement in research on machine learning algorithms for effort estimation in finding duplicate bugs. Besides similar documents that offered important information for the review, the publications indicate that no work has been done to investigate an effort estimation framework for detecting multiple bugs.

CHAPTER THREE

RESEARCH METHODOLOGY

3.1 Introduction

This chapter presents the empirical framework and the experimental set-up of the study. Data pre-processing procedures, feature selection techniques, and model evaluation techniques used in the study are all discussed in the empirical framework. It also includes a list of the datasets and models that were used. In the experimental setup section, the model design technique is described.

3.1.1 Selection Criteria

To extract various datasets used to train multiple bug detection models, the following inclusion and exclusion criteria were defined.

3.1.2 Inclusion Criteria

i. The papers were published between 2010 and 2020.

ii. The study used articles published in reputable sources journals and conferences as its sources.

iii. The study used machine learning techniques for the estimation.

3.1.3 Exclusion Criteria

i. Studies that were not published in English were excluded.

ii. Working papers or work in-progress articles were excluded.

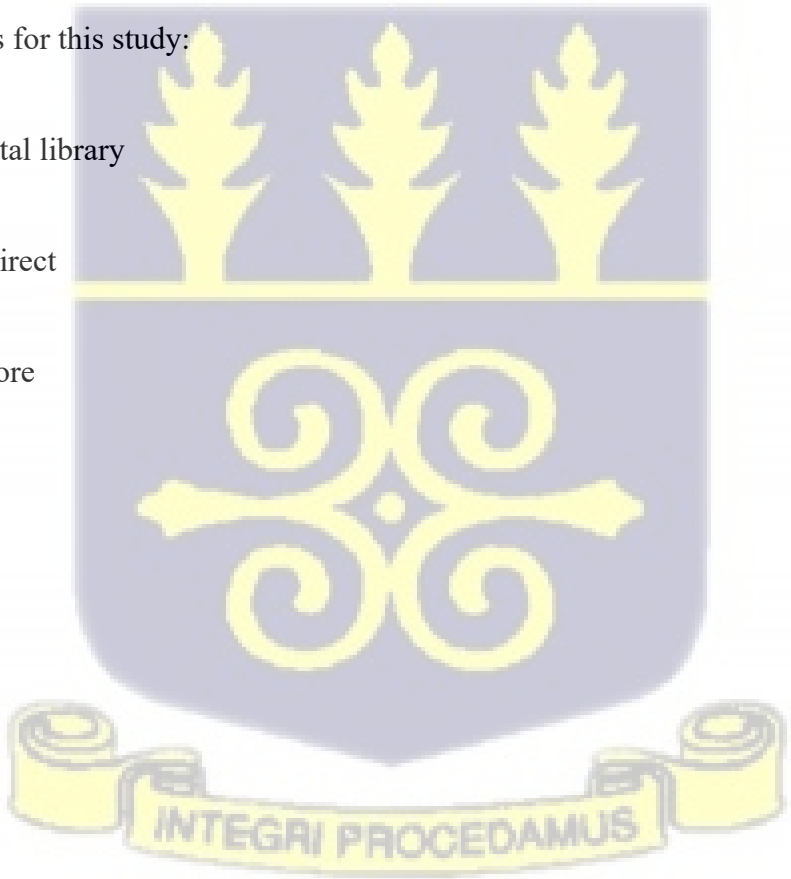
3.2 Search process

A robotic search strategy was used to find suitable studies for the review in four separate databases: ScienceDirect, IEEE Xplore, ACM Digital Library, and Springer. Because the chosen databases are common in the fields of engineering and computer science, it was assumed that they would cover major papers on the research issue. In addition, the percentage of content updates in these databases was examined as part of the selection process.

3.3 Classification of Papers

Papers used in this study are categorised by the publishers or journals in which they were published. The following platforms for full-text database offering journals were used to find research materials for this study:

1. ACM digital library
2. Science Direct
3. IEEE Xplore
4. Springer



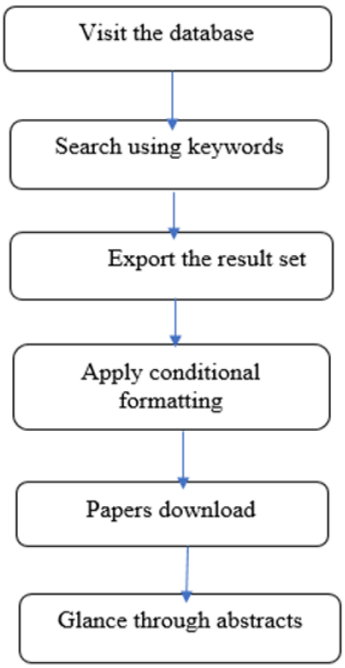


Figure 1. Order of the selection process (SLR protocol)

Table 1.Distribution of search results from selected journals

Journal	Overall database results retrieved	Final results retrieved after exclusion criteria
ACM digital library	39	14
Springer	1	1
Science Direct	175	27
IEEE Xplore	19	8
TOTAL	234	50

3.4 Case Study Setup

The dataset used for the experiment is publicly available bug reports from a free and open-source browser, Mozilla Firefox, and Eclipse, which is an integrated development environment used in computer programming. Datasets obtained from the two platforms have 115814 and 85156 records, respectively, and 11 attributes, that is: priority, issue_id, component, duplicated_issue,

title, descriptions, status, resolution, version, created time, resolved time. The datasets consist of an additional and manually labelled attribute, Classes, which determines the presence or absence of multiple reports made on bug issues and is determined from the 'Duplicated_issue' attribute. In the 'Classes' attribute, 0 represents the absence of multiple reports and 1 represents the presence of multiple reports. The number of 0 records was 80,000 and 35,814 for that of 1 for the Mozilla Firefox, as well as 70752 0 records and 14404 for that of 1 record for the Eclipse platform.

Table 2. Characteristics of multiple bug reports used in the experiment

Dataset	Total Number of bugs	Multiple bugs
Mozilla Firefox	115814	35814
Eclipse	85156	14404

Bug reports stored in Bugzilla issues were extracted from two open-source projects, namely Mozilla Firefox and Eclipse, sourced from recognized repositories. These two open-source projects were used for our case studies.

3.4.1 Bug Reports from Mozilla Firefox

Mozilla Firefox is a well-known Internet browser used by millions of people around the world. Firefox used Mozilla as a crash reporting system and Bugzilla for tracking bug reports. The dataset used for the study is a publicly available Mozilla Firefox bug report dataset containing 115814 records and 11 attributes. The dataset consists of an additional and manually labelled attribute class, which determines the presence or absence of a multiple bug report and is determined from the "duplicate issue" attribute. In the 'Classes' attribute, 0 represents the absence of a multiple and 1 represents the presence of a multiple bug. The number of 0 records was 80,000 and 35,814 for

that of 1. There is a clear cut between crash reports and bug reports for Mozilla Firefox products. Crash reports are robotically reported when a crash usually occurs. The report contains several fields, including stack trace, priority, product, component, and OS version [64]. We extracted multiple bug reports from the Mozilla Firefox website, from BR # 1 to BR # 115814. Mozilla uses "description" and "classes" to group bugs. We extracted reports with the status "RESOLVED MULTIPLE" or "VERIFIED MULTIPLE" and found their corresponding master bug report. In addition, we only included the dataset bug reports that have links to their corresponding crash reports since our method uses stack traces, which are only saved in crash reports. We only added multiple bug reports that have at least two stack traces, the strict minimum number of traces needed to construct the training and testing sets.

3.4.2 Bug Reports from Eclipse

The ECLIPSE dataset comprises bug reports from the Eclipse bug repository. In this repository, stack traces are usually embedded in the bug report description. For Eclipse, the extracted traces were preprocessed to remove noise in the data, such as native methods that are used when a call to the Java library is performed. Unlike Firefox, which uses a variety of systems to track stack traces, Eclipse uses a single system. A bug report has stack traces as part of the bug report description, compared to Mozilla Firefox. We generate multiple groups by putting in separate multiple group stack traces of a master bug report. We only saved multiple bugs with the smallest of two stack traces to construct training and testing. Features of multiple bug report groups in Mozilla Firefox and the Eclipse dataset.

Table 3. Description of case study projects

Metric	Mozilla Firefox	Eclipse
LOC	23738195	129278
Comment	4934094	36637
Developers	16	8
Release	May, 2021	June,2018
Version	91.0	4.8

3.5 Dataset Preprocessing

Since we are dealing with text data written by humans, the data is found to be noisy and dirty as a result of the use of excessive punctuation, slang words, repeated words, mixed-case letters, and misspellings. As a result of the imbalance in text data, it would have to undergo certain data cleaning techniques to qualify it for training and testing. The techniques used here are standard techniques known for preprocessing in natural language processing. Some of the preprocessing techniques used are lowercasing, where we transform all upper-case letters to lowercase for the purpose of dimensionality reduction. Text data in general comes in larger sizes with a lot of unnecessary punctuation, and this punctuation would have a negative impact on the final result and would have to be removed. This procedure, which is an NLP normalization technique, is also performed by removing the suffixes and prefixes of words. This is to help achieve the root form of the texts. Other techniques considered were the removal of stop words, which are common words in the English language that do not convey any meaning. Examples include "he," "the," "is," "have," and so on.

3.6 Feature Extraction

Although at times, data comes in smaller sizes, at most times, they come in tremendously larger sizes, which makes its processing very tiring and time-consuming. This is as a result of the number

of features in the dataset, as the higher the number of features increases, the larger the file size, though they might come in different forms sometimes. The contribution of these types of datasets is often less towards predictive modelling as compared to that of the critical features. Problems caused by these features during implementation can be the unnecessary resource allocation for the features, the poor performance of the model, and more exhaustion for the training model. To solve this issue, as well as maximize the performance of the model, feature extraction is implemented, which would select the most significant features from the dataset. Two feature extractions were used in the study, which included TFIDF and GloVe.

3.6.1 TF-IDF

The term frequency is used to normalize the document term matrix. It is a product of TF and IDF. TF-IDF (term frequency-inverse document frequency) is a statistical measure that evaluates the relevance of a word in a pool of words. One major issue with machine learning and deep learning with natural language is that they are both dealing with the same problem. Its algorithms perform computations and workflow with numbers. However, the natural language is text, and hence there is a need to convert text into numbers, which is classified as text vectorization. Diverse vectorization techniques would have a significant impact on the results; thus, the one that best fits the expected result would need to be chosen. Once the words have been successfully transformed into words that the machine learning algorithms can understand, the TF-IDF output can then be fed to algorithms such as Naive Bayes and Support Vector Machines, greatly improving the results of basic methods such as word counts.

3.6.2 GloVE

The glove is an unsupervised learning algorithm for obtaining vector representations of words. Word embeddings are dense word representations that bridge the gap between human understanding of language and that of a machine. Words with the same meaning have a similar representation in an n-dimensional space. Meaning that similar vectors that are very closely placed in a vector space are said to be words that have a similar meaning. On the other hand, GloVe (Global Vectors for Word Representation) is an alternative method to generate word embeddings that is normally based on matrix factorization methods on the word-context matrix. A large matrix of co-occurrence data is constructed where a count of each "word", i.e., the rows, and how often the word appears in a "context", i.e., the columns in a large corpus, is calculated. Usually, the corpus is first scanned in a manner such that, for each term, a search for context terms within some area is defined by a window size before the term and a window size after the term. Also, less weight is given to words that are more distant.

3.7 Traditional Machine Learning and Deep Learning Models

3.7.1 Traditional Machine Learning

Once there are representations of the input data, the model would go through training to fit the data to be capable of making predictions on unseen reports. With results from TFIDF, data is split into two forms, that is, the training data and the testing data, which would be in the ratio of 90% for training data and 10% for testing data using the Scikit function train test split. The training data is then passed through a machine learning classifier, specifically, the Random Forest classifier. Once it is trained with different training samples, the machine learning model can start or begin to make correct predictions. These predictions from machine learning would serve as a baseline for the deep learning algorithm.

Table 4.Machine learning dataset training details

Dataset	Training Data		Testing Data	
	#Multiple	#All	#Multiple	#All
Mozilla Firefox	32145	104232	3669	11582
Eclipse	12934	76640	1470	8516

3.7.2 Deep Learning

In deep learning, similar processes to those in machine learning would be repeated here. However, in feature extraction, the deep learning model makes use of GloVe word embeddings. With results from the GloVe, a model is built using bidirectional LSTM with an epoch of 5 and a batch size of 128.

Table 5. Deep learning dataset training details

Dataset	Training Data		Testing Data	
	#Multiple	#All	#Multiple	#All
Mozilla Firefox	32246	104232	3568	11582
Eclipse	12946	76640	1458	8516

3.7.3 Bidirectional LSTM model

Researchers and practitioners have presented a number of unique and successful neural network and probabilistic models to handle bug identification challenges. For example, the proposed Bidirectional LSTM model evaluates both the prior and following context of codes to detect

mistakes and offer solutions that will allow testers and developers to make necessary changes quickly. Our approach uses a BiLSTM neural network to model language. The LSTM network computes the output vectors based on the input data.

Now, let I = $i_1, i_2, i_3 \ldots\ldots\ldots i_t$ be the set of encoded IDs of source codes.

An RNN then executes for each encoded ID for t = 1 to n. The output vector of RNN y_t can then be expressed by the following equations:

$$\boldsymbol{h_t = tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h} \quad (1)$$

$$\boldsymbol{y_t = W_{hy}h_t + b_y)} \quad (2)$$

where h_t is the hidden state output,

W is a weight matrix (W_{xh} is a weight connecting input (x) to hidden layer (h)),

of the hidden layer.

The first equation calculates the hidden state output, which receives the preceding state's results. However, because of the gradient vanishing or exploding problem, not all input sequences are successful in an RNN. RNN is extended into LSTM to help combat this and improve results. An LSTM network is conceptually similar to an RNN, except that the hidden layer updating is replaced with a memory cell. The following equations implement LSTM:

$$\boldsymbol{i_t = \sigma(W_{xi}\,x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i)} \quad (3)$$

$$\boldsymbol{f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f)} \quad (4)$$

$$\boldsymbol{c_t = f_t c_{t-1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)} \quad (5)$$

$$\boldsymbol{o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o)} \quad (6)$$

$$\boldsymbol{h_t = o_t tanh(C_t)} \quad (7)$$

where σ is a sigmoid function; c the cell state f the forget gate, i the input, o as the output, and b represents biases. However, there are a few flaws it poses: LSTM networks are slow to train and, as such, it takes a significant number of times for the neural network to learn. Predominantly, it considers only the previous context of the input but cannot consider any subsequent text context. In LSTM, what happens is that the network learns from the left -to- the right and right-to-left contexts separately, then concatenates them. This process it undergoes causes the true meaning of the context to be slightly lost. As such, in this study, we propose the bidirectional LSTM model as a better upscale for our task, where the first is taken as the forward direction for the input and the other serves as the backwards direction. Having Bidirectional LSTM will intend, effectively enhance the amount of information to be made readily available for the network, thus improving the context availed to the algorithm.

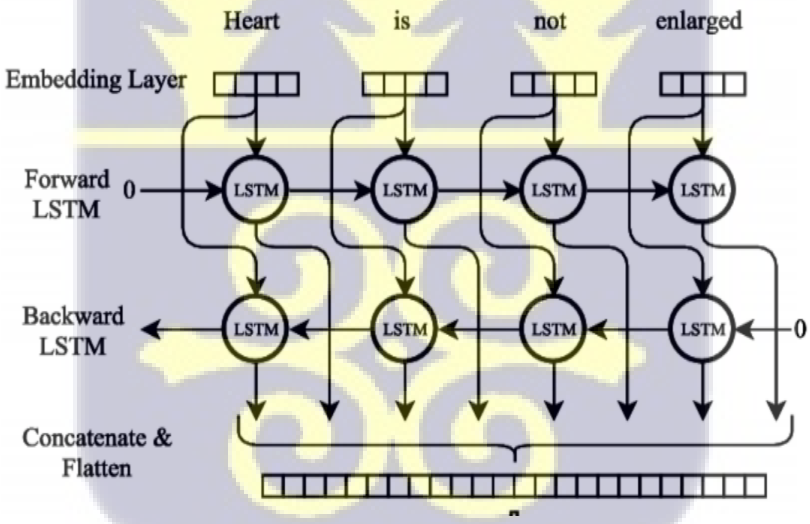


Figure 2. Bidirectional LSTM framework (Sanchoy et al.[69])

3.8 Multiple Bug Detection and Effort Estimation Framework

Figure 3 describes the multiple bug detection and effort estimation framework of the two-study dataset, that is, Mozilla Firefox and the Eclipse bug report repository. Once we have a bug report repository, we extract bug reports with stack traces using the bug detector tool (*bug tracking system*), selecting instances of multiple bug reports based on their severity levels, namely critical or less critical. Note that from Figure 4, we classified *low* and *medium-severity* instances based on the bug report description as *less critical,* and we classified instances with *high* and *critical severity* as *critical.* Descriptions of *low, medium, high,* and *critical* are provided in Figure 3. If active or critical, we compute the identification duration or effort needed to detect the bug. A primary fix means the identified instance of multiple bugs is very critical or high priority, which demands attention for fixing, whereas a secondary fix demands less critical attention.

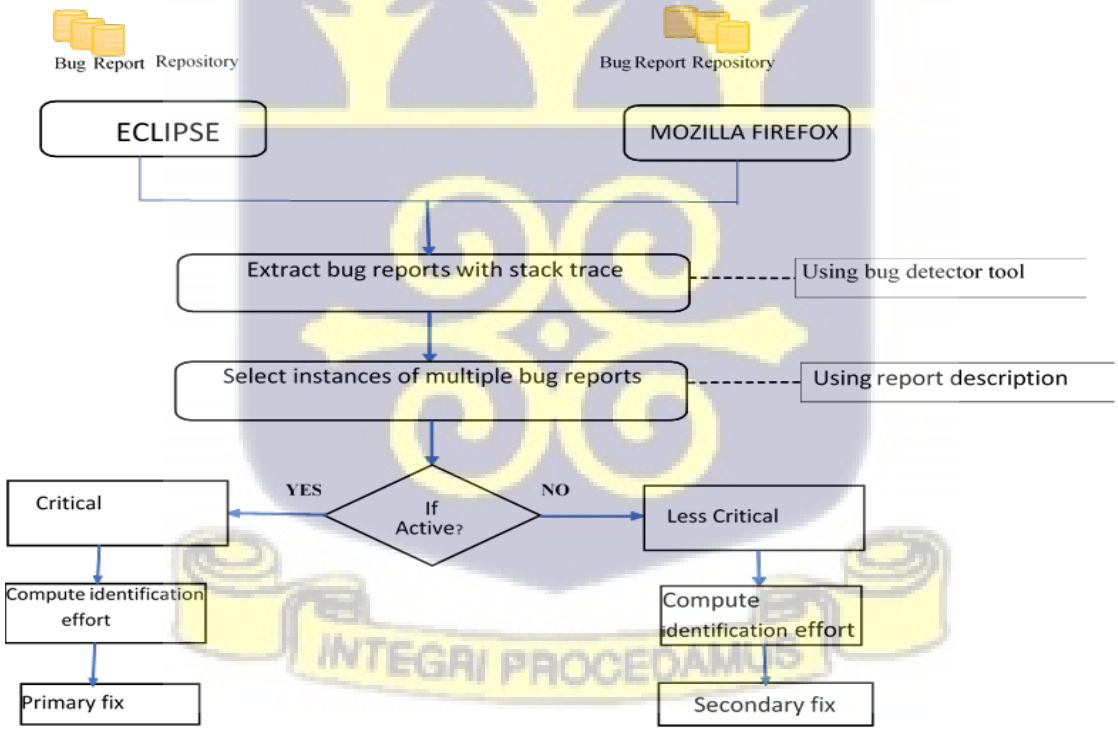


Figure 3. Multiple bug detection and effort estimation framework(mBugDEE)

3.9 Performance Evaluation

The metrics accuracy, precision, recall, and F1score are used to evaluate the performance of the model. Precision is basically known as a metric used to quantify the number of positive class predictions that actually belong to the positive class. Recall is also known to be the sum of positive class predictions made out of all positive instances. The F1 score provides a single score that contains both the concerns of precision and recall.

Precision $$\boldsymbol{P} = \frac{\boldsymbol{TP}}{\boldsymbol{TP+FP}} \quad (8)$$

Recall $$\boldsymbol{R} = \frac{\boldsymbol{TP}}{\boldsymbol{TP+FN}} \quad (9)$$

F1-Score $$\boldsymbol{F} = \frac{\boldsymbol{2RP}}{\boldsymbol{R+P}} \quad (10)$$

The symbol TP, which is "True Positive," normally occurs when the predicted output is positive, which is an indication of attack for this study, and the actual instance is positive. The symbol FP, which is “False Positive,” occurs when the predicted output is falsely positive, an attack for study is carried out. The F1-Score is the average or vocal mean of R (recall) and P (precision).

3.10 How much effort is required to identify and resolve a multiple bug?

Motivation: There are many duplicate bug reports. According to a recent study, 20–40% of the issue reports in Mozilla and Eclipse are duplicates [70]. Duplicate issue reports waste developers' time and resources. As a result, many researchers have suggested automatic duplication report detection methods [61]. Identifying and resolving duplicate reports has received little attention in recent years. Hence, this study investigates a feasible technique to estimate the effort of identifying and resolving duplicate as well as *multiple bug* reports. Note that when a bug is reported by one

reporter on two or more occasions, it is referred to as a "*duplicate bug*" [70]. But in the case of multiple bugs, the same bug is reported by different reporters.

Method: The identification duration statistic measures the time it takes to identify multiple bug reports. That is, the longer it takes to find and fix a large number of bugs, the more work developers must do [71]. We measure the time between a reporter triaging a bug report and another reporter marking it as a multiple problem (s). For non-triaged bug reports, we record the days between reporting and being recognized as a multiple bug. Given a corpus of bug issue reports, the more time (in days) it takes to identify and triage bug reports by different reports, the more developers' effort is required and vice versa. Identification duration (time) is directly proportional to the developers' effort required. For example, given a report **X** that was first reported by reporter A on day one (d_1) at a given time (t_1), and the same bug by reporter B on day four (d_4) at time (t_4), and lastly, the same bug by reporter C on day seven (d_7) at time (t_7). Assuming that the release date for the given version of the open-source project is given as Day 0 at time (t_0), Hence, from the above example, the total duration needed to identify bug X as a multiple bug by each of the reporters is given as follows:

$$Identification\_duration\_by\_ReporterA = d_0 - d_1 \,@\, t_0 - t_1$$

$$Identification\_duration\_by\_ReporterB = d_0 - d_4 \,@\, t_0 - t_4$$

$$Identification\_duration\_by\_ReporterC = d_0 - d_7 \,@\, t_0 - t_7$$

In general, the identification duration by a reporter is given by $d_0 - d_i \,@\, t_0 - t_j$. Where d_0 denotes the first day a reporter identified and reported the given bug X at the initial time, t_0 d_i denotes the i^{th} subsequent day a different reporter also identified and reported the same bug X at the j^{th} time t_j. Thus, the timestamp for a given bug to be identified and reported by a reporter is $d_i \,@\, t_j$.

From the above example, it can be deduced that the average duration for a given bug to be classified as multiple bugs is as follows:

$$Mean\_identification\_duration = \frac{(|\boldsymbol{d_0 - d_1 @ t_0 - t_1}|)+(|\boldsymbol{d_0 - d_4 @ t_0 - t_4}|)+(|\boldsymbol{d_0 - d_7 @ t_0 - t_7}|)}{\boldsymbol{3}} \quad (11)$$

We can deduce from the above example, that the general equation for estimating the average effort required for identifying a given bug X to be classified as a multiple bug by at least two different reporters is defined as follows:

$$Mean\_identification\_duration = \frac{\sum_{i,j=1}^{n}(|\boldsymbol{d_0 - d_i @ t_0 - t_j}|)}{\boldsymbol{n}} \quad (12)$$

where n denotes the number of different reports identifying and reporting the given bug X, i denotes the i^{th} day a given bug is identified and reported by a given reporter at a j^{th} time. Since the identification duration is directly proportional to the developers' effort needed to address a reported bug [70]. Then we can deduce that the *mean_identification_duration*= ($\frac{\sum_{i,j=1}^{n}(|\boldsymbol{d_0 - d_i @ t_0 - t_j}|)}{\boldsymbol{n}}$ |) (13), can be used as the average mean of resolving the identified multiple bug X. Thus, we assume that the developers' effort needed to resolve the reported bug will be at most the duration needed for identifying the given bug.

That is, *mean_developers'_effort* ≤ *mean_identification_duration.*

Note that, this assumption is based on experienced developers resolving the identified bug.

3.10.1 Path Extraction

We extract paths from an AST built for our method instead of straight source code, because an AST usually helps in capturing and improving code features [72]. Through the clear illustration of features through AST, the technique can make dissimilarity much better between different bug reports and, thus, non-bug code structures [72]. We used the widely-recognized Eclipse JDT

package to build an AST for a given Java method [23]. For example, we do not want to lose important or relevant data for each technique for bug detection. We find the smallest figure of four long pathways that can safeguard an AST model's nodes. So, we created an AST and extracted the four long pathways. In an AST, an extended path connects two leaf nodes via the root node, with multiple overlapping nodes [23] (Figure 4).

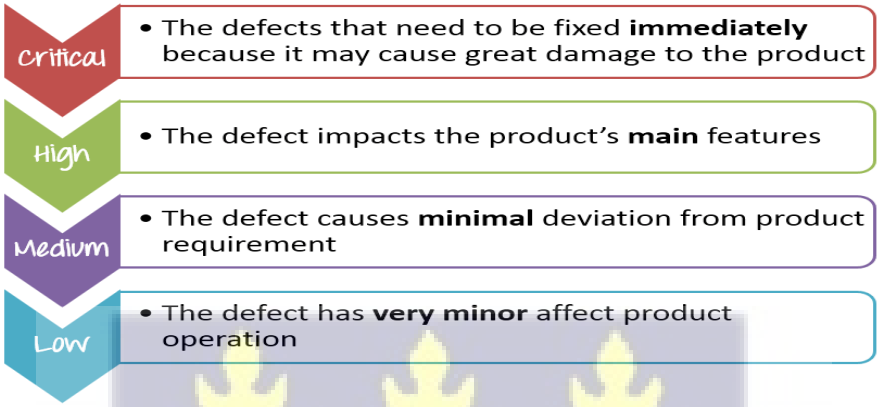


Figure 4. Severity levels of bugs

3.11 Bugs detection stages

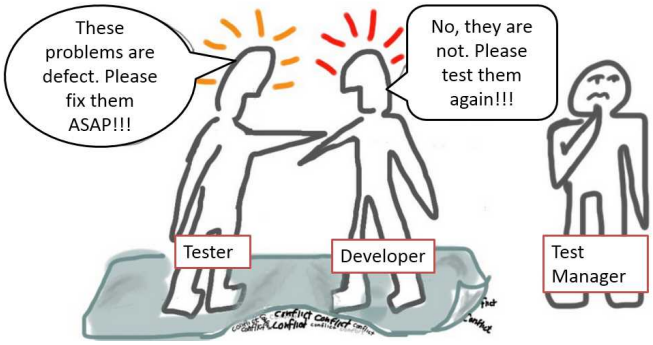
Before a bug can go through its detection and fixation cycle, the user-involved must suspect or detect and discover the anomaly or malfunctioning of a software project [73]. Then, by using software language, the user then categories the anomalies into single or multiple bugs or the categories that may be available based on the type of operation. The user has to inform the test manager or IT personnel who put in place the necessary executable actions to resolve the problem. After the resolution, the test manager has to verify if the bugs are truly resolved or not. If the resolution fails, the action has to be re-launched again. But if the results verify that the bugs have been fixed, the dataset repository is closed and the report is transmitted to the central system. This is shown in Figure 5.

Figure 5. Bugs detection cycle

3.12 Effort Collection Requirements to Improve Bug Detection Process

Data for effort estimation should be gathered based on defined stages and activities [74]. If there is non-homogeneity in the data collection of the effort variable across projects, the overall dataset effort provided spans a variety of phase combinations, causing bug analysis to be erroneous [74]. Thorough planning, such as establishing shorter activities and smaller tasks, improves estimating accuracy and minimizes the extent of estimation errors. To avoid "noisy" information in the effort value, unrelated activities conducted during project execution should be pushed out of the analysis.

Time spent on software projects, as well as time spent on unrelated tasks, should not be included in the effort collecting tool, because the quantity of functionality offered may be directly tied to various aspects of effort estimation [75]. However, it is not possible to construct a relationship with the same capabilities for other types of effort, such as documentation, meetings, and demos. The amount of time spent on multiple projects and unrelated tasks should be differentiated.



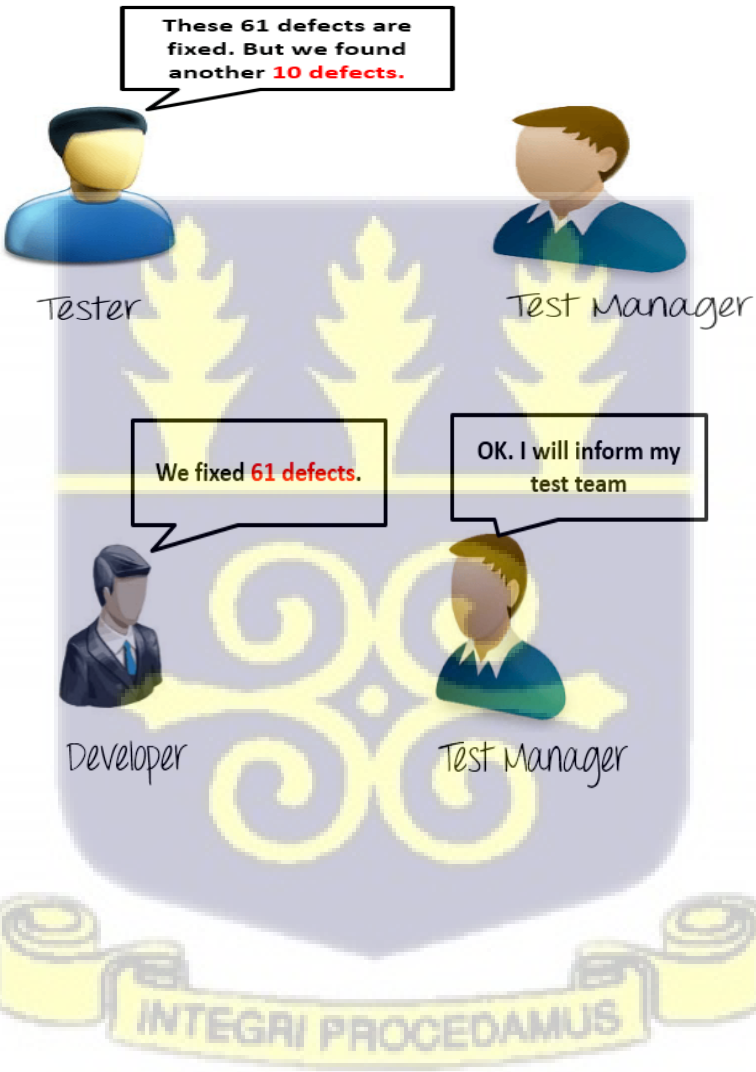


Figure 6. Bugs detection process

3.13 Processes of Dataset Framework

The dataset consists of five processes. Software managers are in charge of providing a high-level overview of all bug estimation methods [3]. The dataset execution level is where all of the data analysis and calibration stages are carried out. The data analysis process uses the effort collection results and BFC size measurements of all the projects for analysis [76]. Then it produces "effort models" for each application type. Other important data collected during project implementation is analyzed for the final effort estimation of the new dataset. The calibration process is carried out regularly at the dataset implementation level [76]. This method is used to do two distinct calibration processes. These are established frameworks for describing bug detection stages, activities, unexpected events, and unplanned events. The Measurement and Analysis Group determines the calibration period primarily based on the rate of arrival of available data.

Figure 7. Multiple bugs detection and fixing algorithm

3.14 Bug Detection and Evaluation Tools

The process of running test cases and assessing the results is known as bug discovery and evaluation. Selecting test cases for implementation, setting up the environment, running the selected tests, recording the execution activities, analyzing probable product failures, and assessing the effort's effectiveness are all part of this process. Execution tools are primarily focused on making running tests easier.

1. Capture and playback tools
2. Memory testing tools
3. Monitoring tools
4. System log reporting tools
5. Coverage analysis tools
6. Mapping tools.

Implementation tools are also used for test execution. Implementation tools take the place of software or hardware that interacts within the system with the dataset to be tested.

3.15 Chapter Summary

The chapter presented the experimental methodology of the study. It includes the descriptions of the two datasets used for the study as well as the description of the framework. It also described the experimental design, which included a discussion of deep learning and traditional machine learning algorithms in detecting multiple bugs. Again, it also provided an effort estimation framework for estimating the effort required for detecting multiple bugs.

CHAPTER FOUR

EXPERIMENTAL RESULT AND DISCUSSIONS

4.1 Review of Existing Techniques for Bug Detection after Inclusion/Exclusion Criteria

The search from the review of existing studies from four electronic databases identified fifty (50) papers after inclusion and exclusion criteria. Figure 8 depicts a clustered bar graph showing both the original number of papers sourced from the four electronic databases and the respective number after the inclusion and exclusion criteria. In Figure 9, twenty-seven (27) of the primary studies, representing 63% of the total, were obtained from ScienceDirect. The majority of these papers were obtained from the Science Direct database, which contains more journals dedicated to the study than the other three databases (IEEE Xplore, Springer, and ACM Digital Library). Fourteen (14) of the primary studies were obtained from the ACM digital library, accounting for 32% of the total, eight (8) from IEEE Xplore, accounting for 3%, and one (1) from Springer, accounting for 2% (Figure 9).

4.2 Results from Techniques for Multiple Bug Detection

The proposed bidirectional LSTM model was tested on a dataset from two separate projects, namely, Mozilla Firefox and Eclipse, each with a different number of multiple bug reports, as indicated in Table 2. The proposed model was compared using existing machine learning techniques. The model was trained and tested with more than 1000 bug reports.

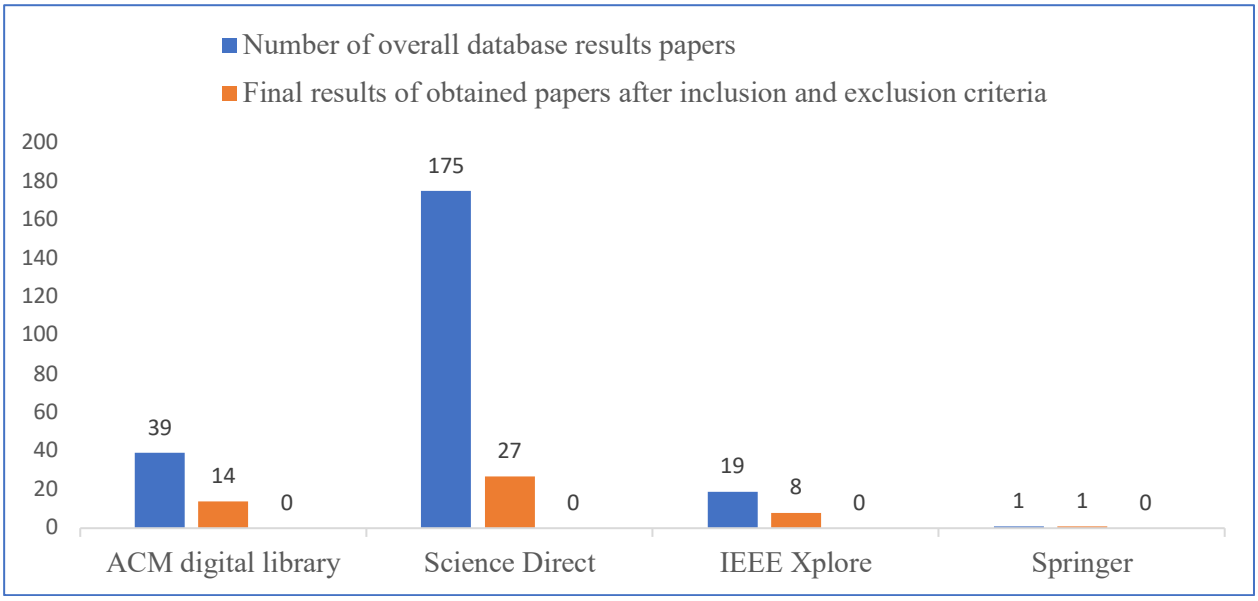


Figure 8. Search results of overall papers and final papers after inclusion and exclusion criteria

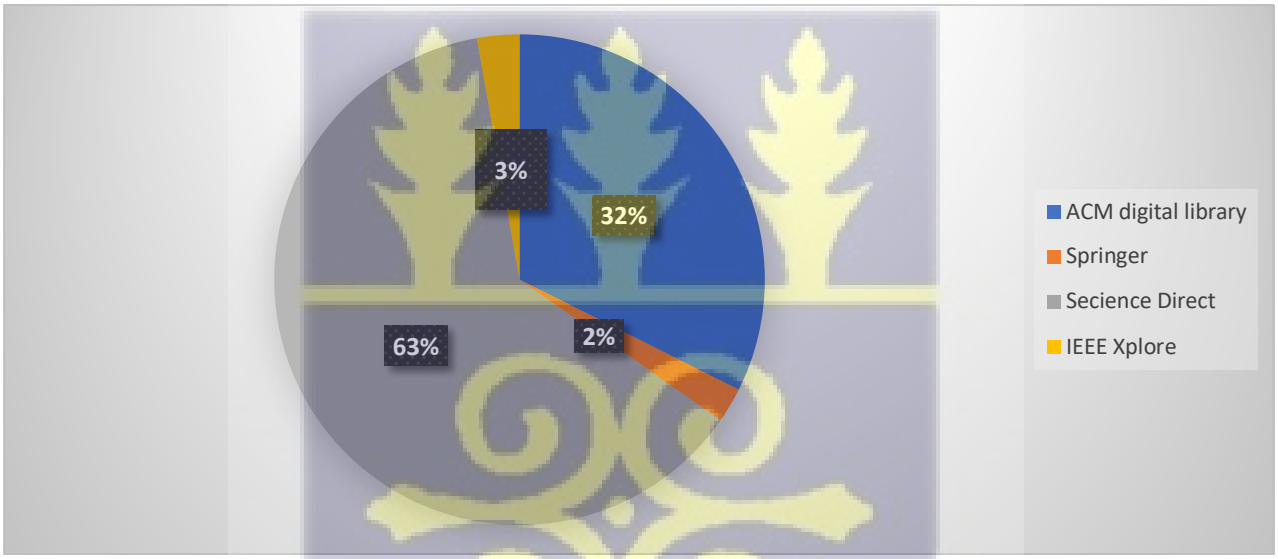


Figure 9. Final papers obtained after inclusion and exclusion criteria

4.3 Research questions

Examining the proposed framework requires answering the following questions:

- *What are the existing techniques for detecting duplicate bugs?*
- *What are the techniques for detecting multiple bugs?*
- *What is the mean identification duration for estimating multiple bugs?*

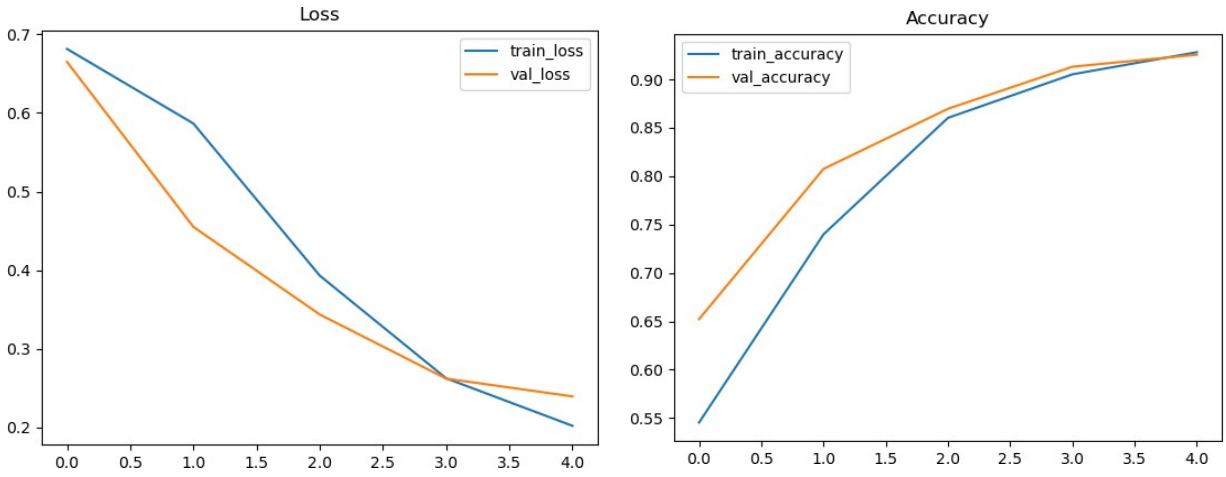
The research questions compare the selected deep neural network (BiLSTM) against other alternatives, and it also examines the performance improvement of the proposed model as indicated in Tables 6, 7, and 8.

4.4 Bidirectional LSTM neural network

After the training epoch, the experiments were conducted on a bidirectional LSTM neural network. The bidirectional LSTM neural network model was created in Python with the help of the Keras deep learning library [3]. Accuracy and loss for validation in the Keras model may vary according to the study's specific circumstances. The epoch is inversely related to the loss, which means that as the epoch length increases, the loss decreases and the accuracy increases. The following scenarios are evaluated for keras validation loss and accuracy:

- When the validation loss increases while the accuracy decline, the model is said to be not learning.
- When both the validation loss and the accuracy increase, the model is said to be overfitting.
- When the validation loss begins to decline and the accuracy begins to increase, the model is considered to be learning.

Figure 10 (i) and 10 (ii) describe the accuracy and loss metrics, which are normally considered to be the return on the variance between the training and testing data.



(i) Model loss (ii) Model accuracy

Figure 10. Comparison between training and validation (loss and accuracy)

Figures 10(i) and 10(ii) show that the model has a higher training accuracy but a lower validation accuracy, implying that it is likely learnable. The training loss is found to be decreasing, which shows that the model is learning to recognize the training set.

4.5 Comparison between performance results from applying BiLSTM, CNN, LSTM, SVM, and Random Forest

After training and testing, the accuracy, precision, recall, and F1 score metrics were used to determine the performance of both traditional machine learning and deep learning models. Table 6 shows the overall performance of results made from Support Vector Machines (SVM), Random Forest, Convolutional Neural Network (CNN), (LSTM), and the proposed BiLSTM model, performed on the Mozilla Firefox dataset. Table 7 shows the overall performance results on the Eclipse dataset and compares them with the proposed BiLSTM model. On average, the proposed model performed better in accuracy as well as precision across the Mozilla dataset.

Table 6. Performance of the models based on the Mozilla Firefox dataset

Model	Accuracy	Precision	Recall	F1 score
Random Forest	0.6789	0.4587	0.2334	0.3094
SVM	0.6838	0.5088	0.0547	0.0989
CNN	0.6221	0.3941	0.3589	0.3757*
LSTM	0.6609	0.4341	0.2316	0.3021
Bidirectional LSTM	0.7109*	0.6830*	0.4570*	0.0855

** denotes the best model across a given performance metric – Accuracy, Precision, Recall, F1 score*

Table 6 shows the comparative results when the models were applied to the Mozilla Firefox dataset. Once again, the proposed bidirectional LSTM model outperformed the LSTM, CNN, SVM, and Random Forest, with an accuracy of 71.09%, precision of 68.30%, and a recall of 45.7%. Despite the fact that the bidirectional LSTM model had the highest precision rate of 68.30%, there were only a few accurate codes classified as incorrect. The 45.7% recall rate indicated that there were few false positives. As discussed before, the F1-score is the vocal mean of the recall and precision ratios and is a key metric to describe model performance. As shown in Table 6, the F1-score of the BiLSTM model was average among the four models, showing that the model produced more true positives with a low percentage of false positives.

Table 7 shows the proportional results when the models were applied to the Eclipse dataset. The proposed BiLSTM and CNN proved the best as compared to LSTM, SVM, and Random Forest

Table 7. Performance of the models based on the Eclipse dataset

Model	Accuracy	Precision	Recall	F1 score
Random Forest	0.8123	0.2953	0.0692	0.1122
SVM	0.8203	0.5000*	0.0027	0.0054
CNN	0.7353	0.2219	0.2129*	0.2173
LSTM	0.8014	0.2721	0.0897	0.1350
Bidirectional LSTM	0.8260*	0.4225	0.0204	0.5089*

** denotes the best model across a given performance metric – Accuracy, Precision, Recall, F1 score*

4.6 Comparison between BiLSTM, LSTM, CNN, SVM and Random Forest results

As shown in Figure 11 and 12, the results of our experiments show that the proposed framework based on bidirectional LSTM neural networks appropriately detects multiple bugs in software projects and that performance can be significantly increased when compared to LSTM, CNN, SVM, and Random Forest. Based on Table 8, and Figure 11, 12, we draw the following conclusions: The proposed model achieves a minor performance improvement. We calculated bidirectional LSTM improvement and compared it to other machine learning techniques such as LSTM, CNN, SVM, and Random Forest. Accuracy results show a 3% improvement for bidirectional LSTM compared with SVM, a 9% improvement accuracy for bidirectional LSTM compared with CNN, a 5% improvement accuracy for bidirectional LSTM compared with LSTM, and a 4% improvement accuracy for bidirectional LSTM compared with Random Forest. Precision values improved by 17% for bidirectional LSTM when compared to SVM, 29% improvement for bidirectional LSTM when compared to CNN, 25% improvement for bidirectional LSTM when compared to LSTM, and 23% improvement for bidirectional LSTM when compared to Random Forst. Recall values of the proposed model improved 39.6% compared to SVM, 9% improvement compared to CNN, 22% improvement compared to LSTM, and 22% improvement compared to Random Forest, which shows that bidirectional LSTM model outperforms the other techniques in detecting multiple bugs in software project.

Table 8. BiLSTM improvement compared to SVM, CNN, LSTM and Random Forest

	SVM, BiLSTM, Improv.			CNN, BiLSTM, Improv.			LSTM, BiLSTM, Improvement			Random(F), BiLSTM, Improv.		
Accuracy	0.68,	0.71	3%	0.62,	0.71	9%	0.66,	0.71,	5%	0.67,	0.71,	4%
Precision	0.51,	0.68	17%	0.39,	0.68	29%	0.43,	0.68,	25%	0.45,	0.68	23%
Recall	0.054,	0.45	39.6%	0.36,	0.45	9%	0.23,	0.45,	22%	0.233	0.45	22%

Figures 11 and 12 shows comparison between BiLSTM, CNN, SVM, LSTM and Random Forest results.

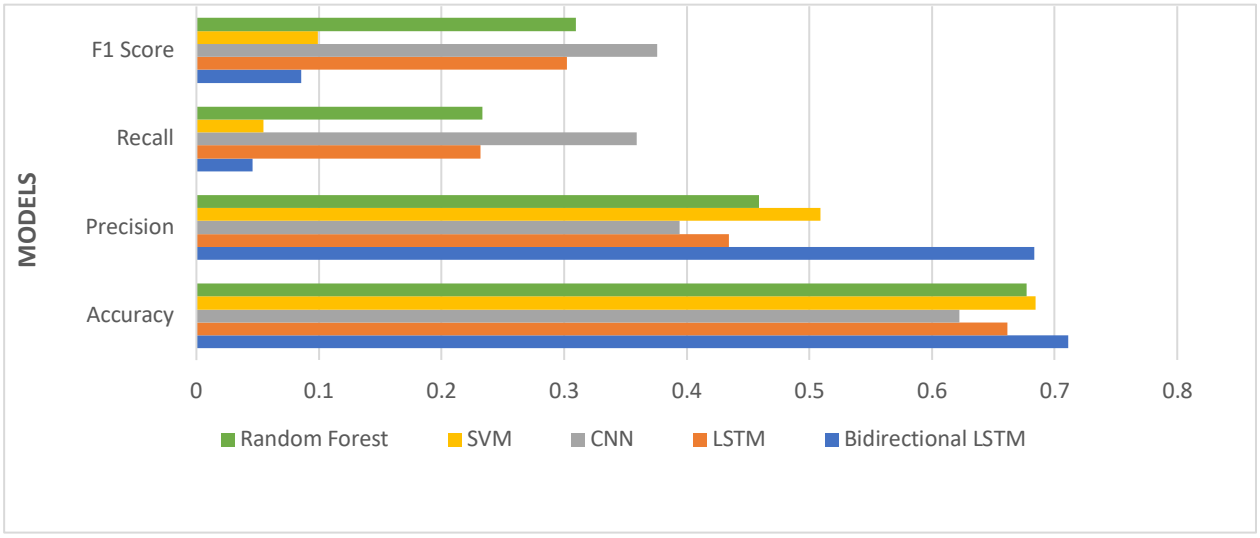


Figure 11. Comparison between BiLSTM, CNN, SVM, LSTM and Random Forest results in Mozilla Firefox dataset

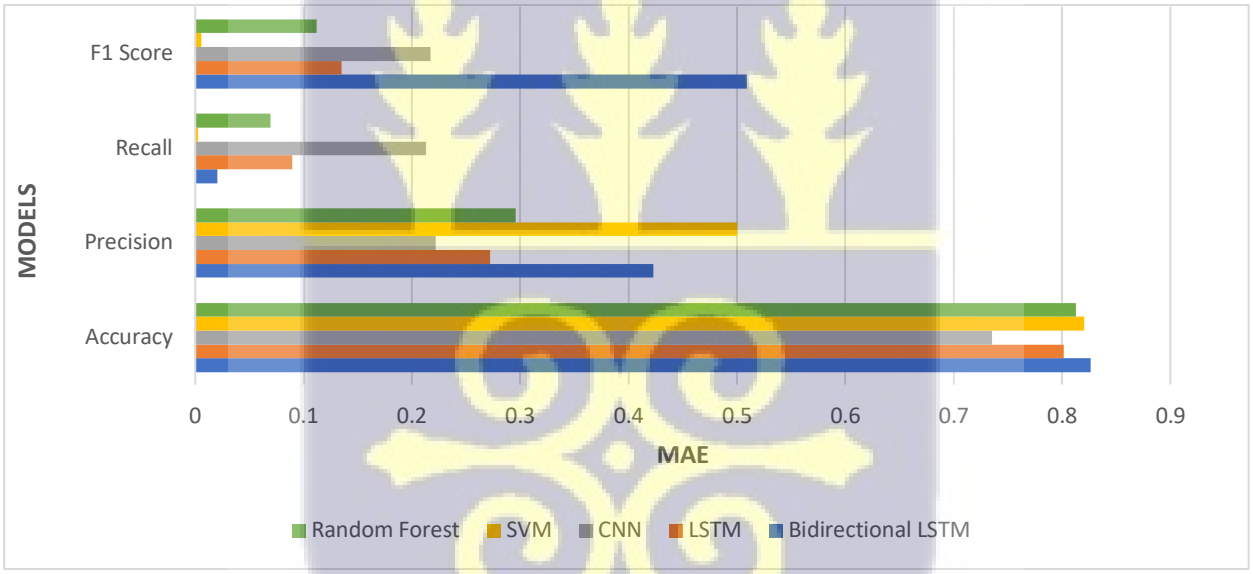


Figure 12. Comparison between BiLSTM, CNN, SVM, LSTM and Random Forest results in Eclipse dataset

4.7 Mean Identification Duration/Estimation Results for Multiple Bugs

Table 9 shows the mean identification duration results obtained from 10 sampled bug reports from the entire Mozilla Firefox dataset. Here it shows one bug, i.e., the referenced bug identified and reported by a user and the reports made by three different users (reports 1, 2, and 3) on the referenced bug and the dates those bugs were reported.

Table 9. Mean identification duration for multiple bug reports (Mozilla Firefox)

Bug ID	Reporter 1 ID	Reporter 2 ID	Reporter 3 ID	Ref Date	Date 1	Date 2	Date 3	Duration 1	Duration 2	Duration 3	Mean Duration
14871	269442	143180	515027	9/24/1999	11/12/2004	5/8/2002	9/7/2009	1876	957	3636	2156.3
153509	340604	610714	334876	6/21/2002	6/6/2006	11/9/2010	4/20/2006	1446	3063	1399	1969.3
513391	475603	505371	506588	8/28/2009	1/27/2009	7/20/2009	7/26/2009	213	39	33	95
513401	495999	482190	540641	8/28/2009	6/2/2009	3/9/2009	1/19/2010	87	172	144	134.3
513415	447571	427280	529958	8/28/2009	7/22/2008	4/5/2008	1/15/2010	402	510	140	350.6
513424	501115	457572	505298	8/28/2009	6/29/2009	9/28/2008	7/20/2009	60	334	39	144.3
513459	465673	403402	493606	8/29/2009	11/18/2008	11/11/2007	5/18/2009	284	657	103	348
516003	494373	502405	502185	9/11/2009	5/22/2009	7/4/2009	7/3/2009	112	69	70	83.6
516108	64488	121839	176312	9/11/2009	2/5/2001	1/25/2001	10/23/2002	3140	3151	2515	2935.3
516115	64488	187930	190785	9/11/2009	1/5/2001	1/6/2003	1/27/2003	3171	2440	2419	2676.6
Average Identification duration								1079.1	1139.2	10499	1089.4

NB: Duration in Days

Table 10. Mean identification duration for multiple bug reports (Eclipse)

Bug ID	Reporter 1 ID	Reporter 2 ID	Reporter 3 ID	Ref Date	Date 1	Date 2	Date 3	Duration 1	Duration 2	Duration 3	Mean Duration
108813	138959	141014	155890	9/6/2005	4/27/2006	5/10/2006	8/31/2006	233	246	359	279.3
135956	139047	138154	138301	4/10/2006	4/27/2006	4/23/2006	4/24/2006	17	13	14	14.7
5835	177407	27381	45659	11/7/2002	3/14/2007	11/29/2002	10/28/2003	1588	22	355	655
21383	49261	72982	103659	7/8/2002	12/22/2003	8/31/2004	7/13/2005	532	785	1101	806
47927	49277	47960	48254	12/2/2003	12/22/2003	12/3/2003	12/8/2003	20	1	6	9
293551	313480	294392	296556	9/10/2028	5/19/2010	11/5/2009	12/1/2009	6689	6884	6858	6810.3
38821	49293	51392	41677	6/12/2003	12/22/2003	2/9/2004	8/19/2003	193	242	68	167.7
39409	49333	41052	44098	6/27/2003	12/23/2003	8/1/2003	10/2/2003	179	35	97	103.7
29923	312103	144234	285735	1/21/2003	5/7/2010	5/29/2006	9/8/2005	2663	1224	961	1616
241155	313406	249677	264043	8/7/2016	10/5/2018	8/10/2004	2/7/2009	789	4380	2738	2635.7
Average Identification duration								1290	1383	1255.7	1309.7

The duration from the date the first or reference reporter makes the report to the date the other reporters also make the same report is also taken into consideration. And here, with all the information extracted, the mean identification duration is then calculated. The mean identification duration is directly proportional to the estimation duration for resolving the multiple bugs. A similar result is shown for the Eclipse dataset in Table 10.

4.8 Chapter Summary

This section presents the results of the empirical experiment. It includes a review of various existing techniques for bug detection. The chapter also presents the results of traditional machine learning algorithms and deep learning algorithms in detecting multiple bugs from two open-source bug repositories. Also, results of the novel metric for estimating the effort of detecting multiple bugs in open-source projects are presented. That is, the mean identification duration results, performed on 10 sampled bug reports from the Mozilla Firefox and Eclipse bug repositories, respectively, are presented.

4.9 Threats to validity

Some features, like other studies, may affect the proposed model's performance. Here are the dangers to our study's validity. Internal validity refers to bidirectional LSTM implementation and no other approaches. We select bidirectional LSTM, because others have shown to be effective for feature extraction and text categorization [3]. The findings are checked to avoid errors and mistakes

External validity makes it hard to simplify the results. As stated earlier the used dataset extracted from bug reports related to two open-source projects. Dataset that has been used already from other projects; it is not certain to reach the same results.

CHAPTER FIVE

CONCLUSION

5.1 Conclusion

This thesis provides a framework for multiple bug detection and effort estimation metrics for open-source projects to reduce the time and effort software testers spend on analyzing bug reports while also improving software reliability and productivity. A bidirectional LSTM effort estimation model based on a deep learning approach is proposed for multiple bug detection and effort estimation for open-source projects. The proposed framework involves the extraction of relevant features, text preprocessing (lemmatization, stemming, and stopwords) and then the extraction of relevant keywords from bug report descriptions. We extracted a dataset from the Bugzilla bug tracking system, which is comprised of two open-source projects and contains more than 1,000 bug reports. The dataset was divided into training and test examples, and the dataset's variation was recorded (90% training, 10% test,).

The proposed model was validated using two publicly available datasets, Eclipse and Mozilla Firefox. The experimental results were evaluated using accuracy, precision, recall, and the F1 score. The proposed Bidirectional LSTM model is compared to four well-known machine learning algorithms: Random Forest, SVM, CNN, and LSTM. The proposed model outperformed the benchmark models in terms of accuracy **(62.60-71.09%)** and precision (**50.0–68.30%).** The proposed model outperforms studies by Sawarkar et al. [44],**(52.42-64.61%),** Shokripour et al. [2], **(44.01-59.76 %)** and Sharma et al.[38],**(68.20-70.00%).**The precision rate of the proposed system improves more than the Mani et al. [40],**(43.10-47.00%),** and Mahmood et al. [45],**(28.32-42.25%)** systems respectively.

5.2 Recommendation and Future Work

We measure effort using three metrics: identification duration, the number of people involved, and identification delay [70]. These metrics do not normally capture the work required to find and detect multiple bug reports. There may be additional ways to quantify the work required to detect multiple bugs in software projects. In future case studies, we intend to investigate other indicators. We plan to test other deep learning techniques on closed-source projects like Maharah and Hashfood. These covers testing the performance of different classifiers such as KNN, SVM, and Random Forest. Integration of the proposed framework with Bugzilla software may be a future work direction.

REFERENCES

[1] M. Suresh, M. Amarnath, G. Baranikumar, M. Jagadheeswaran, and B. Tech, "a Survey on Bug Tracking System for Effective Bug Clearance," *Int. Res. J. Eng. Technol.*, pp. 1622–1630, 2016, [Online]. Available: www.irjet.net.

[2] D. G. Lee and Y. S. Seo, "Systematic review of Bug report processing techniques to improve software management performance," *J. Inf. Process. Syst.*, vol. 15, no. 4, pp. 967–985, 2019, doi: 10.3745/JIPS.04.0130.

[3] H. Bani-Salameh, M. Sallam, and B. Al shboul, "A deep-learning-based bug priority prediction using RNN-LSTM neural networks," *E-Informatica Softw. Eng. J.*, vol. 15, no. 1, pp. 29–45, 2021, doi: 10.37190/E-INF210102.

[4] A. Kumar, M. Madanu, H. Prakash, L. Jonnavithula, and S. R. Aravilli, "Advaita: Bug Duplicity Detection System," no. August, pp. 0–5, 2020, [Online]. Available: http://arxiv.org/abs/2001.10376.

[5] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," p. 291, 2002, doi: 10.1145/581376.581377.

[6] J. E. Corter, Y. J. Rho, D. Zahner, J. Nickerson, and B. Tversky, "Bugs and Biases: Diagnosing Misconceptions in the Understanding of Diagrams," *Proc. 31st Annu. Conf. Cogn. Sci. Soc.*, pp. 756–761, 2009, [Online]. Available: http://csjarchive.cogsci.rpi.edu/Proceedings/2009/papers/133.

[7] I. Gomes, P. Morgado, T. Gomes, and R. Moreira, "An overview on the Static Code Analysis approach in Software Development," *Fac. Eng. da Univ. do Porto, Port.*, p. 16,

2009, [Online]. Available: https://pdfs.semanticscholar.org/ce3c/584c906eea668954f6a1a0ddbb295c6ec5a2.pdf%0A http://paginas.fe.up.pt/~ei05021/TQSO - An overview on the Static Code Analysis approach in Software Development.pdf.

[8] M. J. Coblenz, "User-Centered Design of Principled Programming Languages," no. August, 2020.

[9] A. Goyal and N. Sardana, "Performance Assessment of Bug Fixing Process in Open Source Repositories," *Procedia Comput. Sci.*, vol. 167, no. Iccids 2019, pp. 2070–2079, 2020, doi: 10.1016/j.procs.2020.03.247.

[10] J. Xuan *et al.*, "Towards effective bug triage with software data reduction techniques," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 1, pp. 264–280, 2015, doi: 10.1109/TKDE.2014.2324590.

[11] W. Xiaoyin, Z. Lu, X. Tao, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," *Proc. - Int. Conf. Softw. Eng.*, no. January, pp. 461–470, 2008, doi: 10.1145/1368088.1368151.

[12] W. Xiaoyin, Z. Lu, X. Tao, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," *Proc. - Int. Conf. Softw. Eng.*, pp. 461–470, 2008, doi: 10.1145/1368088.1368151.

[13] K. R. Subramanian, "Analytical Skills – The Wake up Call for Human Resource," *Int. J. Eng. Manag. Res.*, vol. 7, no. 1, pp. 14–19, 2017.

[14] G. Decker and J. Mendling, "Process instantiation," *Data Knowl. Eng.*, vol. 68, no. 9, pp.

777–792, 2009, doi: 10.1016/j.datak.2009.02.013.

[15] P. Mell, V. Hu, R. Lippmann, J. Haines, and M. Zissman, "An overview of issues in testing intrusion detection systems," *Contract*, p. 22, 2003, [Online]. Available: https://pdfs.semanticscholar.org/e87b/4ea56a5cc4e985158c8eaff13ff1c5e0828b.pdf%5Cn http://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir7007.pdf%5Cnhttp://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.8.5163.

[16] A. Nistor, P. C. Chang, C. Radoi, and S. Lu, "CARAMEL: Detecting and fixing performance problems that have non-intrusive fixes," *Proc. - Int. Conf. Softw. Eng.*, vol. 1, pp. 902–912, 2015, doi: 10.1109/ICSE.2015.100.

[17] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *Proc. Conf. Object-Oriented Program. Syst. Lang. Appl. OOPSLA*, no. June, pp. 132–135, 2004, doi: 10.1145/1028664.1028717.

[18] A. Kloptchenko, *Text Mining Based on the Prototype Matching Method the Prototype Matching Method*, no. 47. 2003.

[19] M. Pradel and K. Sen, "DeepBugs: a learning approach to name-based bug detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 1–25, 2018, doi: 10.1145/3276517.

[20] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyande, "AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations," *SANER 2019 - Proc. 2019 IEEE 26th Int. Conf. Softw. Anal. Evol. Reengineering*, pp. 456–467, 2019, doi: 10.1109/SANER.2019.8667970.

[21] N. Ayewah, D. Hovemeyer, D. J. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Softw.*, vol. 25, no. 5, pp. 22–29, 2008, doi: 10.1109/MS.2008.130.

[22] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior," *ACM SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 57–72, 2001, doi: 10.1145/502059.502041.

[23] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 1–30, 2019, doi: 10.1145/3360588.

[24] A. Kumar, M. Madanu, H. Prakash, L. Jonnavithula, and S. R. Aravilli, "Advaita: Bug Duplicity Detection System," pp. 1–5, 2020, [Online]. Available: http://arxiv.org/abs/2001.10376.

[25] S. Koch, "Effort modeling and programmer participation in open source software projects," *Inf. Econ. Policy*, vol. 20, no. 4, pp. 345–355, 2008, doi: 10.1016/j.infoecopol.2008.06.004.

[26] S. Panthaplackel, J. J. Li, M. Gligoric, and R. J. Mooney, "Deep Just-In-Time Inconsistency Detection Between Comments and Source Code," 2020, [Online]. Available: http://arxiv.org/abs/2010.01625.

[27] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," *Oper. Syst. Rev.*, vol. 35, no. 5, pp. 57–72, 2001, doi: 10.1145/502059.502041.

[28] N. Cooper, C. Bernal-Cardenas, O. Chaparro, K. Moran, and D. Poshyvanyk, "A Replication Package for It Takes Two to Tango: Combining Visual and Textual Information for Detecting Duplicate Video-Based Bug Reports," *Proc. - Int. Conf. Softw. Eng.*, no. Cv, pp. 160–161, 2021, doi: 10.1109/ICSE-Companion52605.2021.00067.

[29] J. Zou, L. Xu, M. Yang, X. Zhang, J. Zeng, and S. Hirokawa, "Automated duplicate bug

report detection using multi-factor analysis," *IEICE Trans. Inf. Syst.*, vol. E99D, no. 7, pp. 1762–1775, 2016, doi: 10.1587/transinf.2016EDP7052.

[30] A. Kukkar, R. Mohana, Y. Kumar, A. Nayyar, M. Bilal, and K. S. Kwak, "Duplicate Bug Report Detection and Classification System Based on Deep Learning Technique," *IEEE Access*, vol. 8, no. November, pp. 200749–200763, 2020, doi: 10.1109/ACCESS.2020.3033045.

[31] T. Chappelly, C. Cifuentes, P. Krishnan, and S. Gevay, "Machine learning for finding bugs: An initial report," *MaLTeSQuE 2017 - IEEE Int. Work. Mach. Learn. Tech. Softw. Qual. Eval. co-located with SANER 2017*, pp. 21–26, 2017, doi: 10.1109/MALTESQUE.2017.7882012.

[32] M. Thorstensson, *Using Observers for Model Based Data Collection in Distributed Tactical Operations*, vol. Licentiate, no. 1386. 2008.

[33] F. Brown, D. Stefan, and D. Engler, "Sys: A static/symbolic tool for finding good bugs in good (browser) code," *Proc. 29th USENIX Secur. Symp.*, pp. 199–216, 2020.

[34] L. P. Binamungu, "Detecting and Correcting Duplication in Behaviour Driven Development," 2020.

[35] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, "An empirical study on the correctness of formally verified distributed systems," *Proc. 12th Eur. Conf. Comput. Syst. EuroSys 2017*, pp. 328–343, 2017, doi: 10.1145/3064176.3064183.

[36] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," *Proc. - Int. Conf. Softw. Eng.*, no. November, pp. 499–508,

2007, doi: 10.1109/ICSE.2007.32.

[37] A. Baarah, A. Aloqaily, Z. Salah, M. Zamzeer, and M. Sallam, "Machine learning approaches for predicting the severity level of software bug reports in closed source projects," *Int. J. Adv. Comput. Sci. Appl.*, vol. 10, no. 8, pp. 285–294, 2019, doi: 10.14569/ijacsa.2019.0100836.

[38] H. A. Ahmed, N. Z. Bawany, and J. A. Shamsi, "Capbug-a framework for automatic bug categorization and prioritization using nlp and machine learning algorithms," *IEEE Access*, vol. 9, no. April, pp. 50496–50512, 2021, doi: 10.1109/ACCESS.2021.3069248.

[39] S. F. A. Zaidi, F. M. Awan, M. Lee, H. Woo, and C. G. Lee, "Applying Convolutional Neural Networks with Different Word Representation Techniques to Recommend Bug Fixers," *IEEE Access*, vol. 8, no. November, pp. 213729–213747, 2020, doi: 10.1109/ACCESS.2020.3040065.

[40] T. Thongtan and T. Phienthrakul, "余弦相似度分类," pp. 407–414, 2019, [Online]. Available: https://www.aclweb.org/anthology/papers/P/P19/P19-2057/.

[41] S. Gupta and S. K. Gupta, "A Systematic Study of Duplicate Bug Report Detection," *Int. J. Adv. Comput. Sci. Appl.*, vol. 12, no. 1, pp. 578–589, 2021, doi: 10.14569/IJACSA.2021.0120167.

[42] A. Yadav and S. K. Singh, "Survey Based Classification of Bug Triage Approaches," *APTIKOM J. Comput. Sci. Inf. Technol.*, vol. 1, no. 1, pp. 1–11, 2016, doi: 10.34306/csit.v1i1.37.

[43] T. W. W. Aung, Y. Wan, H. Huo, and Y. Sui, "Multi-triage: A multi-task learning

framework for bug triage," *J. Syst. Softw.*, vol. 184, p. 111133, 2022, doi: 10.1016/j.jss.2021.111133.

[44] Ö. Köksal and B. Tekinerdogan, "Automated classification of unstructured bilingual software bug reports: An industrial case study research," *Appl. Sci.*, vol. 12, no. 1, 2022, doi: 10.3390/app12010338.

[45] Y. Mahmood, N. Kama, A. Azmi, A. S. Khan, and M. Ali, "Software effort estimation accuracy prediction of machine learning techniques: A systematic performance evaluation," *Softw. - Pract. Exp.*, 2021, doi: 10.1002/spe.3009.

[46] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," *Proc. Eur. Conf. Softw. Maint. Reengineering, CSMR*, pp. 385–390, 2012, doi: 10.1109/CSMR.2012.48.

[47] S. N. Ahsan, J. Ferzund, and F. Wotawa, "Program file bug fix effort estimation using machine learning methods for OSS," *Proc. 21st Int. Conf. Softw. Eng. Knowl. Eng. SEKE 2009*, pp. 129–134, 2009.

[48] S. N. Ahsan, M. T. Afzal, S. Zaman, C. Gütel, and F. Wotawa, "Mining Effort Data From the Oss Repository of Developer'S Bug Fix Activity," *J. IT Asia*, vol. 3, no. 1, pp. 107–128, 2016, doi: 10.33736/jita.38.2010.

[49] S. Iqbal, R. Naseem, S. Jan, S. Alshmrany, M. Yasar, and A. Ali, "Determining Bug Prioritization Using Feature Reduction and Clustering with Classification," *IEEE Access*, vol. 8, no. August 2009, pp. 215661–215678, 2020, doi: 10.1109/ACCESS.2020.3035063.

[50] S. Gujral, G. Sharma, S. Sharma, and Diksha, "Classifying bug severity using dictionary based approach," *2015 1st Int. Conf. Futur. Trends Comput. Anal. Knowl. Manag. ABLAZE*

2015, no. February, pp. 599–602, 2015, doi: 10.1109/ABLAZE.2015.7154933.

[51] N. Pandey, D. K. Sanyal, A. Hudait, and A. Sen, "Automated classification of software issue reports using machine learning techniques: an empirical study," *Innov. Syst. Softw. Eng.*, vol. 13, no. 4, pp. 279–297, 2017, doi: 10.1007/s11334-017-0294-1.

[52] F. Porto, L. Minku, E. Mendes, and A. Simao, "A Systematic Study of Cross-Project Defect Prediction With Meta-Learning," 2018, [Online]. Available: http://arxiv.org/abs/1802.06025.

[53] A. Baarah, A. Aloqaily, Z. Salah, and E. Alshdaifat, "Sentiment-based machine learning and lexicon-based approaches for predicting the severity of bug reports," *J. Theor. Appl. Inf. Technol.*, vol. 99, no. 6, pp. 1386–1401, 2021.

[54] H. Valdivia-Garcia and A, "Understanding the Impact of Diversity in Software Bugs on Bug Prediction Models by A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computing and Information Sciences PhD Program in Computing an," *ProQuest*, 2016.

[55] V. R. Basili, "Models and Metrics for Software Management and Engineering.," *Instrumentation in the Pulp and Paper Industry, Proceedings*, vol. 1. pp. 278–289, 1980.

[56] A. Hindle, A. Alipour, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detection and ranking," *Empir. Softw. Eng.*, vol. 21, no. 2, pp. 368–410, 2016, doi: 10.1007/s10664-015-9387-3.

[57] M. Padmanaban and T. Bhuvaneswari, "An Approach Based on Artificial Neural Network for Data Deduplication," *Int. J. Comput. Sci. Inf. Technol.*, vol. 3, no. 4, pp. 4637–4644,

2012.

[58] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-Supervised Bug Detection and Repair," no. NeurIPS, 2021, [Online]. Available: http://arxiv.org/abs/2105.12787.

[59] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," *Proc. Int. Conf. Dependable Syst. Networks*, pp. 52–61, 2008, doi: 10.1109/DSN.2008.4630070.

[60] L. Wu, B. Xie, G. Kaiser, and R. Passonneau, "BugMiner: Software reliability analysis via data mining of bug reports," *SEKE 2011 - Proc. 23rd Int. Conf. Softw. Eng. Knowl. Eng.*, no. December, pp. 95–100, 2011.

[61] N. Ebrahimi, A. Trabelsi, M. S. Islam, A. Hamou-Lhadj, and K. Khanmohammadi, "An HMM-based approach for automatic detection and classification of duplicate bug reports," *Inf. Softw. Technol.*, vol. 113, no. May, pp. 98–109, 2019, doi: 10.1016/j.infsof.2019.05.007.

[62] J. Deshmukh, K. M. Annervaz, S. Podder, S. Sengupta, and N. Dubash, "Towards accurate duplicate bug retrieval using deep learning techniques," *Proc. - 2017 IEEE Int. Conf. Softw. Maint. Evol. ICSME 2017*, pp. 115–124, 2017, doi: 10.1109/ICSME.2017.69.

[63] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," *2012 27th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2012 - Proc.*, pp. 70–79, 2012, doi: 10.1145/2351676.2351687.

[64] N. Ebrahimi, A. Trabelsi, M. S. Islam, A. Hamou-Lhadj, and K. Khanmohammadi, "An HMM-based approach for automatic detection and classification of duplicate bug reports,"

Inf. Softw. Technol., vol. 113, pp. 98–109, 2019, doi: 10.1016/j.infsof.2019.05.007.

[65] D. Sharma, B. Kumar, and S. Chand, "A Survey on Journey of Topic Modeling Techniques from SVD to Deep Learning," *Int. J. Mod. Educ. Comput. Sci.*, vol. 9, no. 7, pp. 50–62, 2017, doi: 10.5815/ijmecs.2017.07.06.

[66] N. Bettenburg, S. Just, A. Schröter, C. Weiß, R. Premraj, and T. Zimmermann, "Quality of bug reports in Eclipse," no. May 2014, pp. 21–25, 2007, doi: 10.1145/1328279.1328284.

[67] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," *2012 27th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2012 - Proc.*, no. September 2015, pp. 70–79, 2012, doi: 10.1145/2351676.2351687.

[68] A. Alipour, A. Hindle, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detection," *IEEE Int. Work. Conf. Min. Softw. Repos.*, pp. 183–192, 2013, doi: 10.1109/MSR.2013.6624026.

[69] M. Rahman, Y. Watanobe, and K. Nakamura, "A Bidirectional LSTM Language Model for Code Evaluation and Repair," pp. 1–15, 2021.

[70] M. S. Rakha, W. Shang, and A. E. Hassan, "Studying the needed effort for identifying duplicate issues," *Empir. Softw. Eng.*, vol. 21, no. 5, pp. 1960–1989, 2016, doi: 10.1007/s10664-015-9404-6.

[71] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?," *Proc. - ICSE 2007 Work. Fourth Int. Work. Min. Softw. Repos. MSR 2007*, no. June 2007, 2007, doi: 10.1109/MSR.2007.13.

[72] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," *ACM SIGPLAN Not.*, vol. 53, no. 4, pp. 404–419, 2018, doi: 10.1145/3192366.3192412.

[73] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," *Proc. 2005 OOPSLA Work. Eclipse Technol. Exch. eclipse'05*, pp. 35–39, 2005, doi: 10.1145/1117696.1117704.

[74] E. Kocaguneli and T. Menzies, "Software effort models should be assessed via leave-one-out validation," *J. Syst. Softw.*, vol. 86, no. 7, pp. 1879–1890, 2013, doi: 10.1016/j.jss.2013.02.053.

[75] V. Lenarduzzi, S. Morasca, and D. Taibi, "Estimating software development effort based on phases," *Proc. - 40th Euromicro Conf. Ser. Softw. Eng. Adv. Appl. SEAA 2014*, no. December 2015, pp. 305–308, 2014, doi: 10.1109/SEAA.2014.54.

[76] D. Baca, *Software, Developing Secure - in an Agile Process*. 2012.

APPENDIX: IMPLEMENTATION CODES

CODES FOR BIDIRECTION LSTM

Importing libraries

```
from tensorflow import keras

from keras.preprocessing.text import Tokenizer

from keras.preprocessing.sequence import pad_sequences

from keras.models import Sequential

from keras.layers import Dense, Embedding, LSTM, Bidirectional

from sklearn.model_selection import train_test_split

import numpy as np

import pandas as pd
```

Reading dataset

```
df = pd.read_csv('/content/drive/MyDrive/Anas datasets/eclipse_cleaned_data.csv')
```

Exploring dataset

```
df.sample()

X = df['Title']

y = df['Classes']
```

Dataspliting

```
from sklearn.model_selection import train_test_split

X_train, X_test , y_train, y_test = train_test_split(X, y , test_size = 0.30)
```

Data preprocessing

```
vocab_size = 1000

oov_token = "<OOV>"

max_length = 100

padding_type = "post"

trunction_type="post"

tokenizer = Tokenizer(num_words = vocab_size, oov_token=oov_token)

tokenizer.fit_on_texts(X_train.astype(str))

word_index = tokenizer.word_index
```

Create Sequence

```
X_train_sequences = tokenizer.texts_to_sequences(X_train.astype(str)

X_train_sequences[10:15]
```

Padding sequence

```
X_train_padded = pad_sequences(X_train_sequences,maxlen=max_length,

padding=padding_type, truncating=trunction_type)

X_train_padded
```

```
X_test_sequences = tokenizer.texts_to_sequences(X_test.astype(str))

X_test_padded = pad_sequences(X_test_sequences,maxlen=max_length,
padding=padding_type, truncating=trunction_type)
```

Prepare Glove Embeddings

```
embeddings_index = {}

f = open('/content/drive/MyDrive/Anas datasets/glove.6B.100d.txt')

for line in f:

    values = line.split()

    word = values[0]

    coefs = np.asarray(values[1:], dtype='float32')

    embeddings_index[word] = coefs

f.close()

print('Found %s word vectors.' % len(embeddings_index))

embeddings_index['attention']

embedding_matrix = np.zeros((len(word_index) + 1, max_length))

for word, i in word_index.items():

    embedding_vector = embeddings_index.get(word)

    if embedding_vector is not None:

        # words not found in embedding index will be all-zeros.
```

```
        embedding_matrix[i] = embedding_vector
```

Embedding Layer

```
embedding_layer = Embedding(len(word_index) + 1,
                            max_length,
                            weights=[embedding_matrix],
                            input_length=max_length,
                            trainable=False)
```

Model Definition

```
embedding_dim = 16
input_length = 100
model = Sequential([
    embedding_layer,
  Bidirectional(LSTM(embedding_dim, return_sequences=True)),
  Bidirectional(LSTM(embedding_dim,)),
  Dense(6, activation='relu'),
  Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
```

```
model.summary()
```

Train Model

```
History = model.fit(X_train_padded, y_train, epochs=20, validation_data=(X_test_padded, y_test))
```

```
import pickle
```

```
f = open('/content/drive/MyDrive/Anas datasets/eclipse_history.pckl', 'wb')
```

```
pickle.dump(history.history, f)
```

```
f.close()
```

Visualizing results

```
import matplotlib.pyplot as plt
```

```
plt.plot(history.history['accuracy'])
```

```
plt.plot(history.history['val_accuracy'])
```

```
plt.title('model accuracy')
```

```
plt.ylabel('accuracy')
```

```
plt.xlabel('epoch')
```

```
plt.legend(['train', 'val'], loc='upper left')
```

```
plt.show()
```

```
import matplotlib.pyplot as plt
```

```
plt.plot(history.history['loss'])
```

```
plt.plot(history.history['val_loss'])

plt.title('model loss')

plt.ylabel('accuracy')

plt.xlabel('epoch')

plt.legend(['train', 'val'], loc='upper left')

plt.show()

y_pred = model.predict(X_test_padded)

from sklearn.metrics import accuracy_score,precision_score,recall_score,f1_score

accuracy = accuracy_score(y_test, np.round(abs(y_pred)))

print('Accuracy: %f' % accuracy)

precision = precision_score(y_test, np.round(abs(y_pred)))

print('Precision: %f' % precision)

recall = recall_score(y_test, np.round(abs(y_pred)))

print('Recall: %f' % recall)

# f1: 2 tp / (2 tp + fp + fn)

f1 = f1_score(y_test, np.round(abs(y_pred)))

print('F1 score: %f' % f1)

import numpy as np
```

```
(unique, counts) = np.unique(y_train, return_counts=True)


frequencies = np.asarray((unique, counts)).T

print(frequencies)

(unique, counts) = np.unique(y_test, return_counts=True)

frequencies = np.asarray((unique, counts)).T

print(frequencies)

y_train.shape
```

evaluate the model

```
train_acc = model.evaluate(X_train_padded, y_train)

test_acc = model.evaluate(X_test_padded, y_test)

print("train acc", train_acc)

print("test acc", test_acc)
```

CODES FOR LSTM

Importing Libraries

```
import pandas as pd

from nltk.corpus import stopwords

import re
```

```
from keras.preprocessing.text import Tokenizer

from keras.preprocessing.sequence import pad_sequences

from sklearn.model_selection import train_test_split
```

Reading Data

```
df = pd.read_csv('/content/drive/MyDrive/Anas datasets/eclipse_cleaned_data.csv')

df.head()
```

Data Tokenization and Embeddings

```
# The maximum number of words to be used. (most frequent)

MAX_NB_WORDS = 50000

# Max number of words in each complaint.

MAX_SEQUENCE_LENGTH = 250

EMBEDDING_DIM = 100

tokenizer = Tokenizer(num_words=MAX_NB_WORDS, filters='!"#$%&()*+,-
./:;<=>?@[\]^_`{|}~', lower=True)

tokenizer.fit_on_texts(df['Title'].astype(str))

word_index = tokenizer.word_index

print('Found %s unique tokens.' % len(word_index))

X = tokenizer.texts_to_sequences(df['Title'].astype(str))
```

Padding Sequence

```
X = pad_sequences(X, maxlen=MAX_SEQUENCE_LENGTH)

print('Shape of data tensor:', X.shape)

Y = pd.get_dummies(df['Classes']).values

print('Shape of label tensor:', Y.shape)
```

Data Splitting

```
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size = 0.10, random_state = 42)

print(X_train.shape,Y_train.shape)

print(X_test.shape,Y_test.shape)
```

Defining Model

```
from keras.models import Sequential

from keras.layers import Dense

from keras.layers import LSTM, SpatialDropout1D

from keras.layers.embeddings import Embedding

from keras.callbacks import EarlyStopping

model = Sequential()

model.add(Embedding(MAX_NB_WORDS,EMBEDDING_DIM, input_length=X.shape[1]))

model.add (SpatialDropout1D (0.2))
```

```
model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))

model.add(Dense(2, activation='softmax'))

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

epochs = 5

batch_size = 128

history = model.fit(X_train, Y_train, epochs=epochs,

batch_size=batch_size,validation_split=0.1,callbacks=[EarlyStopping(monitor='val_loss',

patience=3, min_delta=0.0001)])

model.summary()
```

evaluate the model

```
train_acc = model.evaluate(X_train, Y_train)

test_acc = model.evaluate(X_test, Y_test)

print("train acc", train_acc)

print("test acc", test_acc)

# model.save('/content/drive/MyDrive/Anas datasets/mozilla_model.h5')

from matplotlib import pyplot
```

plot loss during training

```
pyplot.subplot(211)

pyplot.title('Loss')
```

```
pyplot.plot(history.history['loss'], label='train')

pyplot.plot(history.history['val_loss'], label='test')

pyplot.legend()
```

plot accuracy during training

```
pyplot.subplot(212)

pyplot.title('Accuracy')

pyplot.plot(history.history['accuracy'], label='train')

pyplot.plot(history.history['val_accuracy'], label='test')

pyplot.legend()

pyplot.show()

classes = model.predict(X_test).argmax(axis=1)

from sklearn.metrics import accuracy_score,precision_score,recall_score,f1_score

import numpy as np

accuracy = accuracy_score(np.argmax(Y_test, axis=1), np.round(abs(classes)))

print('Accuracy: %f' % accuracy)

precision = precision_score(np.argmax(Y_test, axis=1), np.round(abs(classes)))

print('Precision: %f' % precision)

recall = recall_score(np.argmax(Y_test, axis=1), np.round(abs(classes)))
```

```
print('Recall: %f' % recall)

f1 = f1_score(np.argmax(Y_test, axis=1), np.round(abs(classes)))

print('F1 score: %f' % f1)
```

CODES FOR NLP

Import Libaries and load data

```
import nltk

import pandas as pd

import re

from sklearn.feature_extraction.text import TfidfVectorizer

import string
```

Reading Data

```
Data = pd.read_csv('/content/drive/MyDrive/Anas datasets/eclipse_cleaned_data.csv')

data.head()

data['Classes'].value_counts()
```

Extract features

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(min_df=0, lowercase=False)
```

```
vectorizer.fit(data)

vectorizer.vocabulary
```

Data Splitting

```
from sklearn.model_selection import train_test_split

X=data[['Title']]

y=data['Classes']

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.1, random_state=42)

nltk.download('stopwords')
```

Data Preprocessing

```
stopwords = nltk.corpus.stopwords.words('english')

ps = nltk.PorterStemmer()

def clean_text(text):

    text = "".join([word.lower() for word in text if word not in string.punctuation])

    tokens = re.split('\W+', text)

    text = [ps.stem(word) for word in tokens if word not in stopwords]

    return text
```

Vectorize text

```
tfidf_vect = TfidfVectorizer(analyzer=clean_text)
```

```
tfidf_vect_fit = tfidf_vect.fit(X_train['Title'].values.astype('U'))

tfidf_train = tfidf_vect_fit.transform(X_train['Title'].values.astype('U'))

tfidf_test = tfidf_vect_fit.transform(X_test['Title'].values.astype('U'))
```

Final evaluation of models

```
from sklearn import metrics

from sklearn.ensemble import RandomForestClassifier

from sklearn import svm

rf = RandomForestClassifier (n_estimators=150, max_depth=None, n_jobs=-1)

rf_model = rf.fit(tfidf_train, y_train)

y_pred = rf_model.predict(tfidf_test)

print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

print("Precision:",metrics.precision_score(y_test, y_pred))

print("Precision:",metrics.recall_score(y_test, y_pred))

print("F1score:",metrics.f1_score(y_test, y_pred))
```

Create svm Classifier

```
clf = = svm.SVC(kernel='linear') # Linear Kernel
```

Trained the model using the training sets

```
clf.fit.(tfidf_train, y_train)
```

Predict the response for test dataset

svm_y_pred == clf.predict.(tfidf_test)

Print ("Accuracy:",metric.accuracy_score (y_test, svm_y_pred))

Print ("Precision: “metric. precision score (y_test, svm_y_pred))

Print ("Precision: “metric. recall_score(y_test, svm_y_pred))

Print ("F1score:"metric. f1_score(y_test, svm_y_pred))