On Empirically Examining The Effectiveness Of Deep Learning-Based Bug Localization Models

by

Sravya Polisetty

Bachelor of Technology, Jawaharlal Nehru Technological University, 2012

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the program of

Computer Science

Toronto, Ontario, Canada, 2018

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my dissertation may be made electronically available to the public.

On Empirically Examining The Effectiveness Of Deep Learning-Based Bug Localization Models

Master of Science 2018

Sravya Polisetty

Computer Science

Ryerson University

Abstract

Software Bug Localization involves a significant amount of time and effort on the part of the software developer. Many state-of-the-art bug localization models have been proposed in the past, to help developers localize bugs easily. However, none of these models meet the adoption thresholds of the software practitioner. Recently some deep learning-based models have been proposed, that have been shown to perform better than the state-of-the-art models. With this motivation, we experiment on Convolution Neural Networks (CNNs) to examine their effectiveness in localizing bugs. We also train a SimpleLogistic model as a baseline model for our experiments. We train both our models on five open source Java projects and compare their performance across the projects. Our experiments show that the CNN models perform better than the SimpleLogistic models in most of the cases, but do not meet the adoption criteria set by the practitioners.

Acknowledgements

This master's thesis is submitted to fulfill the requirements of the MSc of Computer Science at Ryerson University in Toronto, Canada. The work carried out in this thesis was supervised by Dr. Andriy Miranskyy and Dr. Ayse Bener.

First and foremost, I would like to express my sincere gratitude to Dr. Miranskyy for his enormous support and guidance throughout my study. He has set an example of excellence as a researcher, professor, mentor and a role model. I consider myself very fortunate to have had the opportunity to work under his supervision.

I would also like to sincerely thank Dr. Bener for inspiring me to work hard and aim high. Her discipline and tenacity are very admirable and motivating. I also thank her for giving me the opportunity to work in different industrial projects which gave me immense exposure and knowledge.

I will always be grateful to my supervisors for financially supporting my research study, and also for giving me the opportunity to attend various conferences and meet reputed researchers in the field.

Besides my supervisors, I would like to express my gratitude to all the members of my thesis committee.

I would also like to appreciate the feedback and support I received from my fellow lab mates in Ryerson DSL and AMiR Labs - Shirin Akbarinasaji, Mefta Sadat, Jorge Lopez, William Pourmajidi, Sheik Mamun and Mujahid Sultan.

I would like to acknowledge the Department Of Computer Science at Ryerson University and Compute Canada (www.computecanada.ca) and its regional partners — WestGrid (www.westgrid.ca) and Compute Ontario (www.computeontario.ca) — for providing the computation resources needed for my research.

Last but not least, I would like to thank my husband Pandu, whose love and support made this possible.

Dedication

To my husband, Pandu, who has been a constant source of support during my graduate school life. This work is also dedicated to my parents, Anantha and Bhanu, who have always loved me unconditionally and inspired me to work hard to fulfill my dreams.

Contents

Declaration . . . ii
Abstract . . . iii
Acknowledgements . . . iv
Dedication . . . v
List of Tables . . . ix
List of Figures . . . x

1 Introduction **1**
1.1 Terminology . . . 2
1.1.1 Software Bugs . . . 2
1.1.2 Bug Reports . . . 2
1.1.3 Bug Localization . . . 4
1.2 Motivation . . . 4
1.3 Objective . . . 6
1.4 Proposed Solution . . . 7
1.5 Novelty & Contribution . . . 8
1.6 Outline . . . 8

2 Literature Review **10**
2.1 Traditional Approaches . . . 11
2.2 Deep Learning for NLP . . . 13
2.3 Deep Learning for Software Engineering . . . 15
2.4 Practitioner's Expectations . . . 16

3 Methodology **20**
3.1 The Learning-To-Rank Problem . . . 21

3.2 Deep Learning . . . 22
3.2.1 Deep Learning For Bug Localization . . . 29
3.3 Convolution Neural Nets (CNNs) . . . 30
3.3.1 On Textual Data . . . 35
3.3.2 Effect Of Hyper-Parameters . . . 40
3.4 SimpleLogistic . . . 44
3.5 Evaluation Metrics . . . 46

4 Evaluation **50**
4.1 Data Extraction . . . 50
4.2 Data Analysis . . . 53
4.3 Data Preprocessing . . . 56
4.4 Experimental Setup . . . 57
4.4.1 CNN Model . . . 58
4.4.2 SimpleLogistic Model . . . 62
4.5 Results . . . 62
4.6 Discussion . . . 65
4.6.1 Minimizing The Lexical Gap Between Bug Reports And Source Files 65
4.6.2 CNN vs SimpleLogistic models . . . 67
4.6.3 Performance Across Projects . . . 69
4.6.4 Effect Of Varying Buggy Files . . . 70
4.6.5 Practical Relevance . . . 75
4.7 Threats To Validity . . . 81
4.7.1 Internal Validity . . . 81
4.7.2 Construct Validity . . . 81
4.7.3 Conclusion Validity . . . 81
4.7.4 External Validity . . . 82

5 Conclusion and Future Work **83**
5.1 Conclusion . . . 83
5.2 Future Work . . . 86

Appendices **87**

A Additional Figures **88**

B Data Extraction Scripts **93**

C Source Code Parser Scripts **97**

D Data Preprocessing Scripts **102**

E Scripts For CNN Model **110**

F Scripts For LMT Model **120**

G Evaluation Metrics Scripts **122**

References **136**

List of Tables

3.1 Effect of filter region size with several region sizes on the Movie Reviews dataset 42
3.2 Performance of different activation functions 43

4.1 The list of open source projects used in this study 51
4.2 Linked And Not-Linked Records 53
4.3 Bug Reports And Source Files Statistics 54
4.4 Dictionary Size After Preprocessing 57
4.5 Hyperparamters - CNN Models 59
4.6 Hyperparameters - SimpleLogistic Models 62
4.7 Memory And Training Time - 'All Files' 64
4.8 Memory And Training Time - 'Buggy Files' 64
4.9 Memory And Training Time - 'Very Buggy Files' 64
4.10 Results - 'All Files' 65
4.11 Results - 'Buggy Files' 65
4.12 Results - 'Very Buggy Files' 65
4.13 Comparison Of HyLoc With Other Models [67] 77
4.14 Comparison Of DNN-based Models With Other Models [126] 77

List of Figures

1.1 Life Cycle Of A Bug [2] . . . 3
1.2 Sample Eclipse Bug Report In Bugzilla . . . 4

3.1 Machine Learning vs. DNN Models . . . 23
3.2 A Single Neuron . . . 24
3.3 Simple Neural Network . . . 24
3.4 GPU-Accelerated Computing . . . 29
3.5 Convolutions With 3×3 Filter . . . 32
3.6 Image Classification Using CNN . . . 33
3.7 Sigmoid Function . . . 36
3.8 Tanh Function . . . 36
3.9 ReLU Function . . . 36
3.10 Softplus Function . . . 36
3.11 Architecture Of The CNN Model . . . 38
3.12 CNN For Sentiment Analysis . . . 39

4.1 Sample Eclipse Bug Report . . . 52
4.2 Code From PartRenderingEngine.java . . . 52
4.3 Analysis Of Linked Records - AspectJ . . . 55
4.4 Analysis Of Linked Records - Tomcat . . . 56
4.5 Epoch Vs Accuracy & Loss - AspectJ . . . 61
4.6 Epoch Vs Accuracy & Loss - Tomcat . . . 61
4.7 Plots For Metrics - CNN Model - AspectJ - 'All Files' . . . 71
4.8 Plots For Metrics - CNN Model - AspectJ - 'Buggy Files' . . . 72
4.9 Plots For Metrics - CNN Model - AspectJ - 'Very Buggy Files' . . . 73

A.1 Analysis Of Linked Records - SWT 88
A.2 Analysis Of Linked Records - Eclipse 89
A.3 Analysis Of Linked Records - JDT 89
A.4 Plots For Metrics - CNN Model - Tomcat - 'All Files' 90
A.5 Plots For Metrics - CNN Model - Tomcat - 'Buggy Files' 91
A.6 Plots For Metrics - CNN Model - Tomcat - 'Very Buggy Files' 92

Chapter 1

Introduction

The quality of the software developed is important for the success of any software project. Software quality assurance is the process that ensures that the software being developed, meets all the expected quality standards [107]. A software *bug* is an anomaly in the software product that causes the software to perform incorrectly or to behave in an unexpected way [32]. Software quality assurance aims to detect, analyze and correct bugs in the software and many organizations spend a significant amount of time and resources during this process [33]. In fact, it is one of the most resource consuming tasks in the entire software development life cycle [119]. In spite of this, software systems are often shipped with bugs. Large software projects receive huge number of bugs every day. For instance, the Eclipse [15] project had nearly 13,016 bugs reported in a span of one year (2004 - 2005), with an average of 37 bugs reported per day, and a maximum of 220 bugs reported in a single day [24].

Once a bug is reported and confirmed, the software developer has to find the root cause of the bug in the source code and then fix the root cause. This process is typically manual and can take 30% – 40% of the total time needed to fix a problem [84]. This is a painstaking process, especially for large software projects with hundreds of thousands of source files. As a result, the bug fixing time increases, along with maintenance cost of the project. Although the bug fixing task constitutes sub-tasks like understanding the bug, validating the bug, locating the cause of the bug, and finally fixing the bug, it is *"locating the cause of the bug (Bug Localization)"*, that consumes most of the developer's time [34]. Hence, there is a need for automated tools or approaches which perform bug

localization.

1.1 Terminology

Throughout the study the following **terminology** is used.

1.1.1 Software Bugs

The users of a software product often encounter a problem or a *fault* or an *error* in the software that leads to an undesired outcome or even a software *failure* [107]. This problem or *flaw* is also known as a software bug. Bugs arise from errors made by developers in source code or its design.

1.1.2 Bug Reports

A bug report passes through various phases, before it is fixed and closed. Figure 1.1 illustrates the life cycle of a bug in Bugzilla [2] bug tracking system. On detecting a bug or fault in the software, a user or software tester, reports the bug in a document called a *bug report* or an *issue report*. A bug report is a software artifact which is logged by a user or tester in a bug tracking system, such as Bugzilla or JIRA [11]. Before a bug report is considered valid, it goes through various stages of quality checks. During these stages, a bug is checked for duplicity, validity and, completeness (i.e., whether the information given in the bug report is sufficient to identify the bug). After a bug passes through these stages, it is assigned to an expert ("Assignee"), who is expected to have the knowledge to fix the bug. The Assignee, refers to the information in the bug report to identify the source files which need to be modified, in order to solve the issue in the bug report. Bug reports typically consist of various fields such as Title, Version, Component, Product, Quality Assurance (QA) Contact, Assignee, Reported Date, Importance, Attachments, which contain description of the bug, screen shots or snapshots of an error message, stack traces, etc. Figure 1.2 shows an Eclipse bug report from Bugzilla.

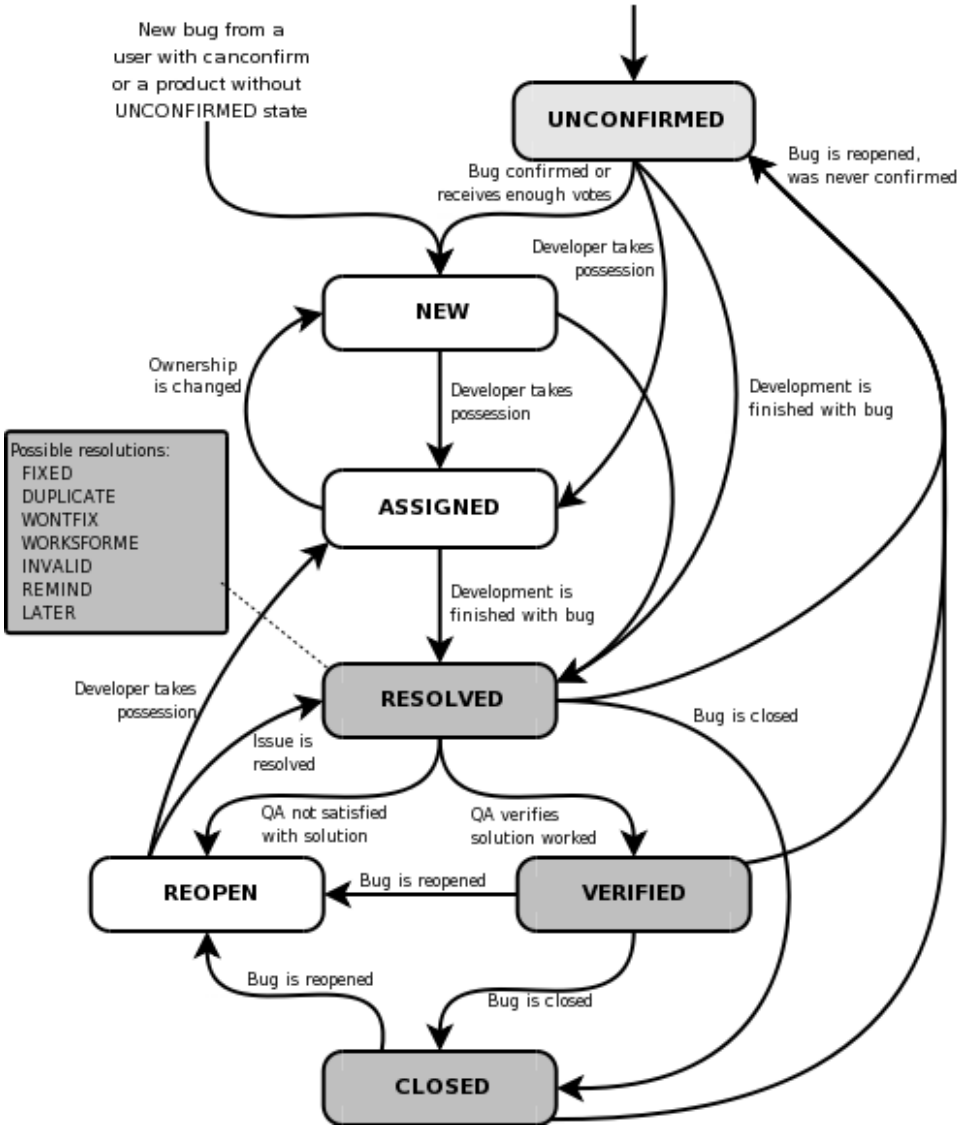


Figure 1.1: Life Cycle Of A Bug [2]

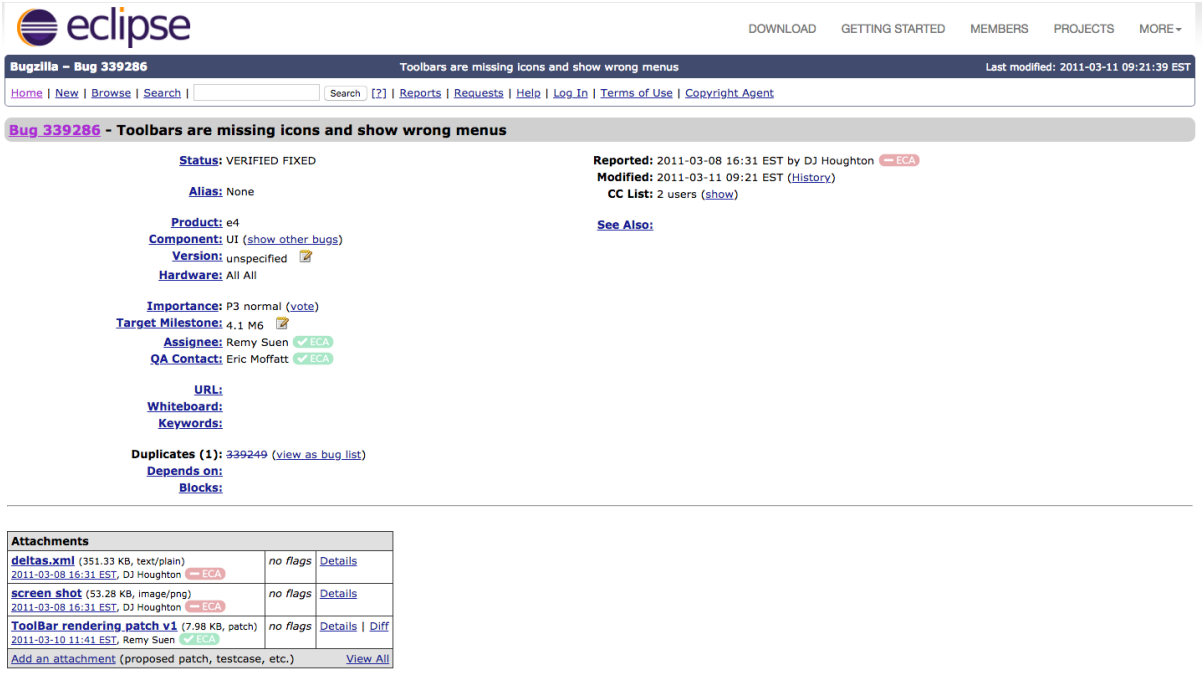


Figure 1.2: Sample Eclipse Bug Report In Bugzilla

1.1.3 Bug Localization

"Bug Localization is the process of identifying the specific location(s) or region(s) of source code (at various granularity levels, such as the directory path, file, method or statement) that is faulty or buggy, and needs to be modified to repair or fix the fault or bug" [66].

1.2 Motivation

As mentioned earlier, a bug is assigned to a developer, who is expected to fix the issue in the bug report. The developer refers to the information in the bug report (to identify the source files which need to be modified) to fix the issue in the bug report. Manually identifying the bug from the source files is a rigorous task, especially for large and complex software systems. It involves a significant amount of time and effort on the part of software developers. To reduce this workload, many researchers have proposed various approaches or models, to pinpoint the locations of bugs in the source files. These

approaches can be classified into *dynamic* and *static* approaches. The dynamic approach [23, 58, 95] uses the semantics of the program and the test case execution information to localize bugs. The static approach [52, 77, 92] relies purely on the source code and bug reports information. It uses pre-defined bug patterns and Information Retrieval (IR) or Machine Learning (ML) to localize bugs.

IR can be defined as "Retrieving relevant document or documents that satisfy user information needs, from large and unstructured collection of documents" [78]. IR models are popular in bug localization domain mainly because of scalability and language independence [92]. This means, even if a software project grows in size and complexity, that the IR model will still remain applicable to the project. On querying an IR model with a bug report, the model retrieves a ranked list of potential buggy source files, which contain the fix for the issue in the bug report. Most of these models use IR-based techniques like *Term Frequency Inverse Document Frequency* (TF-IDF) [72], *Latent Semantic Analysis* (LSA) [43], *Latent Dirichlet Allocation* (LDA) [30], *Vector Space Model* (VSM) [97], and *revised Vector Space Model* (rVSM) [136].

ML has also been used by many researchers to propose various models for localizing bugs. ML is a subset of Artificial Intelligence, in the field of Computer Science, that often uses statistical techniques to give computers the ability to *"learn"*, i.e., progressively improve performance on a specific task, with data, without being explicitly programmed [98]. Traditional ML approaches, such as topic modelling [29] and Naive Bayes [73], have been used to build tools that can localize bugs.

Though all the above state-of-the-art models are able to localize bugs to some extent, none of them could meet the expectations of the software practitioners, who are the end users of these models [64]. These models are unable to bridge the lexical gap between the bug reports and source code. To bridge this gap, recently some researchers [54, 67] have proposed deep learning-based bug localization models.

Deep Learning [41] is the most sought after approach by many researchers nowadays. This is because deep learning-based models have proved to perform better than traditional ML models in the areas of image processing [65], speech recognition [49], and natural language processing (NLP) [35]. In fact, this is the reason why many recent works use deep learning on software data to solve various software engineering problems, such as user profiling [38], defect prediction [129], software artifact traceability [48], code suggestion [121], processing programming languages [83], and bug localization [54, 67].

In the past few years, different architectures of Deep Neural Nets (DNN) have been proposed to localize bugs. Our objective to examine the effectiveness of these models is motivated by the fact that all these deep learning-based models have been shown to outperform other state-of-the-art bug localization models [54, 67, 68, 126].

1.3 Objective

Our primary objective is to examine the effectiveness of the deep learning model in meeting the expectations of the software practitioner. We reach our primary objective by answering the following research questions.

RQ1: How can we minimize the lexical gap between natural language texts in bug reports and technical/domain corpus in source code files in order to automatically localize bugs?

RQ2: How effective are the Convolution Neural Net (CNN) models in meeting the expectations of the software practitioner?

The secondary objectives include (a) to compare the performance of our deep learning model with a traditional ML model, (b) to apply deep learning-based bug localization models to five open source software datasets and compare the performance of the models across the datasets, and (c) to observe the effect of varying source files on the performance of the model. We reach these secondary objectives by answering the following research questions.

RQ3: How do the CNN models perform in comparison with a standalone logistic regression model like SimpleLogistic models, on software bug localization data?

RQ4: How do the CNN models perform across different open source software bug localization datasets?

RQ5: How does varying the source files in the dataset affect the performance of the

CNN and SimpleLogistic Models?

1.4 Proposed Solution

In order to address RQ1, we give a detailed explanation of the existing state-of-the-art traditional and deep learning models and why a deep learning-based model could be a potential solution for bridging the lexical gap between bug reports and source files. For addressing RQ2, we train CNNs on open source datasets, which have been widely used in the past bug localization studies. We then compare the performance of the CNN models, with the expectations of the software practitioner [64]. We experiment with CNNs to solve the Learning-To-Rank bug localization problem using the classification approach, which generates scores for each sample in the dataset. This score gives the degree of likeliness that the sample belongs to the positive or negative class.

In order to make a comparison with the CNN model, we initially chose one of the classic models (namely, Naive Bayes) as our base-line model. However, classifiers like Naive Bayes do not assign well-calibrated scores to the suggested source files. As bug localization is a Learning-To-Rank IR problem, we need calibrated results (those which do not have ties between the scores assigned to the suggested source files for a bug report) in order to be able to correctly measure the performance of the resultant IR model. Hence, we need a logistic regression model like the SimpleLogistic model that can give us calibrated output scores and can also meet the computation and memory constraints, which are inherent when dealing with large datasets like the ones used in this study. Thus, RQ3 is addressed by training a SimpleLogistic model on the bug localization datasets used in this study. The same experimental set up is used to train both the CNN and SimpleLogistic models.

Next, for answering RQ4, we train the CNN model on five open source bug localization datasets and evaluate the performance of the model on each of them. The experimental set up is kept similar while training the models on each dataset, so as to make a fair comparison between the models. We address the next research question, RQ5 by varying the source files in each dataset. In the first case, we use all the source files in the project repository. In the second case, we use only the source files which have been mapped to at least one bug report in the past. In the last case, we consider only those source files

which are *Very Buggy*, i.e., which have more than one bug report mapped to it in the past. We evaluate each of these models to address RQ5.

1.5 Novelty & Contribution

To the best of our knowledge, no other work empirically examines the effectiveness of deep learning-based bug localization models. Through our experiments, we study the relevance of deep learning and traditional models in meeting the expectations of the software practitioner. This study highlights the drawbacks of the current state-of-the-art models and also the recent deep learning models, which is helpful to the future bug localization research. Researchers who aim to build models that localize bugs can take our study as a reference point. Also, the end-users, i.e., the software practitioners should refer to our study before using any bug localization model, as it helps them identify the setbacks of the current models. The major contributions of this work can be summarized as follows.

- Through the experiments carried out in this study, we examine the relevance of a deep learning-based bug localization model to the software industry.
- We evaluate a deep learning model (CNN) against a traditional ML model (SimpleLogistic) on bug localization data.
- We extract the source files for all the bug reports based on the bug-commit mapping for five open source datasets and train bug localization models on all of them.
- We empirically show the effect of varying the source files on the performance of both the CNN and SimpleLogistic bug localization models.

1.6 Outline

In Chapter 2, we provide related works and a literature review on bug localization. In Chapter 3, we introduce the methodologies that we use to build the bug localization models and the approaches followed to evaluate the models. In Chapter 4, we analyze the datasets used in the study and the experimental setup along with a detailed discussion

of the results obtained. Finally, in Chapter 5, we provide a summary of the study, and directions for future research.

Chapter 2

Literature Review

As discussed in Chapter 1, the need for automated tools for bug localization, which help software developers narrow-down potential buggy files from the entire source base, has motivated many researchers to investigate this area and propose new approaches. The motivation behind this study is based on the fact, that most of the state-of-the-art bug localization models do not meet the expectations of the practitioners, who are the end users of these models [64]. Recently some deep learning-based models [54, 67, 126], have been proposed that link high-level, abstract concepts between bug reports and source files. They do this by learning to relate the terms in bug reports and different code tokens and terms in source files. They have been shown to be able to bridge the lexical gap between the bug reports and source code. The performance of these models has been shown to be better than the other traditional ML models in the literature. With this motivation, we build a bug localization model using a CNN that is widely used in the past for text classification. We compare its performance with a traditional ML model like SimpleLogistic and also empirically examine its relevance to the software industry.

In this chapter, we discuss some significant research in bug localization, that uses the traditional IR and ML approach. Also, we briefly mention research related to the application of deep learning models to the bug localization problem. A brief description of significant work done in the area of deep learning for NLP and for software engineering problems is also included in this chapter. We conclude the chapter with an overview of a survey, which highlights the expectations of the software practitioner on the bug localization models and the future directions to be taken by researchers, to build effective

models that are relevant to the industry.

2.1 Traditional Approaches

The automated approaches in bug localization can be broadly classified into two categories: *dynamic* approach and *static* approach. The semantics of the program and its execution information with test cases, i.e., pass/fail execution traces are used in the dynamic approach. In this approach, we can localize a bug with higher granularity, i.e., pinpoint the buggy statement or a block in the source file. But this approach needs information pertaining to the execution of the test cases i.e. the pass/fail execution traces. This makes these approaches dependent on the quality of the test suite used. In reality, the adoption of functional testing in many software projects is poor [63]. Hence this approach for bug localization may not be effective for all software projects.

The dynamic approaches proposed in the literature are: *Spectrum-based fault localization* and *Model-based fault localization*. The Spectrum-based approach uses program traces to correlate program elements at statement-, block-, function-, or component-level in a program to the program failures. These approaches have been used to localize faults in [23, 58]. Their techniques are based on the idea that a failed program element is more suspicious, if it frequently appears in failed execution traces rather than passed ones. Saha et al. [95] have proposed a model that uses this approach for localizing faults in data-centric programs, which interact with databases. Model-based fault localization techniques have been proposed by Feldman et al. [44] and Mayer et al. [79]. These models are based on expensive logical reasoning of formal models of programs but are often more accurate than the Spectral-based models.

The second type of automated approach, i.e., the static approach, does not need the dynamic program information, like the execution traces. They rely only on the source code and the bug reports information. These static approaches can be further classified into two groups: *program analysis-based* and *IR-based*. The program analysis-based approach uses predefined bug patterns to localize bugs. Hovemeyer et al. [52] used this approach to propose a model called *FindBugs*. This model does not perform well as it gives a large number of false positives and also false negatives [114].

The second type of static approach uses IR and ML techniques, to automate the

search for relevant or potential buggy source files for a given bug report. They use IR-based techniques like TF-IDF, LSA, LDA, VSM, and rVSM. All these techniques work on the same principle, i.e., the content of the bug report is the query, and the source files in the project are the collection of documents returned by the query. Here, the bug localization problem is treated as a *Learning-To-Rank* IR problem.

Rao et al. [92] probed into many of these IR techniques and concluded that simpler techniques, such as TF-IDF, work better than complex ones. Lukins et al. [77] proposed a bug localization model using LDA, which is a topic-modelling approach. Zhou et al. [136] proposed the rVSM approach, which is a refined vector space model to leverage the similarity between the bug reports and source files. Saha et al. [96] proposed a structured retrieval model that employs the structure of bug reports and source code files, to achieve better performance.

Kim et al. [60] have also experimented using Naive Bayes algorithm, with the previously fixed files as classification labels. The trained model is used to assign source files to each bug report. Ye et al. [130] used the adaptive Learning-To-Rank approach, to train the features extracted from source files, API (Application Programming Interface) descriptions, bug-fixing, and change history. Both papers [60, 130] used additional features like the metadata pertaining to bug reports (version, platform, priority, etc.) and source files (bug-fixing histories). Ye et al. also used the text in the documentation of the open source projects used in their study, to train the *word embeddings* for bug reports and source files. Later, Cosine Similarity Analysis (CSA) [103] was performed on the encoded bug reports and source files. On the other hand, Kim et al. did not use the source file's content for feature extraction. They only used the names of the fixed files and labels for their classification model. Le et al. [70] have proposed a multi-modal technique for bug localization that uses the dynamic information like program spectra (i.e., a record of which program elements are executed for each test case) along with the bug reports. The goal here is to create an adaptive model which is specific to each bug report and map it to its possible location.

Most of these state-of-the-art models consider the source code and bug reports to be of the same lexical space and try to correlate these artifacts by measuring their similarity in the same space. It is important to note that none of these traditional approaches for bug localization meet the expectations of the practitioners [64]. We discuss more about this in Section 2.4. Recently, some deep learning-based bug localization models have

been proposed, to solve the bug localization problem. We discuss these models in the next section.

2.2 Deep Learning for NLP

Before we discuss the application of deep learning to the bug localization problem, we give a brief overview of research, which involves the application of deep learning for NLP.

In NLP, learning word encodings (or word embeddings) is a crucial part of the entire training process. Deep Learning is used to study these word vectors. Collobert et al. [36] proposed a multi-layer neural network that works on many NLP tasks. The main motivation behind this model was to avoid task-specific engineering. Another important work, which led to a breakthrough in NLP (text classification) is [132]. Kim [132] has experimented on a basic Convolution Neural Network (CNN) on open source datasets, like Movie Reviews and proved that a simple CNN (with little hyper-parameter tuning and using static word vectors), is capable of performing exceptionally well on the sentence classification task. Kim experimented on four different variations of encoding of the corpus (random, static, non-static, multi-channel). Kim's experiments also proved that, learning task-specific word vectors instead of static vectors improves the performance of the model. A further improvement in the model was seen through the use of a combination of static and non-static word vectors.

Santos et al. [39] trained character-level embeddings and combined them with the pre-trained word embeddings. These were fed into the convolution layers of a CNN model for identifying the Parts Of Speech in the text. Zhang et al. [128, 134] experimented on character-level CNN models for classifying text. Their experiments proved that CNN perform better than the other traditional models (such as *bag-of words*, *n-grams*, and TF-IDF models) and also other deep learning models (such as word-based CNN and Recurrent Neural Networks (RNN)). They also proved that CNN do not require knowledge of words in terms of semantics or context of words, to perform text classification.

Kim et al. [61] explored the application of character-level CNN for language modelling. The output of the character-level CNN was used as the input to an LSTM at each time step. The model proved to be effective for modelling different languages. Johnson and Zhang [56] trained a CNN from scratch without using pre-trained word vectors

(*Word2Vec* [81] or *GloVe* [88]) and applied convolutions directly on *One-Hot* [101] vectors. They also proposed a different version of the traditional bag-of-words approach. This approach is space-efficient and also reduces the number of parameters that the network needs to learn. The authors extended this model in [57] by proposing a new version of word encodings called *region embeddings*. These embeddings are derived by using a CNN which predicts the context of text regions in a sentence. The region embeddings proved to be very effective on long-form texts (like Movie Reviews data) but not on short texts (like tweets). The pre-trained word embeddings perform well on short texts rather than long texts.

Zhang et al. [135] performed a sensitivity analysis on the effect of varying hyper-parameters in a CNN model. This model proves to be very useful to researchers aiming to train CNN models on different datasets. Nguyen et al. [85] built a CNN model for relation extraction and classification tasks. Apart from using the pre-trained word vectors, they also used the relative positions of the words to the entities, as inputs to the convolution layer. This model requires that the positions of the entities is pre-defined and that each input contains a relation. Sun et al. [111] and Zeng et al. [133] have also explored similar models.

Another important application of CNN in NLP was proposed by Gao et al. [47] and Shen et al. [102]. They experimented on how semantically meaningful representation of sentences can be used for IR. One of the applications stated in their work is recommending useful or potentially interesting documents to users, based on their past or current reading list. They train the sentence representations based on the search engine log data of the users. Another important work, on investigating the meaningfulness of the word embeddings learned by a CNN model, was presented by Weston et al. [120]. They use a CNN to predict the hashtags for Facebook posts, and also used the learned embeddings for another task like recommendation systems.

Another type of DNNs, which are widely used and well recognized as a perfect fit for NLP needs, is RNN. They have produced significant breakthroughs in many NLP tasks, e.g., language modelling [82], machine translation [25], semantic entailment [113]. Long Short-Term Memory (LSTM) is a variant of RNN which are proposed by Hochreiter et al. [50, 51] and proved to be better than other DNN models for NLP. Tai et al. [113] proposed a tree-structured LSTM for the task of semantic relatedness between natural language sentences.

In the next section, we discuss some important works, which applied deep learning models to software engineering data.

2.3 Deep Learning for Software Engineering

Recently, deep learning models are being widely used to solve software engineering problems. Researchers in the software engineering community have demonstrated the usefulness of applying deep learning techniques to software engineering corpora to solve problems in the areas of test case prioritization, defect prediction, bug localization, user profiling, etc.

In order to profile the usage of a software product to improve the quality of the software, Curro et al. [38] proposed an automatic approach to extract information about user actions from publicly available video tutorials of a product. They used a Deep Convolutional Neural Network (DCNN) to recognize user actions and classify them. Their work demonstrates the effectiveness of DCNN-based methods for extracting software usage information from videos. White et al. [121] applied deep learning to model software engineering data like source code files and their model is applicable to other types of software artifacts exhibiting sequential structure, such as execution traces, design documents, and requirement documents. They also applied the deep learning models to code suggestion and demonstrated higher effectiveness than other state-of-the-art models. Mou et al. [83] have proposed a novel tree-based CNN architecture (TBCNN) for processing programming languages. The structure of the program's abstract syntax tree is captured by the convolution kernel. TBCNN proved to be effective in program analysis tasks (e.g., program classification based on functionality) and in detecting unhealthy code (which implements an inefficient algorithm). It also outperformed their baseline Support Vector Machine (SVM) [37] model, including several commonly used neural network models like DNN and RNN.

Guo et al. [48] applied RNN and their variants to establish links between software engineering artifacts, namely Requirements and Design documents. They encoded the corpora in the artifacts, using Word2Vec encoding and found the semantic vector for each artifact using RNN. They calculated the relatedness score between these semantic vectors using the architecture adapted from [113]. Another significant work, which applied deep

learning to software engineering data was proposed by Lam et al. [67]. They applied Restricted Boltzmann Machine (RBM)-based [105] DNN in combination of rVSM on bug localization datasets. The features from the bug reports, source files, and API descriptions were extracted using an autoencoder. The extracted features were later fed in a DNN, which outputs a relatedness score between a bug report and source file. The higher the relatedness score, the greater the probability that the source file contains the root cause of that particular bug report.

Another recent work, which applied deep learning to bug localization data, is by Huo et al. [54]. They adapted a *Pairwise* Learning-To-Rank approach to classify the combined corpora of bug reports and source files into linked and non-linked records. They proposed a new architecture called Natural Language And Programming Language CNN (NP-CNN). Their experiments show that NP-CNN performs better than the other state-of-the-art bug localization models. Huo et al. [126] also proposed another new architecture for the bug localization problem using a combination of LSTM and CNN called LS-CNN. They compared the performance of LS-CNN with 1) NP-CNN, 2) a simple CNN [132], 3) a simple LSTM, and other state-of-the-art bug localization models, such as Buglocator [136], Two-Phase [60], and HyLoc [67]. Their experiments proved that the LS-CNN model outperforms the rest of the deep learning models, and also the traditional models, on the same datasets.

2.4 Practitioner's Expectations

There are numerous studies in the area of bug localization, some of which have been discussed in the previous sections. Unfortunately, very few studies in the literature have investigated the expectations of practitioners. In this section, we discuss one empirical study that surveyed practitioners from diverse backgrounds, about their expectations of research in bug localization [64]. This particular work has been one of our motivating factors, to investigate and examine the relevance of a deep learning-based bug localization model like CNN in meeting the expectations of the practitioners.

The main goal behind the survey conducted by Kochar et al. [64], was to compare the needs of the industry to the current state-of-the-art research in this area. They did this, by surveying 386 practitioners from more than 30 countries across 5 continents,

about their expectations of research in bug localization. They then compared what practitioners need, and the current state of research, by performing a literature review of papers on bug localization techniques published in *International Conference on Software Engineering* (ICSE), *Foundations of Software Engineering* (FSE), *Transactional Software Engineering* (TSE), etc. in the years between 2011 and 2015.

The survey revealed that most of the practitioners are enthusiastic about the research in this area, but have high thresholds of adoption. They expect any bug localization tool to meet certain criteria, e.g.: the availability of debugging data, granularity level of the file when suggesting buggy files for a bug report, reliability, scalability, efficiency, ability to provide a rationale behind every suggestion of buggy file, and integration into an Integrated Development Environment (IDE). We evaluate our CNN and also the SimpleLogistic model against each of these criteria in Chapter 4 of our study. We discuss each of these criteria in details below:

- *Importance of bug localization*: The survey showed that majority of practitioners rated the bug localization research as "Essential" or "Worthwile", with a minority (less than 10%) rated the research as "Unimportant" or "Unwise".

- *Availability of debugging data*: Most of the studies in bug localization assume that debugging data like specification, test cases, bug reports are always available. The survey showed that almost 80% of practitioners agree that bug reports are available "most of the times".

- *Preferred level of granularity*: Different studies propose bug localization models that localize bugs in source files at different levels of granularity, i.e., file, method, block, statement. 51.81% of practitioners in the survey prefer method level granularity. Only 26.4% of practitioners prefer file level granularity.

- *Minimum success criteria*: The success criteria is defined in terms of *Top-k* rank which is an IR metric used to evaluate the performance of bug localization models. As we know, these models suggest a ranked list of source files as the potential buggy files for a bug report. If the relevant or the buggy file appears at the bottom of the list, then a developer is better off doing manual debugging rather than using this model. The survey found that about 74% of developers gave **Top-5** as the minimum success criteria.

- *Trustworthiness*: Based on the above criteria for success, the success rate of a bug localization technique can be determined. A technique that is successful most of the time (i.e., meets the minimum success criteria) is considered trustworthy. The survey found that in order to achieve a satisfaction rate of 50%, 75%, and 90%, a bug localization model has to be successful 50%, 75%, and 90% of the time respectively.

- *Scalability*: The survey found that in order to achieve a developer satisfaction rate of 50%, 75%, and 90%, the bug localization model needs to be scalable enough to deal with programs of size 10,000, 100,000, and 1,000,000 Lines Of Code (LOC), respectively.

- *Efficiency*: In terms of efficiency, the survey found that in order to achieve a developer satisfaction rate of at least 50%, the model should have a run time of less than a minute. This threshold for efficiency was approved by almost 90% of the practitioners who participated in the survey.

- *Willingness to adopt*: Almost all the practitioners in the survey are willing to adopt a bug localization technique, if it satisfies the above criteria of trustworthiness, scalability, and efficiency.

The most important implications that can be drawn from the survey conducted by Kochhar et al. [63] are given below:

- *Demand for bug localization solutions*: It was found that almost all the practitioners who participated in the survey recognized research in this field to be essential or worth while. This is a clear indication that researchers need to continue innovating and propose new solutions for the bug localization problem.

- *Presence of a high adoption barrier*: In spite of the above stated interest of the practitioners in this area of research, most of them are not willing to adopt any bug localization technique unless it satisfies all of the above stated criteria.

- *A need for large improvement in reliability*: It was found that even the best performing studies could not satisfy even 75% of the respondents in the survey. Many of the studies that can satisfy 50% of the practitioners work at a granularity that

is considered very coarse by most of the practitioners, i.e., class- or file-level. One particular study by Qi et al. [90] proposed a bug localization model that works on the preferred level of granularity (method) and satisfies more than 50% of the practitioners, but it has been shown to be effective on only small or medium sized projects (less than 100k LOC). Hence, the existing bug localization techniques need to improve to a large extent to satisfy the software practitioners.

Chapter 3

Methodology

As discussed in the previous chapters, most of the state-of-the art traditional machine learning models are unable to bridge the lexical gap between the bug reports and source code. This can be derived from the fact that none of these models meet the expectations of the practitioners in terms of performance, reliability, granularity, scalability, etc. [64].

Recently some deep learning-based models [54, 67, 126] have been proposed to link bug reports and source code. They do this by learning to relate the terms in bug reports and different code tokens and terms in source files. They have been shown to be able to bridge the lexical gap between the bug reports and source code. The performance of these models have been shown to be better than the other traditional machine learning models in the literature. With this motivation, we build a bug localization model using a CNN that is widely used in the past for text classification. We compare its performance with a traditional machine learning model like SimpleLogistic and also empirically examine its relevance to the software industry. This chapter provides a foundation for the experimentation done in our study. We briefly discuss the merits of deep learning and its application in the area of bug localization. We also explain how CNNs are used for text classification and how the same architecture can also be applied as a Pairwise Approach to the Learning-To-Rank bug localization problem. We also give a brief overview of our baseline model, built using SimpleLogistic. In Section 3.1 we discuss the Learning-To-Rank IR problem and the different approaches used to tackle it. Next, we present some basics of deep learning, followed by how and why deep learning is used in the area of bug localization in Section 3.2. In Section 3.3 we explain the basics of CNNs, their application

on textual data and then the effect of hyper-parameters on their performance. Next, we go into some of the details of SimpleLogistic models [69, 109], in Section 3.4. We end this chapter with a brief description of the various evaluation metrics used in our study. Throughout this chapter, we use the term relevant file and buggy file interchangeably, as by relevant file we mean the file that has the potential fix for a bug report. Similarly, the terms non-relevant and non-buggy are used interchangeably, as by non-relevant we mean the source files which are not buggy or do not have a fix for that particular bug report.

3.1 The Learning-To-Rank Problem

As specified in the previous chapter, automated bug localization approaches, which analyze the content of bug reports, are based on IR. They use IR to identify the source files which have the potential fix for a given bug report. The query of this IR model is a bug report and the retrieved documents is a ranked list of source files, with the most relevant source file at the top and the least relevant at the bottom of the list. Hence, a bug localization problem is a Learning-To-Rank IR problem. Tie-Yan Liu [76] has analyzed the existing algorithms for Learning-To-Rank problems and categorized them into three groups: Pointwise Approach, Pairwise Approach, and Listwise Approach. Each of these approaches are briefly explained below:

Pointwise Approach

In this case, the Learning-To-Rank problem is approximated by a *regression* problem, i.e., given a bug report-source code pair, the model predicts a relatedness score. The higher the score, the greater is the probability that the source file is the relevant buggy file for a given bug report. The pointwise approach has been used by many researchers in the area of bug localization. This approach has also been applied to find the semantic relatedness between natural language sentences [113] and also to establish traceability of software artifacts like Requirement and Design Documents [48].

Pairwise Approach

In this case, the Learning-To-Rank problem is approximated by a *classification* problem, i.e., learning a binary classifier that can identify the buggy source files in a given set of

source files for a given bug report. This approach has been used in bug localization more recently [54, 126]. Each bug report corpus and source file corpus is merged into a sentence and fed into a classification model. Hence the Learning-To-Rank problem is converted into a binary classification problem, with *linked* and *non-linked* as the class labels. The linked records will be the bug reports and the corresponding source files changed in order to fix the issue in the bug report. The non-linked records will be all the combinations of bug reports and source files barring the linked records. The experimental set up and the bug report split used in training and testing the model varies from study to study [54, 67].

Listwise Approach

In this case, the order of a list of source files will be considered for prediction. The objective here, is to produce a ranking model that minimizes the dissimilarity to rankings in the training data. This approach learns a ranking function by taking individual lists as instances and minimizes a loss function defined on the predicted list and the ground-truth list [127]. To the best of our knowledge, this approach has never been used in bug localization research.

It is important to note that, from a practical perspective, for a given bug report, we are only interested in the capability of distinguishing it's buggy source files from all the other source files in the repository. Besides ranking buggy source files over the other source files (non-buggy files), there is no further interest in the relationship among buggy or non-buggy files. Therefore, the Pairwise approach is the most suitable one for bug localization problem [75].

3.2 Deep Learning

The modern deep learning models and training methods originated from research in Artificial Neural Networks (ANNs). ANNs are inspired from biological neural networks that constitute the brain [115]. They were designed to approximate complex functions of the brain by interconnecting large number of computational units in a multi-layered structure. A DNN is an ANN with multiple hidden layers between the input and output

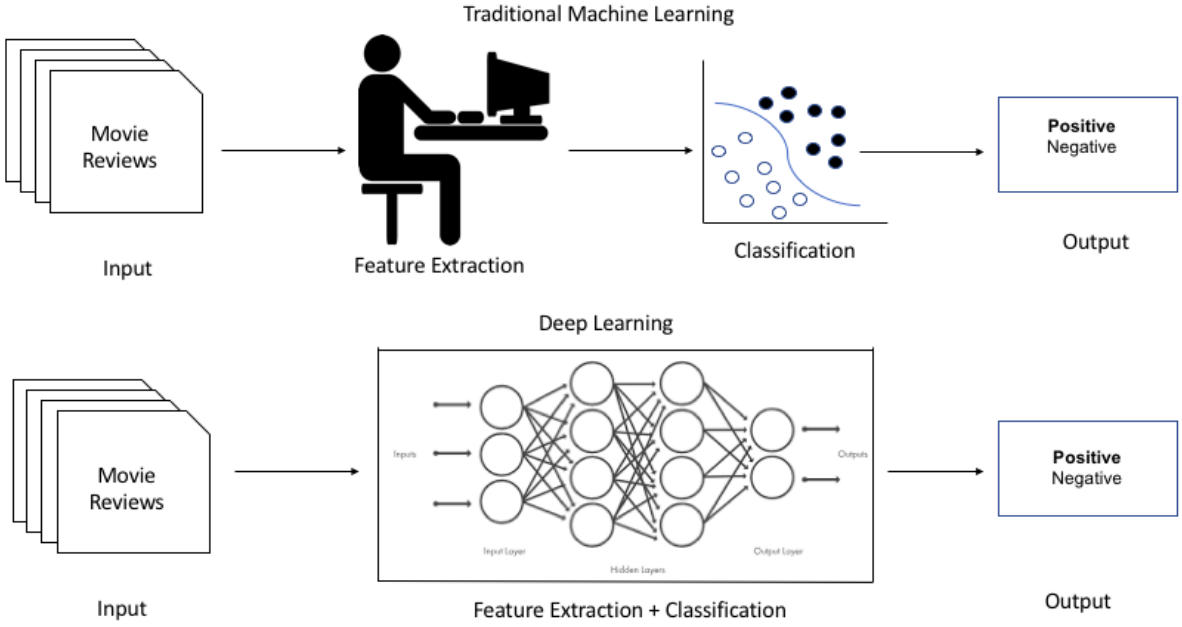


Figure 3.1: Machine Learning vs. DNN Models

layers [27, 99]. The benefit of this complex structure of a DNN, is the ability to represent the data in multiple layers. This is one of the key advantages of DNNs over traditional machine learning models, in which humans have to explicitly extract features from the data, before training the model (Figure 3.1).

ANNs automatically discover "good internal representations", i.e., features that make the learning easier and more accurate, through *backpropagation* [94]. Backpropagation is an effective and a widely recognized method for training a DNN. It calculates the gradient of the error function with respect to the neural network's weights. Consider a simple neuron as shown in Figure 3.2. A neuron maps the inputs $x = \{x_1, \ldots, x_K\}$ to a scalar output y through a weight vector $w = \{w_1, \ldots, w_K\}$ and a non-linear function f. The output y is given by:

$$y = f(\sum_{i=0}^{K} w_i x_i) = f(w^T x). \tag{3.1}$$

An additional element which is equal to 1 is added to the input vector along with an additional weight called the *bias*. The function f which provides the non-linearity

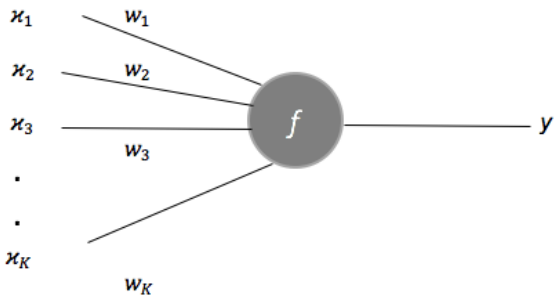


Figure 3.2: A Single Neuron

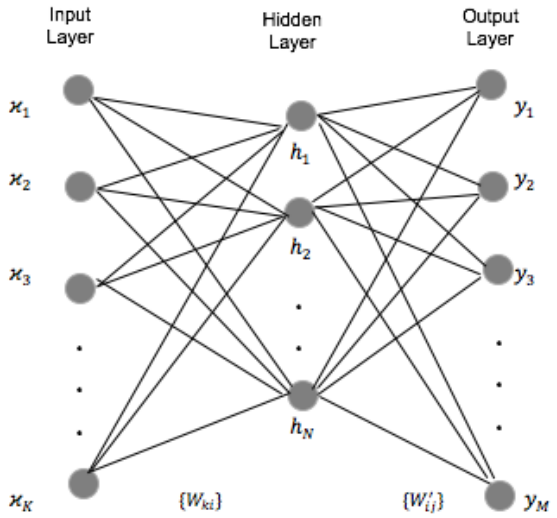


Figure 3.3: Simple Neural Network

between the input and output is called the *activation* function. Some common activation functions are discussed in detail in the next section. Let us consider a *logistic function* f as the activation function, which is given by:

$$f(x) = 1/(1 + e^x). \tag{3.2}$$

So y becomes,

$$y = 1/(1 + e^{w^T x}). \tag{3.3}$$

If we plot y, we get a smooth and differentiable curve bound between 0 and 1 (see Figure 3.7). The derivative of this function will be used when we learn the weight vector w via *Stochastic Gradient Descent* (SGD). When we train a neural network, the ulitimate goal is to learn the weights by minmizing an objective function. Traditionally, objective functions measure the difference between the actual output t and the predicted output $f(w^T x)$. If we use a squared loss function, then the objective function is given by:

$$E = \frac{1}{2}(t - y)^2 = \frac{1}{2}(t - f(w^T x))^2. \tag{3.4}$$

We use SGD to find weights w to minimize the above objective function. To do this, we iteratively update the weight parameters in the direction of the gradient of the loss

function until we reach a minimum. The gradient of E is given by:

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial w_i} = (y - t) \cdot y(1 - y) \cdot x_i. \tag{3.5}$$

We now calculate the gradient of E using backpropagation for a simple neural network shown in Figure 3.3. It consists of an input layer, output layer, and a hidden layer. The input layer consists of the input vector $x = \{x_1, \ldots, x_K\}$. The hidden layer consists of a vector of N neurons $h = \{h_1, \ldots, h_N\}$. The output layer consists of one neuron for every element of the output vector $y = \{y_1, \ldots, y_M\}$.

Each element in the input layer is connected to every neuron in the hidden layer, with w_{ki} as the weight associated with the connection between the k-th input element and the i-th hidden neuron. A similar connection exists between the hidden and output layers with w'_{ij} denoting the weight associated with the connection between the i-th hidden neuron and the j-th output neuron. Intuitively, we can think of w_{ki} as the the (k, i)-th entry in a $K \times N$ weight matrix W and similarly weight w'_{ij} as the (i, j)-th entry in a $N \times M$ weight matrix W'. The output of each neuron is simply the logistic function applied to the weighted sum of the neuron's inputs. For instance, the output of an arbitrary neuron in the hidden layer h_i is given by:

$$h^i = f(u_i) = f\left(\sum_{k=1}^{K} w_{ki} x_k\right), \tag{3.6}$$

and similarly output of an arbitrary output neuron y_j is given by:

$$y^j = f(u\prime_j) = f\left(\sum_{i=1}^{N} w\prime_{ij} h_i\right). \tag{3.7}$$

The objective function in this case will be same as equation 3.5, but will be summed over all the elements in the output layer. It is given by:

$$E = \frac{1}{2} \sum_{j=1}^{M} (y_j - t_j)^2. \tag{3.8}$$

Unlike the previous case, we now need to construct update equations for both sets of weights, i.e., the input-to-hidden layer weights w_{ki} and the hidden-to-output weights w'_{ij}.

For this, we need to compute the gradient of our objective function E with respect to w_{ki} as well as the gradient with respect to w'_{ij}. In order to compute $\frac{\partial E}{\partial w'_{ij}}$, we use the chain rule in calculus. From the chain rule, we first take the derivative of E with respect to $y\prime_j$. Then we take the derivative of y_j with respect to w'_{ij} which needs yet another application of the chain rule. The final equation obtained, after following the above steps is given by:

$$\frac{\partial E}{\partial w_{ki}} = \sum_{j=1}^{M} \left[(y_j - t_j) \cdot y_j \left(1 - y_j\right) \cdot w\prime_{ij} \right] \cdot h_i \left(1 - h_i\right) \cdot x_k. \tag{3.9}$$

The above process is called backpropagation because we begin with the final output error $y_j - t_j$ for the output neuron j and this error gets propagated backwards throughout the network in order to update the weights.

In the previous paragraph, we mentioned the gradient descent optimization. Gradient descent minimizes an objective function parameterized by the model's parameters, by updating the parameters in the opposite direction of the gradient of the objective function with respect to the parameters. There are many other optimization algorithms apart from SGD (like *Momentum*, *Adam*, *Adadelta*, *RMSProp*, *Adamax*, *Nadam*, and *AMSGrad*) that are used for optimizing DNNs, see [93] for details. In our study, we have used the Adam optimization algorithm [62] for the CNN models. Adam stands for *Adaptive Moment Estimation*. It is straightforward to implement and computationally efficient, with little memory requirements. Adam is different from the classical SGD. SGD maintains a single learning rate for all weight updates and the learning rate does not change during training, whereas Adam computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. Adam optimization has proved to be effective and well suited for learning problems that are large in terms of data. This is supported by the fact that most of the recent deep learning researchers [54, 126, 132] use this optimizer when training on large datasets.

As mentioned in Chapter 2, deep learning has been widely used for natural language tasks like parsing, sentiment analysis, question answering and machine translation. They have also been applied to solve software engineering problems like code suggestion, detect code snippets of certain patterns and most importantly bug localization. The need for deep learning came from the limitation of traditional *Feedforward Networks* or *Multi-Layer Perceptrons* (MLPs) whose network parameters grew extremely large as the width

and the depth of the network increased. DNNs on the other hand, can model complex non-linear relationships. Their architecture generates compositional models where an object is expressed as a layered composition of primitives [112]. In case of text, the object is a sentence encoded using Word2Vec [81], GloVe word embeddings, or One-Hot Encoding [101]. A detailed description of each of these encodings is given in Section 3.3.2. The extra layers in a DNN enable composition of features from lower layers, potentially modeling complex data with fewer units than a similarly performing shallow network [27].

Different DNN architectures have been proposed to target various types of practical problems. CNNs are one type of DNNs, which are widely used in image recognition and video analysis tasks. RNNs on the other hand, are widely used for NLP-related tasks as they proved to be a good fit for sequential data like NLP, audio data or temporal data.

LSTMs [51], are a variant of RNNs and have been proposed to address the *vanishing gradient* problem which is prominent in the standard RNN models. Vanishing or Exploding gradients is a phenomenon that occurs when the network degrades when long dependencies exist between the sequences. LSTMs include a *memory cell* that can maintain information in memory for long periods of time. This architecture of LSTM enables it to tackle the vanishing gradient problem.

Many issues arise when DNNs are naively trained, two of the common issues being: *overfitting* and *computation time*. Overfitting occurs when a machine learning algorithm learns even the noise in the data. This occurs when the model or the algorithm fits the data too well. In such cases the model shows low bias but high variance. Overfitting occurs in DNNs when they try to model rare dependencies in the training data through the multiple added layers of abstraction.

There are many ways to avoid overfitting of training data, the most significant being: *regularization*, *dropout* and *k-fold cross validation*. Regularization methods, such as Ivakhnenko's unit pruning [55], weight decay (L_1 regularization), sparsity (L_2 regularization), can be applied during training [26].

Alternatively, another simple but effective method to avoid overfitting is dropout regularization, where units from the hidden layers are omitted randomly during training [40, 108]. Dropout does this by setting the activation of these hidden units to zero. This prevents the co-adaption of feature detectors, as the remaining units are not influenced anymore by the dropped-out units. k-fold cross validation is yet another method which

is widely used not just in case of DNNs, but with any machine learning algorithm to avoid overfitting. In k-fold cross validation, we divide the data randomly in k distinct folds. We then train the model on $k-1$ folds and then test the model on the remaining fold. This process is repeated k times, each time selecting a different fold as the test fold. During each iteration we calculate the performance of the model on the test fold. The final performance of the model is the average performance over the k testing folds. The pseudo code for k-fold cross validation is given below [86]:

1. Let the initial input data X be of $n \times D$ dimensions and output Y be of $Y \times 1$ dimensions.
2. Let the final estimated performance of the model be P_{k-fold}
3. Divide the input data set $[1, 2, 3, \ldots, n]$ into k folds, F_1, F_2, F_3,..., F_k such that $F_1 \cup F_2 \cup F_3$,...$\cup F_k$ and $F_i \cap F_j = \emptyset$ for $i \neq j \in [1, 2, 3, \ldots, k]$
4. For all $i = 1$ to k
   (a) Test Data Indices = $I_{te} = F_i$
   (b) Train Data Indices = $I_{tr} = [1, 2, 3, \ldots, n] \setminus F_i$
   (c) Train the model F_i using $X(I_{tr})$ and $Y(I_{tr})$
   (d) Calculate performance P_i of F_i using $X(I_{te})$ and $Y(I_{te})$
5. Performance $P_{k-fold} = \dfrac{\sum_{i=1}^{k} P_i}{k}$

The second most commonly faced issue with DNNs is the computation time. The use of the right set of hyper-parameters that fit the training data, such as the size (number of layers and number of units per layer), the learning rate, and initial weights are very crucial in training a DNN as the performance of the model depends on it. At the same time searching through the entire configuration space for the optimal set of parameters, is not feasible due to the cost in time and computational resources. Certain workarounds like using mini-batches of training data where the gradient is computed on a mini-batch at once, rather than on individual examples, do speed up the computations. But, the most commonly used workaround which gives a significant speedup is using *GPU-Accelerated*

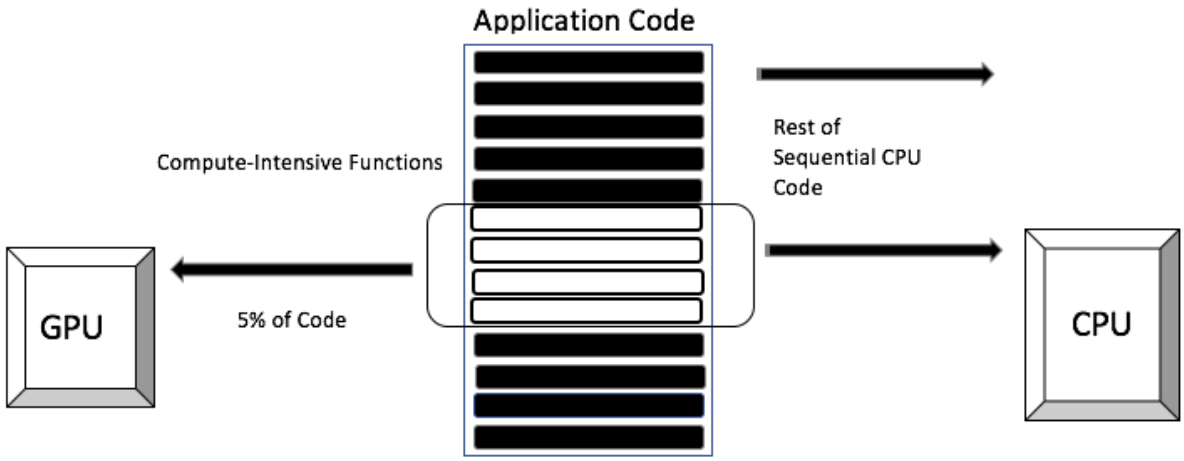


Figure 3.4: GPU-Accelerated Computing

Computing. Pioneered by NVIDIA in 2007, GPU-accelerated computing is the use of a graphics processing unit (GPU) together with a CPU to accelerate DNNs [9]. Unlike a CPU, which consists of a few cores optimized for sequential serial processing, a GPU has a massively parallel architecture consisting of thousands of smaller cores designed for handling parallel workloads. GPU-Accelerated computing offloads the compute-intensive portions of the training process to the GPU, while the remaining process still runs on the CPU. Figure 3.2 depicts how GPU acceleration works. Most of the DNN libraries (like *Tensorflow* [16], *Keras* [17], *Caffe* [3], and *Theano* [18]) support multi-GPU implementation, allowing users to significantly speed up the entire training process. As our study involves empirically examining different CNN models with varying sample sizes and text preprocessing techniques on different large open source projects, we have used GPU-Accelerated computing to speed up our training process.

3.2.1 Deep Learning For Bug Localization

Given the fact that RNNs make more intuitive sense in dealing with NLP-related problems, it seems that CNNs would not do well on textual data. But this is not the case in reality. CNNs have been proved to perform well on NLP tasks like Sentiment Analysis [132], Spam Detection [125], and Topic Categorization[56]. A recent study in bug lo-

calization has compared the performance of CNN and LSTM models using the Pairwise Learning-To-Rank approach and the results were encouraging [126]. The corpus of a bug report and that of a source file are combined into a single sentence and encoded using One-Hot Encoding. This encoded data is used to train the classification model. The architecture of the CNN model was adapted from [132]. As for the LSTM model, the architecture has been adapted from [51]. After training the CNN and LSTM models, the authors concluded that LSTMs performed better than the CNN models, but the difference in performance was not significant. This can be attributed to the fact that bug localization datasets do not contain pure natural language tokens but rather a mix of natural language tokens, domain-related tokens, and other tokens (like source code identifiers), which are neither natural language nor domain-related. This shows that the sequence, i.e., the order of tokens in a sentence does not have a significant effect on the performance of the model in bug localization. Hence losing information about locality (which is prevalent in a CNN model) does not deteriorate the performance of the model significantly. This makes the CNN models worth exploring on bug localization data.

Another advantage of using CNNs over LSTMs for bug localization is the memory and speed. CNNs are faster and easier to train than LSTMs. Also, LSTMs need more computation power and Random Access Memory (RAM) compared to a CNN. Taking into account the above stated points, we have explored the CNN models on different open source bug localization datasets, to empirically examine them. A more detailed explanation about CNNs and the architecture of the CNN model used in our study, is given in the next section.

3.3 Convolution Neural Nets (CNNs)

In this section we discuss basic concepts of CNNs.

CNNs are responsible for a major breakthrough in the field of Image Classification [65]. They form the core of most of the Computer Vision systems in today's world, from Facebook's automatic photo tagging to Waymo's self-driving cars. As mentioned in the previous section, recent research has shown that CNNs perform well on NLP-related tasks. A CNN consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of convolutional layers, pooling layers, and

fully connected layers [71].

In simplest terms, *convolution* can be visualized as a sliding window function applied to a matrix. Figures 3.5 illustrates convolution operations on an image with a 3×3 filter. In this case, the input is an image and each entry in the input matrix corresponds to a pixel (0 for black and 1 for white). The sliding window is called the *kernel, filter or feature detector*. Each of the values in the filter are multiplied element-wise with the original matrix. The result is then summed up to get the final value. The same operation is repeated with each element by sliding the filter over the whole matrix to get the result of the full convolution.

CNNs consist of several layers of convolutions with non-linear activation functions like *Rectified Linear Units (ReLU)* or *Tanh* applied to the convolution results. Unlike the traditional fully connected networks, where we connect each input neuron to each output neuron in the next layer, in CNNs each region of the input is connected to a neuron in the output. These local connections are a result of convolutions applied to the input to compute the output. Each layer in a CNN consists of hundreds or even thousands of filters of varied sizes. By training a CNN, we enable it to learn the values of its filters based on the task that we want to accomplish. For instance, in Image Classification a CNN may learn to detect edges from raw pixels in the first layer and then use the edges to detect simple shapes in the second layer and then use the shapes to learn higher-level features. The last layer in such cases is a classifier that uses the higher level features.

Figure 3.6 shows the architecture of a CNN for Image Classification. Another critical aspect of a CNN is the *pooling* layer. These layers are applied after the convolution layers. They subsample the input from convolution layer and the most common method used for this is *max pooling*. It computes the global maximum over feature maps, resulting into a feature vector. There are a couple of reasons why pooling is done. The first is that pooling provides a fixed output matrix, which is required for classifying the data (image or text). In case of textual data, pooling allows us to use variably sized sentences and variable sized filters and still get an output of similar dimensions every time, to feed into a classifier. The second reason is that pooling reduces the output dimensionality, but at the same time keeps the required important features. This is more relevant in cases where we encode our input sentences by simply converting each word in a sentence to its index in the dictionary. This results in large dimensional matrices which are very sparse.

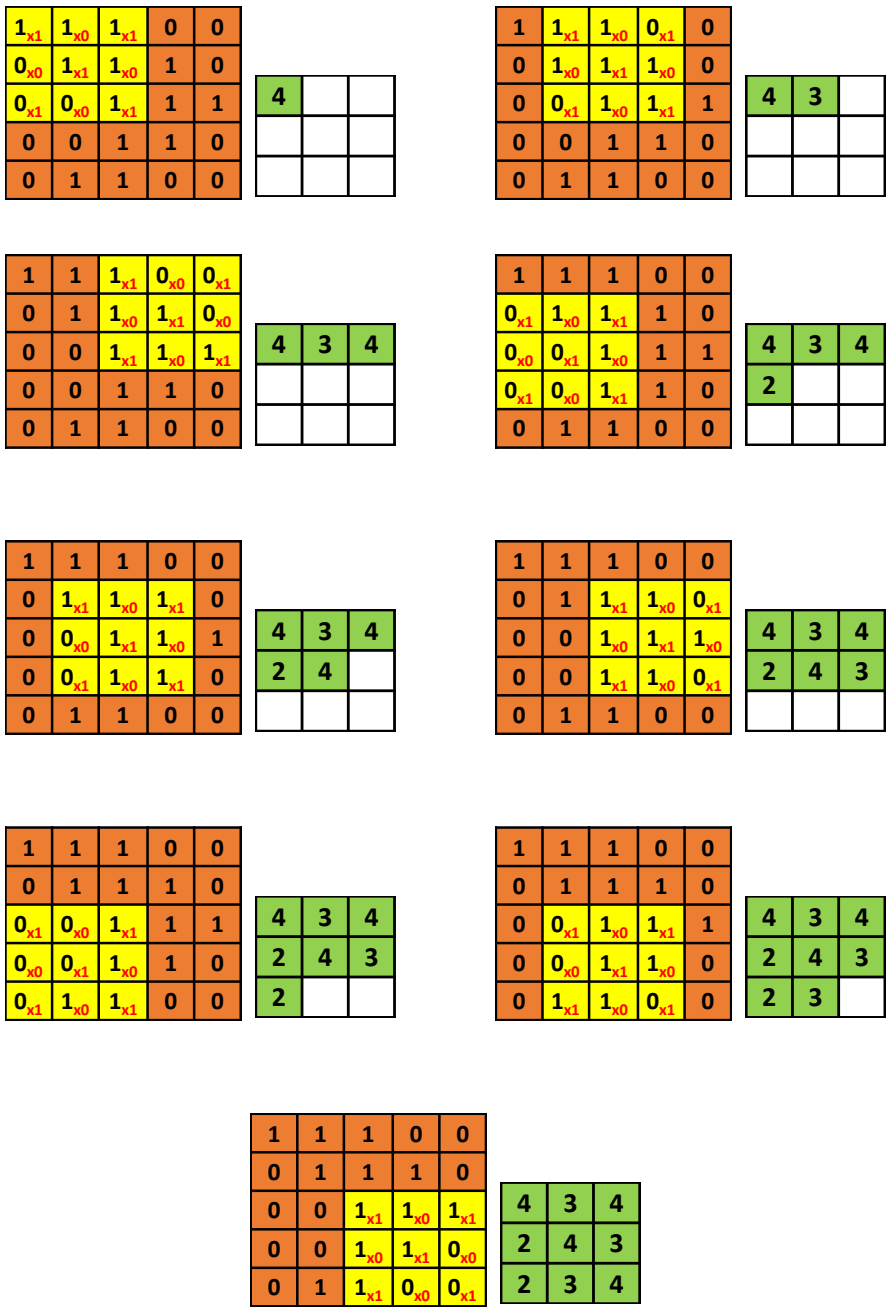


Figure 3.5: Convolutions With 3×3 Filter. The red-coloured matrices are the inputs, with the kernel/filter shaded in yellow. The subscripts in the yellow region are the values of the kernel. The result of convolution in each step is shown to the right of the input matrix in green.

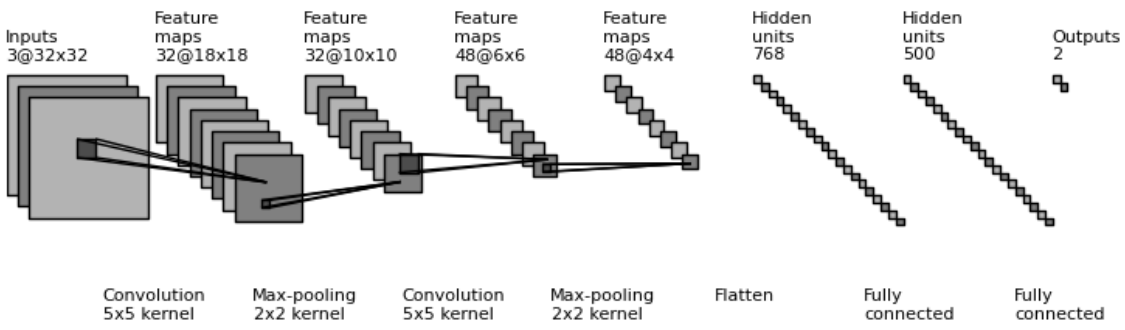


Figure 3.6: Image Classification Using CNN [4]

Pooling in such cases greatly helps in reducing the dimensions and yet, keep the salient features in the data. A detailed explanation of the different pooling strategies is given in the next section.

Local Invariance is another important aspect of CNN. Local Invariance means that the information about the exact occurrence of a feature is lost. This occurs because of the max pooling operation. In image-related applications, this does not affect the performance of the model. For example, consider the case of Image Classification where the task is to identify the image of a dog. We do not care *where* the dog occurs in the image as long as CNN can identify the image as that of a dog. The same may apply even for some types of textual corpora. For example, for a corpus which does not consist of natural language words/tokens, i.e., where the order of word in a sentence is not essential, local invariance does not affect the model. This is not the case for corpora which consist of pure natural language words/tokens where the occurrence of a word in a particular sentence is crucial. Hence in such cases, we lose global information about locality, but the filters capture local information. Despite of this, CNNs applied to pure natural language data perform quite well. We will discuss more about how CNNs perform on textual data in the next section.

Compositionality is yet another important feature of a CNN. As explained earlier, CNNs are feedforward neural networks, where each neuron in a layer receives input from the neurons in the previous layer. These local receptive fields, allow CNN to recognize more and more complex patterns in a hierarchical way, by combining lower-level,

elementary features into higher-level features. This property is called Compositionality.

Filter Size is another tunable hyper-parameter of a CNN model. It refers to the dimensions of the kernel or filter used for the convolution operation in the model. The shaded yellow region of the input "Image" matrix in Figure 3.5 is a convolution filter with size 3×3. In simple terms, the filter size tells us the number of neighbours in the input matrix that can be seen by each neuron, when processing the current layer. If the filter size is 3×3, that means each neuron can see towards it left, right, top, down, upper left, upper right, lower left, and lower right, i.e., a total of 8 neighbour's information. Typically in NLP-related applications, filters that slide over the full rows of the input matrix are used. This means that the width of the filters is kept similar to the width of the input matrix.

Number Of Filters is another hyper-parameter which gives the filter count to be used in each filter region. Naively, one can think of filters as feature detectors. The significance of the number of feature detectors intuitively tells us the number of features that the network can potentially learn. Also, each filter generates a feature map, which in turn allow the network to learn the explanatory factors within the data. Hence, the more the number of filters, the more will be the features that will be exposed to the model. But this does not mean that increasing the filter count is always beneficial to the model, as it might result in overfitting of the training data after hitting a saturation point. We discuss more on this in the next section.

The *Stride Size* defines by how much we want to shift the filter at each stride. Stride Size of 1 is typically used, but larger stride leads to a fewer applications of the filter and hence a smaller output size.

Activation Function (or non-linearity) takes the the result of the convolutions and performs a certain fixed mathematical operation on it. Some of the commonly used activation functions are *Sigmoid, Tanh,and ReLU*. Each of these activation functions are explained in detail below.

- Sigmoid: The Sigmoid non-linearity is shown in Figure 3.7. It takes a real-valued number and "pushes" it into a value between 0 and 1. This means that large negative numbers become 0 and large positive numbers become 1. It has the following mathematical form:

$$\sigma(x) = 1/(1 + e^{-x}). \tag{3.10}$$

- Tanh: Figure 3.8 shows a Tanh Function. This non-linear function "squashes" a real-valued number to the range [-1, 1]. Unike the Sigmoid function, the output of Tanh is zero-centered. Therefore, in practice the Tanh non-linearity is always preferred to the Sigmoid non-linearity. It is defined as:

$$\tanh(x) = 2\sigma(2x) - 1. \tag{3.11}$$

- ReLU: The Rectified Linear Unit (ReLU) is a very popular non-linear activation function which is being frequently used in the last few years. Figure 3.9 shows the ReLU function. In ReLU, the activation is simply thresholds at zero. The advantage of using ReLU over Tanh and Sigmoid functions is the computation cost. Tanh/Sigmoid neurons involve expensive operations (like exponential function), whereas the ReLU can be implemented by thresholding a matrix of activations at zero. Yet another advantage is that, ReLU was found to accelerate the convergence of SGD compared to the other two functions [65]. It is given by:

$$r(x) = \max(0, x). \tag{3.12}$$

- Softplus: The Softplus function is a newer function compared to Sigmoid and Tanh functions, and was first introduced by Dugas et al. [42] in 2001. Figure 3.10 illustrates the Softplus function. Unlike Sigmoid and Tanh functions, which have upper and lower limits, a Softplus function's output is in the range $(0,\infty)$, i.e., it has a lower limit but no upper limit. It is given by:

$$s(x) = \ln(1 + e^x). \tag{3.13}$$

3.3.1 On Textual Data

As discussed in Section 2.2, recent research has shown that CNNs perform well on textual data for many text-related tasks like sentiment analysis, text classification, question answering, and machine translation. This section discusses in detail how CNNs process the textual data.

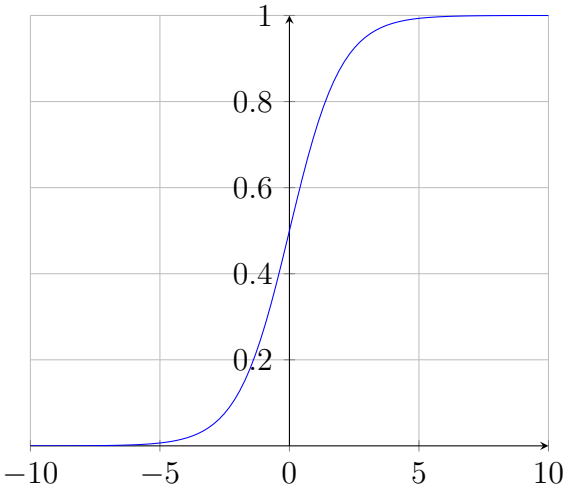


Figure 3.7: Sigmoid Function

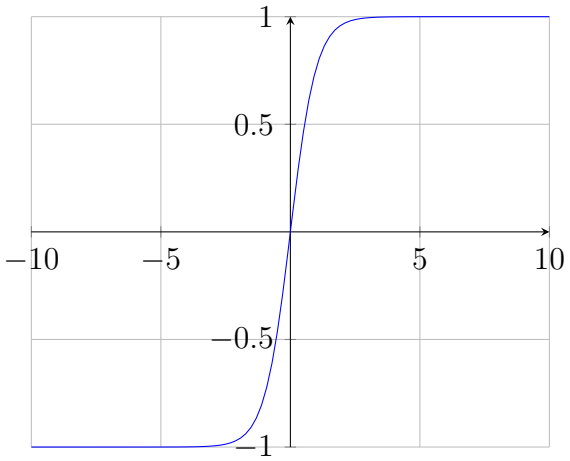


Figure 3.8: Tanh Function

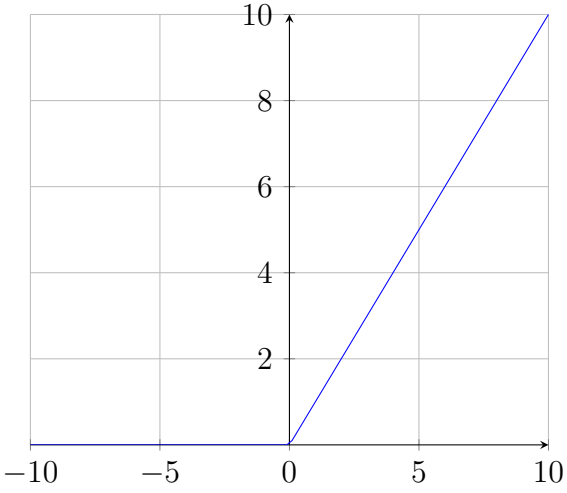


Figure 3.9: ReLU Function

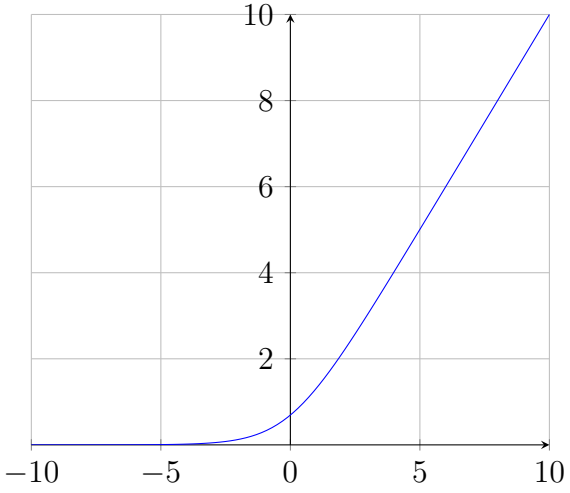


Figure 3.10: Softplus Function

As explained above, Local Invariance and Compositionality are two important features of CNNs. CNNs perform well on textual data due to these two features. In text, n-grams can be constructed from lower-order features and ordering is crucial locally but not at the document or record level. For example, in trying to classify a movie review as positive or negative, the location of "not bad, quite good" in the sentence is not important. We need to only capture local order, i.e., the fact that "not" precedes bad and so forth. It is important to note that CNNs are not able to encode dependencies which are long in range. This is the reason that for tasks like language modelling where long-range dependencies matter, RNN (more specifically LSTM) perform better.

In case of textual data, the input to a CNN would be sentences or documents represented as a matrix. Each row of the matrix represents a token (which is typically a word, though it could also be a character) in a sentence. This means each row is a vector that represents a word. As mentioned in the previous sections, these vectors can be encoded using Word2Vec or GloVe word embeddings. These vectors could also be One-Hot vectors that index the word into a vocabulary. For example, if we have a 100 word sentence using a 50-dimensional embedding, the input matrix would be a 100×50 matrix. In case of One-Hot encoding, if the dictionary size is 70, then the input matrix would be a 100×70 matrix. In image-related applications, the filters slide over local patches of an image, but in case of text, filters typically slide over the full rows of the matrix (words). It means that when using CNNs for NLP-related tasks, the width of the filter is kept similar to the width of the input matrix [132]. This is typically the case when the higher One-Hot dimensionality of the input data is reduced beforehand to a lower dimension using Word2Vec or GloVe encodings.

We now discuss the architecture of the CNN model used in our study. Figure 3.11 [135] illustrates the CNN model used in our study. The architecture of this model has been proposed by Yoon Kim in [132] (Figure 3.12). In Figure 3.11, d is the length of the dictionary, i.e., the number of unique words/tokens in the entire corpus of our dataset. Hence the dimensions of the input sentence matrix will be equal to $n \times d$, where n is the maximum sentence length and d is the dictionary size. It is important to note that we pad all the sentences with a single unique token to make the sentence length of all the documents equal to n. There are three filter regions of size 2, 3 and 4, each of which have 2 filters. Each filter performs convolution on the input sentence matrix and generates feature maps of variable length, i.e., 2 feature maps for each region size. After this step,

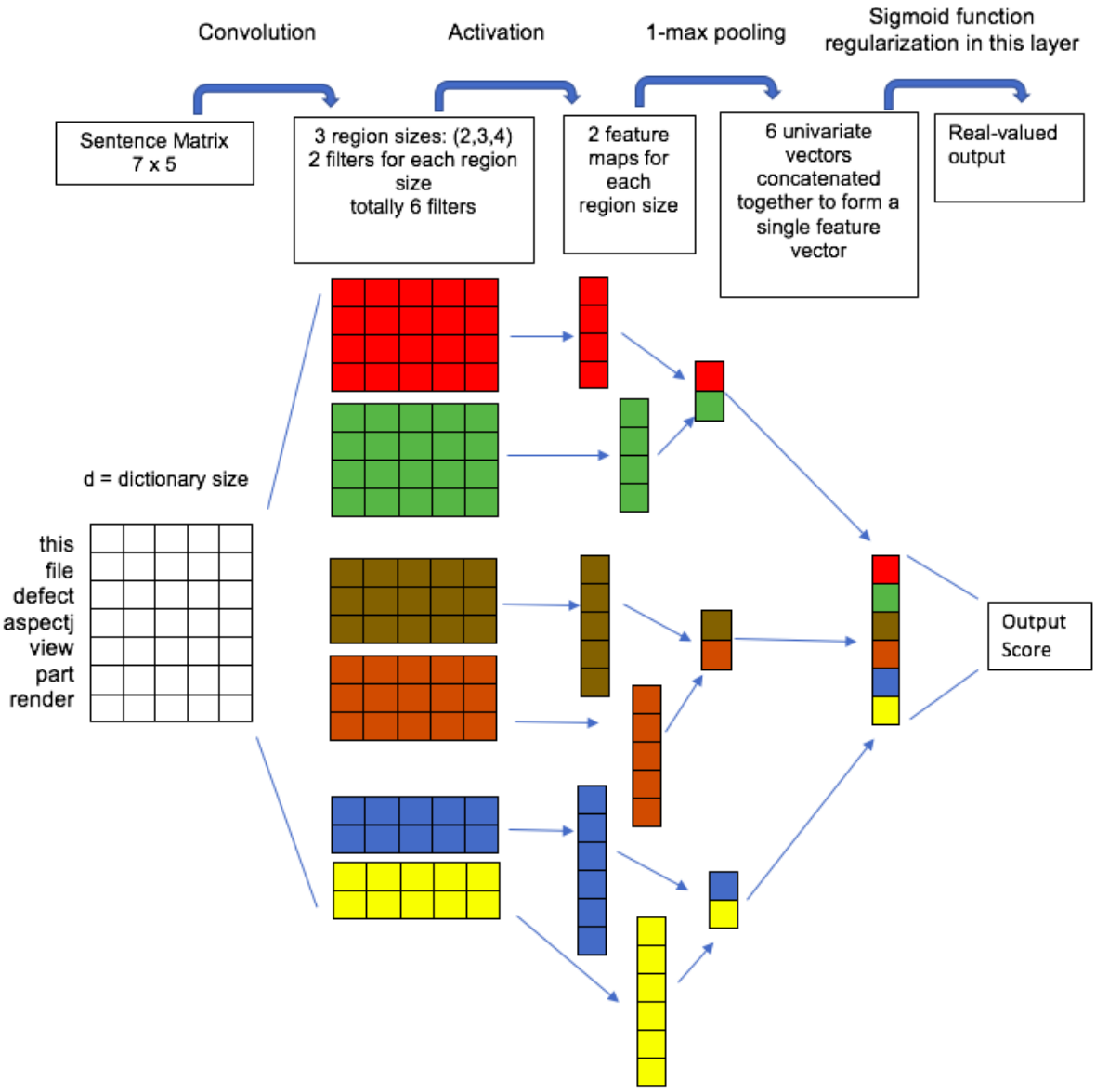


Figure 3.11: Architecture Of The CNN Model [135]

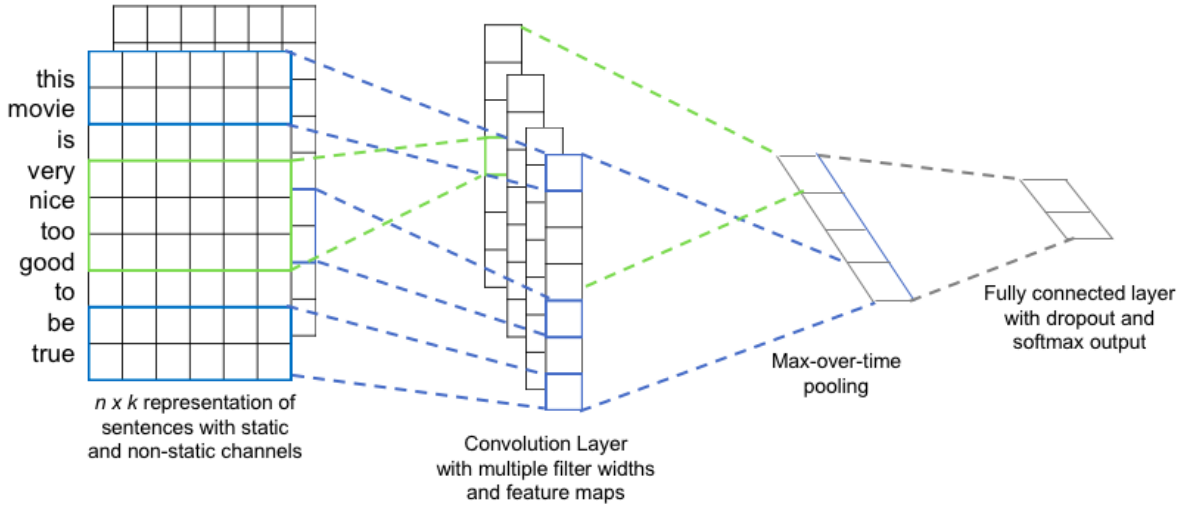


Figure 3.12: CNN For Sentiment Analysis [132]

1-max pooling is applied to each of these feature maps, which results in 6 univariate vectors. These vectors are then combined to form a single feature vector which is then fed into a sigmoid layer. The sigmoid layer uses the feature vector to produce a real-valued output which gives the degree of likeliness that the documents belongs to the positive class (given that this is a binary classification problem). As mentioned in the previous section, convolutions and pooling operations lose information about the local order of words, but in spite of this, the network in Figure 3.11 performs well compared to other machine learning models like *Support Vector Machines* for sentiment analysis task on the Movie Review dataset [87].

The only change we made to the original architecture of Kim, is the use of a sigmoid function instead of softmax function. The sigmoid layer generates a real-valued score which gives the degree of likeliness that a particular document belongs to the positive class. As explained in the first section of this chapter, we are using the Pairwise Learning-To-Rank approach to localize bugs. Hence in order to convert the binary classification model to a Learning-To-Rank problem, we need a score which helps in ranking the documents. In this case, for a given bug report, we rank its corresponding source files based on this score. The higher the score, the higher is the degree of likeliness that the particular source file is linked to a bug report.

3.3.2 Effect Of Hyper-Parameters

We now briefly discuss how each of the hyper-parameters affect the performance of a CNN model for text classification. Zhang et al. [135] performed a sensitivity analysis using the network shown in Figure 3.11, on nine sentence classification datasets which are: the Movie Reviews dataset (MR) [87], Stanford Sentiment Treebank dataset (SST-1) [106], SST-2 dataset (derived from SST-1, but pared to make it binary classification problem), Subjectivity dataset (Subj) [87], Question classification dataset (TREC) [74], Customer review dataset (CR) [53], Opinion polarity dataset (MPQA) [122], Opinosis dataset (Opi) [46], and Irony dataset [117].

Input Word Vectors

In order to train a CNN model on textual data, we need to convert the words in the sentences or documents into some set of word vectors. This is possible by encoding the words. The kind of encoding applied to the input sentences has an impact on the performance of the model. A brief explanation of these word encodings (also called word embeddings) is given below.

One-Hot Encoding: This is a straightforward way to convert the words into numeric vectors. Each word is converted into a sparse representation with only one element of the vector set to 1, the rest being 0. For each token, its index in the vocabulary dictionary defines the position of the one-hot element in the resultant vector. For example, consider a dataset with V as the vocabulary, S as a sentence in the dataset and H as the resulting One-Hot encoded vector.

```
V = {"bug", "defective", "file", "is", "this"}

S = "This file is defective"

H = [00001   00100   00010   01000]
      this    file    is   defective
```

Word2Vec Encoding: This encoding maps the high dimensional One-Hot style of representation of words to a low dimensional vector. The Word2Vec model was first proposed by Mikolov in [81]. It is a shallow, two-layer neural networks that are trained

to reconstruct linguistic contexts of words. This encoding proved very effective in all NLP-related tasks [132]. There are two different approaches for training a Word2Vec model: *Skip-gram* and *Continuous Bag Of Words*(CBOW). Skip-gram involves taking an input word and then attempting to estimate the probability of other words appearing close to that word. CBOW takes the context words as input and tries to find the single word that has the highest probability of fitting that context.

GloVe Encoding: This encoding was proposed by Pennington et al. [88]. GloVe stands for *Global Vectors Encoding*. This model is trained by aggregating global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space. GloVe tries to capture the counts of overall statistics of how often a word appears in the context of another word whereas Word2Vec captures co-occurrences, one window at a time.

For natural language data, Word2Vec or GloVe encoding proves better than One-Hot encoding, however different representations perform better for different tasks [135]. For natural language sentence classification, One-Hot encoding performs poorly compared to the other two encodings. However this may not be the case if the training data is very large [135]. Nevertheless, as some prior works like Kim et al. [132] on sentence classification using CNN showed that Word2Vec gives the best performance compared to the other form of encodings, we first experimented with Word2Vec encoding on our data. The resultant models performed poorly, which could be due to the fact that our corpus has more domain-related words/tokens than natural language words or tokens. We then experimented on One-Hot encoding of the corpus, which proved to be performing better than the Word2Vec encoded models. So we One-Hot encode the data, before feeding them into the CNN model.

Filter Region Size

The filter region size has some effect on the performance of the CNN models. Researchers [135] have experimented on the Movie Reviews dataset [87] with multiple sizes for the filter region. Table 3.1 shows the accuracy of the CNN model with multiple region size. The change in accuracy can be seen with change in the filter region size, with filters (7, 7, 7, 7) giving the maximum accuracy in this case. This shows that the filter region size is a hyper-parameter which has an effect on the performance and hence may need to be

Multiple Region Size	Accuracy(%)
(7)	81.65
(3,4,5)	81.24
(4,5,6)	81.28
(5,6,7)	81.57
(7,8,9)	81.69
(10,11,12)	81.52
(11,12,13)	81.53
(3,4,5,6)	81.24
(6,7,8,9)	81.62
(7,7,7)	81.63
(7,7,7,7)	**81.73**

Table 3.1: Effect of filter region size with several region sizes on the Movie Reviews dataset [87, 135]

tuned when training a CNN model. In this study, we use four filters with region size of (2, 3, 4, 5). This particular region size proved to be better compared to any other region size, for training a CNN model on bug localization data [54].

Number Of Feature Maps

The best number of feature maps depends on the type of dataset. However, the experiments made in [135] show that increasing the number of maps beyond 600 yields very marginal returns, and often deteriorates performance which is due to the model overfitting the training data. This means that when tuning this parameter, a search over a space of 100 to 600 feature maps is recommended. Given the fact that increasing feature maps increases the training time and may also give a worse performance, it is more prudent to tune over the range mentioned above. In this study, we use 100 feature maps. This particular number of feature maps, has been used widely in previous CNN models for sentiment classification [132] and also in bug localization studies [54, 126].

Activation Function

Zhang et al. [135] experimented on various activation functions like ReLU, Tanh, and Softplus. They have also considered a case, where they do not use any activation function.

Dataset	Tanh	Softplus	Iden	ReLU
MR	81.28	80.58	**81.30**	81.16
SST-1	47.02	46.95	46.73	**47.13**
SST-2	**85.43**	84.61	85.26	85.31
Subj	**93.15**	92.43	93.11	93.13
TREC	91.18	91.05	91.11	**91.54**
CR	84.28	83.67	**84.55**	83.83
MPQA	89.48	**89.62**	89.57	89.35
Opi	**89.35**	64.77	65.32	65.02
Irony	**67.62**	66.20	66.77	66.46

Table 3.2: Performance of different activation functions [135].

They denote this by the identity function, "Iden". The performance in terms of accuracy, of different activation functions, on all the datasets used in the study is shown in Table 3.2. It can be observed from the results, that for 8 out of 9 datasets, the best activation functions is either Iden or ReLU or Tanh. For some datasets, the performance of Tanh is better than the other activation functions. This can be due to its zero centering property [135]. As for ReLU, it is being more widely used recently by deep learning researchers, due to its non-saturating form compared to a Sigmoid activation, and also because it has been known to accelerate the convergence of SGD [65]. Another important observation, that can be made from the results in Table 3.1 is that for some datasets, not applying any activation function (i.e., Iden) gives better results. This shows that in some cases, linear transformations capture the correlation between the input word vectors and the output label better than non-linear transformations, though this cannot be said for a network with multiple layers [135]. In our study, we have used ReLU activation function, as it has been show to give good performance in previous bug localization studies [54, 126].

Pooling strategy

There are many pooling strategies that can be applied on a CNN model. Zhang et al.[135] have experimented on each of the below pooling strategies.

- *1-Max Pooling*: 1-Max pooling computes the global maximum over feature maps resulting into a feature vector of length 1 for each filter.
- *Local Region Pooling*: This pooling strategy pools over small equal sized local

regions instead of the entire feature map. This kind of pooling strategy was first applied by Boureau et al. [31]. Every small local region of the feature map generates a single number from pooling. These numbers are then concatenated to form a feature vector for one feature map.

- *k-Max Pooling*: This pooling strategy is similar to the one applied by Kalchbrenner et al. in [59]. In this method, the maximum k values are computed from the entire feature map, and the relative order of these values is preserved.

- *Average Pooling*: In this pooling strategy, instead of the maximum, the average of the regions is computed.

The analysis done by Zhang et al. [135] on the above pooling strategies showed that 1-max pooling consistently performed better than the other strategies on the sentence classification task. This can be attributed to the fact, that the location of predictive contexts does not matter, and certain n-grams in the sentence can provide more information to the model on their own, rather than when considered jointly with the entire sentence. In our study, 1-max pooling strategy was employed.

3.4 SimpleLogistic

As explained in the introduction section of this chapter, the goal of our study is to examine the effectiveness of a deep learning-based model in meeting the expectations of the practitioner and also to compare its performance with a traditional machine learning model. In other words, we are also comparing a non-linear learning model like CNN with a linear generalized logistic model called SimpleLogistic. We trained the SimpleLogistic model on the same set of datasets and the experimental setup, that has been used to train the CNN models, to make a fair comparison between the two.

The SimpleLogistic model was proposed by Landwehr et al. [69]. It is a standalone logistic regression model with in-built attribute selection. Landwehr et al. built these regression models using the *LogitBoost* [45] algorithm which selects a subset of attributes from the data. The experiments conducted by Landweher et al. proved that these models are compact and perform better than the other state-of-the-art classifiers.

Logistic regression (LR) models the posterior class probabilities $P_r(G = j|X = x)$ for the J classes via linear functions in x, where x is an independent variable. It ensures that they sum up to one and remain in the range of $[0, 1]$. The model is of the form

$$P_r(G = j|X = x) = \frac{e^{F_j(x)}}{\sum_{k=1}^{J} e^{F_k(x)}}, \tag{3.14}$$

where $F_j(x) = \beta_j^T \cdot x$. The estimates for β_j is found by approaching the maximum likelihood solution iteratively through numeric optimization algorithms.

LogitBoost is one such iterative algorithm. Friedman et al. [45] proposed this algorithm for fitting additive logistic regression models by maximum likelihood. The pseudo code for the LogitBoost algorithm [45] for J classes is given below:

1. Start with weights $w_{ij} = 1/n$, $i = 1, \ldots, n$, $j = 1, \ldots, J$, $F_j(x) = 0$ and $p_j(x) = 1/J$ $\forall j$.

2. Repeat for $m = 1 \ldots M$:

   (a) Repeat for $j = 1 \ldots J$:

      i. Compute working responses and weights in the j^{th} class

$$z_{ij} = \frac{y_{ij}^* - p_j(x_i)}{p_j(x_i)(1 - p_j(x_i))}, \tag{3.15}$$

$$w_{ij} = p_j(x_i)(1 - p_j(x_i)). \tag{3.16}$$

      ii. Fit the function $f_{mj}(x)$ by a weighted least-squares regression of z_{ij} to x_i with weights w_{ij}.

   (b) Set $f_{mj}(x) \leftarrow \frac{J-1}{J}(f_{mj}(x) - \frac{1}{J}\sum_{k=1}^{J} f_{mk}(x))$, $F_j(x) \leftarrow F_j(x) + f_{mj}(x)$.

   (c) Update

$$p_j(x) = \frac{e^{F_j(x)}}{\sum_{k=1}^{J} e^{F_k(x)}}. \tag{3.17}$$

3. Output the classifier $\text{argmax}_j F_j(x)$.

y_i is the class label of example x_i, y^*_{ij} gives the observed class membership probabilities for instance x_i and is defined as:

$$y^*_{ij} = \begin{cases} 1, & \text{if } y_i = j \\ 0, & \text{if } y_i \neq j \end{cases}. \tag{3.18}$$

The $p_j(x)$ are the estimates of the class probabilities for an instance x given by the model fit so far. LogitBoost performs forward stage-wise fitting, i.e., in every iteration, it computes the response variables z_{ij} that encode the error of the currently fit model on the training examples and then tries to improve the model by adding a function f_{mj} to the committee F_j, fit to the response by weighted least-squared error.

If we constrain f_{mj} to be linear in x, and continue to run the algorithm until convergence, then we get a linear logistic regression model. But, if we further constrain f_{mj} to be a linear function of only the attribute that results in a least squared error, then we arrive at the SimpleLogistic algorithm that performs automatic attribute selection. In SimpleLogistic model we use cross-validation to determine the best number of LogitBoost iterations. In this process, only those attributes that improve the performance of the model on unseen instances are included.

3.5 Evaluation Metrics

The following metrics have been widely used for evaluating the performance of the Bug-Localization models [64, 67, 96, 130, 137]. Also, as mentioned earlier, we are trying to solve the Learning-To-Rank bug localization problem through classification approach. So we also include the metrics used to evaluate the performance of our classifier, like binary classification accuracy, cross entropy loss, and AUC below.

Mean Average Precision (MAP):

MAP provides a single measure of quality of IR, when a query may have multiple relevant documents [100] (like in the case of bug localization, where a single bug report can have multiple relevant files). The IR model for bug localization will return a list of source files for a given bug report. An effective algorithm would return all the valid links close to

the top of the list [48]. MAP is computed as follows:

$$MAP = \sum_{i=1}^{n_2} \sum_{j=1}^{n_1} \frac{Prec(j) * t(j)}{N_i}, \tag{3.19}$$

where n_1, n_2 are the number of candidate source files and retrieved bug reports, respectively, and N_i is the number of relevant files to a bug report i, and $t(j)$ indicates whether the instance in rank j is relevant or not (buggy or non-buggy). $Prec(j)$ is the precision at the given cut-off rank j defined as

$$Prec(j) = \frac{Q(j)}{j}, \tag{3.20}$$

where $Q(j)$ is the number of relevant source files in the top j positions.

Mean Reciprocal Rank (MRR):

MRR is a statistic measure for evaluating an IR model that returns a list of possible documents (source files) to a sample of queries (bug reports), ordered by probability of relatedness/correctness. The reciprocal rank of a query response is the multiplicative inverse of its rank of the first correctly retrieved document: 1 for the first place, 1/2 for the second place, 1/3 for the third place and so on. MRR is the average of the reciprocal ranks of results for a sample of queries Q and is given by [91, 116]:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i}. \tag{3.21}$$

It is important to note that, unlike the MAP metric which considers all the relevant files in the retrieved list, MRR considers only the rank of the first relevant source file. The other relevant files in the list are ignored.

Top-k Rank:

It is the number of bugs whose relevant source files are ranked in the top k of the returned results. Given a bug report, if the top query results contain at least one source file that is relevant(buggy) to the bug report, then the particular bug is considered to be located.

The percentage of all such located bugs gives the Top-k Rank. The higher the Top-k Rank, the better is the performance of the bug localization model.

Binary Classification Accuracy:

Accuracy is one of the metrics used to evaluate the performance of a classification model. Accuracy is given by:

$$Accuracy = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{3.22}$$

More specifically for binary classification it is given by:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}, \tag{3.23}$$

where TP stands for True Positives, TN – for True Negatives, FP – for False Positives, and FN – for False Negatives.

In the bug localization context,

- TP = combinations of bug report and source code files, which are correctly classified by the model as 'linked' records.
- TN = combinations of bug report and source code files, which are correctly classified as 'not-linked' records by the model.
- FP = combinations of bug report and source code files, which are wrongly classified as 'linked' records by the model.
- FN = combinations of bug report and source code files, which are wrongly classified as 'not-linked' records by the model.

Though we use accuracy as a metric to evaluate our model, it is alone not sufficient to give a complete picture of the model's performance. This is especially for cases where we have an imbalanced dataset. Given that bug localization datasets are highly imbalanced (negative samples higher than positive samples), accuracy alone is not sufficient to evaluate the performance of the model. It is for this reason we also use the AUC metric, which is explained in detail below.

Area Under Curve (AUC):

AUC is the area under the Receiver Operating Characteristic Curve (ROC), which is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. AUC values lie between 0.5 and 1, where 0.5 denotes a bad classifier and 1 denotes an excellent classifier. AUC is not specific to the bug localization problem. It is used to measure the performance of any classification model, trained on imbalanced datasets. Bug localization is an imbalanced learning problem, as a particular bug report will have only few relevant files (positive instances) but can have many non-relevant source files (negative instances). The higher the AUC, the better is the ability of the model to identify the relevant files from all the other source files in the repository.

Cross Entropy Loss:

Cross entropy loss or log loss measures the performance of a classifier which outputs a probability score in the range [0, 1]. Log loss increases as the predicted probability of a sample differs from the actual label. For instance, if the model predicts a probability of .05 when the actual observation label is 1, then this would results in a high loss value. A perfect classifier will give a log loss of 0. Log loss penalizes the model heavily for predictions that are predicted with high confidence but are actually incorrect. For binary classification, the log loss is given by:

$$L = -[y \log(p) + (1 - y) \log(1 - p)], \tag{3.24}$$

where y is a binary indicator (0 or 1) which gives the actual class label of a sample and p is the model's predicted probability that a sample belongs to a class.

Chapter 4

Evaluation

In this chapter, we discuss the various experiments carried out in this study, followed by an in-depth analysis of our findings. Section 4.1 consists of the data extraction process for the five open source software projects used in the study. Next, we give some descriptive statistics and a brief analysis on the extracted data in Section 4.2. Section 4.3 consists of steps followed to preprocess the data and build the *traceability matrix*. In Section 4.4, we mention the experimental set up and the hyperparameters used to train the CNN and SimpleLogistic models. In Section 4.5, we give the results of our experiments on CNN and SimpleLogistic models on all the datasets. In Section 4.6, we address the research questions stated in Chapter 1 along with an adequate explanation for each. We conclude this chapter with the threats to validity in Section 4.7.

4.1 Data Extraction

In this study, we experimented on five open source benchmark datasets. The bug reports and bug-commit mappings are publicly available[1]. We extracted the actual source files based on the bug-commit mapping from the Git Hub [10] repository of each project. Table 4.1 contains the time interval between which the bug reports have been created and also the number of bug reports for each of the five projects. The bug reports for all the projects are also available on the Bugzilla bug tracking system and the source files are available in the Git Hub version control systems [5]. The datasets used in this study are

[1]http://dx.doi.org/10.6084/m9.figshare.951967

Project	Time Range	Bug Reports
AspectJ	2002/03 - 2013/12	593
Tomcat	2002/07 - 2014/01	1056
SWT	2002/02 - 2014/01	4151
Eclipse	2001/10 - 2013/12	5262
JDT	2001/10 - 2014/01	6269

Table 4.1: The list of open source projects used in this study

all benchmark datasets which have been used widely in previous bug localization studies [54, 60, 67, 68, 126, 137].

We now briefly describe each of the open source software projects used in this study. AspectJ [14] is an aspect-oriented extension for the Java programming language. It is available in Eclipse Foundation [8] open-source projects repository both as a stand-alone as well as integrated into Eclipse. The Eclipse Foundation is an open source community working to build a development platform consisting of the frameworks, tools and run-times needed for "building, deploying and managing software across the lifecycle" [6]. Apache Tomcat [1], often referred to as Tomcat Server, is an open-source Java Servlet Container developed by the Apache Software Foundation. The Standard Widget Toolkit (SWT) [13] is a graphical widget toolkit for the Java platform. Eclipse Platform User Interface [12] consists of several components, which provide the basic building blocks for user interfaces built with Eclipse. The Java Development Tools (JDT) [7] project provides the tool plug-ins that implement a Java Integrated Development Environment (IDE) supporting the development of any Java application, including Eclipse plug-ins.

As mentioned earlier, the source files for each project have been extracted from their respective GitHub repositories. A before-fix version of the source code package in the git repository has to be checked out for each bug report. This will enable us to get the actual version of the source file, before a fix has been made to it, to resolve the issue in the bug report.

We now iterate over all the commits; and for each commit, we checkout the before-fix version of the source code package. Then we find the difference between the before-fix version and the current version of the source code. For example, consider AspectJ bug *423257*. This bug was fixed at commit *dd88d21*. To check out the before-fix version

Bug ID : 339286
Summary : Toolbars missing icons and show wrong menus.
Description : The toolbars for my stacked views were: missing icons, showing the wrong drop-down menus (from others in the stack), showing multiple drop-down menus, missing the min/max buttons ...

Figure 4.1: Sample Eclipse Bug Report

```
public class PartRenderingEngine implements IPresentationEngine {
       private EventHandler trimHandler = new EventHandler()
       {
       public void handleEvent(Event event)
       { ...
           MTrimmedWindow window =
           (MTrimmedWindow) changedObj;
       ...
       }
   ...
   }
...
}
```

Figure 4.2: Code From PartRenderingEngine.java

of the source code package, we use the command `git checkout dd88d21~1` and then we find the diff for all changes made. Instead of indexing all the source files again, we index only the changed files, i.e., *Added*, *Modified*, or *Deleted* Java files between the two commits. The shell scripts used for the above extraction are included in Appendix A.

The end result of the above extraction process will be the bug reports and their corresponding buggy files. These buggy files can have one or more bug reports associated with them. For extracting the content of all the other source files, we clone the current git repository of the project and extract all the Java source files in it. We identify the buggy files from the entire list of source files and label them as *buggy*. The rest of the source files are labelled as *non-buggy*. Figure 4.1 shows a sample Eclipse bug report and

Project	Bug Reports	Linked Records	Non-Linked Records
AspectJ	593	2394	1,350,239
Tomcat	1056	2571	2,829,621
SWT	4151	8281	9,327,318
Eclipse	5262	10907	24,267,961
JDT	6269	16310	70,654,127

Table 4.2: Linked And Not-Linked Records

its buggy source file is shown in Figure 4.2.

Now that we have the buggy and also the non-buggy files, we create the traceability matrix. We create an $m \times n$ Cartesian product of the list of all source files (both buggy and non-buggy files) and bug reports. So if we have B bug reports and S source files, we end up with a total of $B \times S$ records. We identify the linked or the relevant records L in the $B \times S$ records, based on the bug-file mapping. The remaining records in the traceability matrix after removing the linked records will be the non-linked records, deemed NL. Thus, the number of non-linked records in the traceability matrix will be $NL = (B \times S) - L$. The linked records will be labelled as *positive* and the non-linked records will be labelled as *negative*. Table 4.2 shows the linked and non-linked records statistics for each project along with the number of bug reports.

4.2 Data Analysis

In order to address RQ5, we perform all our experiments on three variations of source files. They are as follows.

- *All Files*: In this case, we consider all the source files and all the linked records in the traceability matrix.
- *Buggy Files*: In this case, we consider only the source files which have at least one bug report linked to it in the traceability matrix. Also, all the linked records in the traceability matrix will be considered.
- *Very Buggy Files*: In this case, we reduce the set of buggy files, i.e., we consider only the set of source files which have more than one bug linked to it in the trace-

Project	Bug Reports	All Files	Buggy Files	Very Buggy Files
AspectJ	593	4439	2281	602
Tomcat	1056	2682	1041	437
SWT	4151	2249	1181	657
Eclipse	5262	4614	2868	1581
JDT	6269	11273	4610	1817

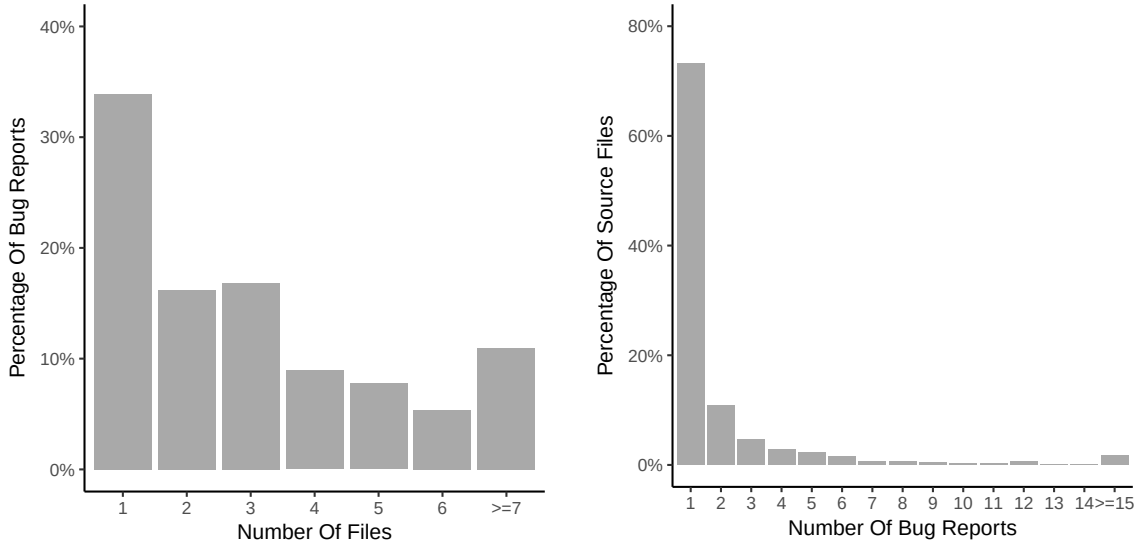
Table 4.3: Bug Reports And Source Files Statistics

ability matrix. The linked records in the traceability matrix, will also be reduced accordingly, i.e., we keep only the linked records of those source files which have more than one bug linked to them.

Table 4.3 shows the statistics for the datasets under study. Next, we analyze the linked records in our traceability matrix, i.e., the bug reports and the source files linked to them. Figure 4.3(a) shows the number of source files changed across the bug reports in AspectJ. As observed from the bar plot, for the majority of the bug reports (about 65%), there is more than one source file mapped to the bug report. This shows that to resolve an issue in a bug report, a developer will have to fix code in more than one source file. The last bar shows that about 12% of bug reports have more than 7 source files linked to them. Also, about 35% of bug reports have only 1 file mapped to them.

Figure 4.3(b) shows a bar plot for the number of bug reports associated with source files in AspectJ. This histogram clearly shows that the highest number of source files (more than 70%) are linked to not more than one bug report. The long tail of this histogram indicates the presence of outliers in the data, i.e., certain files with large number of bug reports associated with them. Figure 4.4(a) shows similar analysis on the Tomcat dataset. In case of Tomcat, the majority of bug reports (about 65%) are linked to one source file. Only about 5% of bug reports have more than 7 source files linked to them.

Figure 4.4(b) shows the number of bug reports associated with source files in Tomcat. Similar to AspectJ, the majority of files (about 58%) are mapped to one bug report. About 1% of files are linked to more than 15 bug reports. Also like AspectJ, the distribution is skewed with the long tails of the histogram indicating the outliers in the dataset (i.e. certain files associated with large number of bug reports). Similar plots

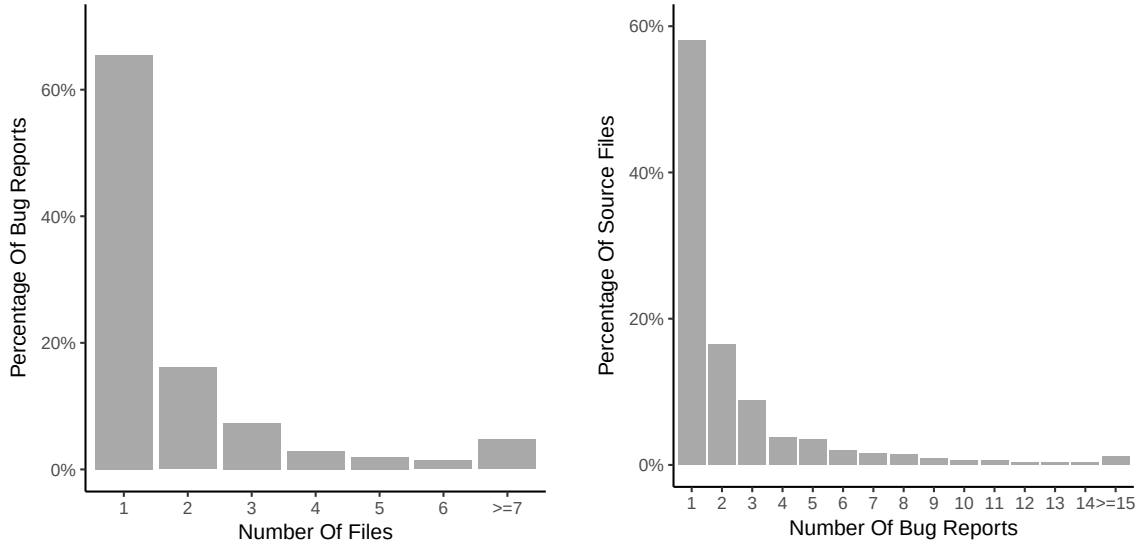


(a) Files Changed Across Bug Reports (b) Bug Reports Associated With Files

Figure 4.3: Analysis Of Linked Records - AspectJ

were obtained for the other three datasets - SWT, Eclipse, and JDT, and are included in Appendix A. It can be observed from all the figures that the distributions for the number of bug reports associated with files for all the five datasets are skewed with long tails indicating the presence of outliers (source files with larger number of bug reports linked to them).

It is important to note that all the datasets used in this study are highly imbalanced. This can be observed from Table 4.3. The number of non-linked records, as explained in Section 4.1, will be the product of source files and bug reports, with the linked records deducted from the resultant product. The number of non-linked records is 500 - 12,000 times greater than the number of linked records. The performance of a neural network classification model on a skewed (imbalanced) dataset tends to improve when the imbalance is reduced [80]. We treat for imbalance by assigning larger weights to the positive class, i.e., we penalize the models with a higher cost for every misclassification of a positive sample. Also, in order to examine how reducing sample size to treat imbalance affects the model, we experiment on the three variations of source files shown in Table 4.2.



(a) Files Changed Across Bug Reports (b) Bug Reports Associated With Files

Figure 4.4: Analysis Of Linked Records - Tomcat

4.3 Data Preprocessing

In this section, we discuss about the steps involved in preparing our data before training our models. After we extract the data as per the procedure stated in the previous section, we preprocess the corpus of bug reports (which is obtained from combining the summary and the description fields of the bug report (see Figure 4.1)) and the corpus of the source files. The following are the preprocessing steps:

- Punctuation and numbers: All punctuation symbols and numbers are removed from the bug report and source file corpus.
- Java-related keywords, comments and package names: We also removed all java language-related keywords and also the comments, copyright information, package names from the source file corpus.
- Camel case: Camel case is applied to all the words in bug report and source file corpus. For example the word 'aspectView' is split into 'aspect' and 'View'; the word 'WorkBench' is split into 'Work' and 'Bench'.

Project	Dictionary Size
AspectJ	5391
Tomcat	6449
SWT	10417
Eclipse	7583
JDT	8068

Table 4.4: Dictionary Size After Preprocessing

- Porter Stemming: Porter Stemming [89] is applied to all the words to reduce them to their stem form. All words which are deviationally related words like 'programming', 'programmer', 'programs' are converted to the same stem 'program'.
- Lowercase: All the words/tokens in the corpus are converted to lowercase.

The scripts used to parse the source code and preprocess the bug report and source code corpus are included in Appendices C and D. After the above preprocessing steps, we build the traceability matrix based on the heuristics specified in the previous section. After this step, we merge the corpus of each bug report and its corresponding source file in the traceability matrix. This will enable us to have a single text field for each positive or negative record in the traceability matrix.

We initially experimented on a combination of distinct words/tokens in bug report corpus with distinct words/tokens in source code corpus. We also experimented on the bag-of-words approach, where we combine the corpus of bug reports and source files and then remove duplicate words/tokens from the combined text field of each record in the traceability matrix. The bag-of-words approach outperformed the distinct words variation. Hence we adopt the bag-of-words approach, i.e., we keep only the distinct words or tokens in the combined bug report and source code text field in the traceability matrix for all the datasets. Table 4.4 shows the dictionary size for each dataset, i.e., the number of unique words/tokens in the entire traceability matrix after preprocessing.

4.4 Experimental Setup

In this section, we discuss the experiments performed in our study on the five open source datasets along with the exact setup used to train the CNN and SimpleLogistic models.

As mentioned in the previous sections, we perform all our experiments on three sets of variations of source files shown in Table 4.3 for both the CNN and SimpleLogistic models.

4.4.1 CNN Model

As mentioned in Chapter 3, the architecture of our CNN model is adapted from Kim et. al. [132]. For each dataset, we first set aside 10% of the data as test set and the remaining 90% will be used for training and validation. The 90% data are split into 10 folds. Then 10-fold cross-validation is performed, i.e., one fold will be used as the validation set and the remaining nine folds will be used to train the model. The same is repeated until each fold becomes a validation set once[2].

After each epoch, the trained model is tested on the validation set and its accuracy and loss are recorded. At this point, we also compare the validation accuracy of the model with the previously recorded validation accuracies. If at a particular epoch the validation accuracy is higher than its previous values, then we save the weights of the model and use the same model to predict on the 10% test set. The reason behind following this approach is due to the fact that the model which gives the best validation accuracy is more generalizable and will perform well on new/unseen data, i.e., the test set. At the end of each fold, the performance of the model on the test set is measured using the metrics explained in Chapter 3. The same process is repeated for all the folds.

Table 4.5 shows the hyperparameters used for training the model on all the datasets. The values for most of these hyperparameters like the input encoding, number of filters, filter size, number of hidden units, activation function, dropout rate, learning rate, and optimizer type have been adapted from [132] and [126]. As for the batch size hyperparameter, most similar CNN models in literature used batch sizes of 32 or 64. Also, as our datasets are large in size, batch size greater than 16 will lead to an overflow of the GPU memory (as available GPU memory is 12 GB per GPU). It is for this reason that for the larger projects (i.e., SWT, Eclipse and JDT) we use a batch size of 16. For the other projects, we use a batch size of 32.

As for the number of epochs needed to train the model, this was decided based on the observations made during training. We train the model till the network converges, i.e., when the training and validation accuracies are similar. Also, we make sure we do

[2]The 10-fold cross-validation has been explained in detail in Chapter 3.

Parameter	Value
Input Encoding	One-Hot
Filter Size	[2,3,4,5]
Number Of Filters	100
Number Of Hidden Units	100
Activation Function	ReLU
Drop-out Rate	0.5
Batch Size	16,64
Optimiser	Adam
Learning Rate	1e-4
Epochs	5,10,50

Table 4.5: Hyperparamters - CNN Models

not over-train the model (i.e., when training accuracy is very high, but the validation accuracy deteriorates), as this leads to overfitting. We explain more about overfitting and the steps taken to avoid it in the coming paragraphs.

As explained in Chapter 3, there are predominantly three types of word encodings used in the field of text analytics or NLP - One-Hot encoding, Word2Vec, and Glove. Kim's [132] work on sentence classification using CNN showed that Word2Vec gives the best performance compared to the other form of encodings. Hence we experimented with Word2Vec encoding on our data. The resultant models performed poorly, which could be due to the fact that our corpus has more domain-related words/tokens (stack trace in bug reports and all words/tokens in source code) than natural language words or tokens (users' or testers' comments in bug reports). We then experimented on One-Hot encoding (explained in Chapter 3) of the corpus. The One-Hot encoded models outperformed Word2Vec encoded models. This finding is in-line with the findings made by Huo et al. in [54, 126]. They have also used One-Hot encoding instead of Word2Vec when experimenting on different open source bug localization datasets.

As mentioned in the previous sections, all the datasets used in this study are highly imbalanced. But we train the model on a balanced dataset. We balance the training data by applying higher class weights to the smaller class (i.e. positive class) which will penalize the model for every misclassificaltion of a positive sample (linked record). However, it is important to note that neither the validation set nor the test set are balanced. During the training phase, after each epoch, we test the trained model on

the imbalanced validation set. We save the weights of the model that gives the highest validation accuracy across all the epochs in a particular fold. After all the epochs for a fold, the best model obtained, i.e., one with the highest validation accuracy will be the final model that is used to make predictions on the test set. The performance of the model is calculated based on the predictions made by the model on this test set. This gives us the performance of the model for the particular fold. We repeat the same process for all the folds. The final performance of the model is average of the performance of the model in each fold.

Another important issue which is commonly faced when training a machine learning or deep learning model is overfitting. As explained in Chapter 3, overfitting means that the model starts memorizing the training data instead of learning the abstract features or patterns in the data. When a model starts to overfit the training data, its performance on the test set will be low. The 10-fold cross validation approach is one effective way to combat overfitting. Apart from this, we also adopt two regularization methods called dropout and early stopping to avoid overfitting.

As explained in Chapter 3, dropout is a form of regularization used to avoid co-adaptation of the training units by randomly dropping out the hidden units with certain probability (0.5 in this case). Early stopping is another form of regularization, where we stop training the models once the validation set error starts to deviate by a large extent from the previously observed validation error values. If we continue to train the models past this point, the training accuracy improves at the cost of a deteriorating validation accuracy. Such models will give a high generalization error, making them unfit for new or unseen data. Figures 4.5 and 4.6 illustrate the accuracy and loss plots for AspectJ and Tomcat respectively.

The CNN model is implemented using the Keras deep learning library [17] with TensorFlow [16] as the back-end. The source code of the model is included in Appendix E. All the models were trained using multiple GPUs. I/O and other non-compute intensive operations were performed on a CPU and the compute-intensive operations were carried out on 2 - 3 GPUs (depending on the availability of Compute Canada resources).

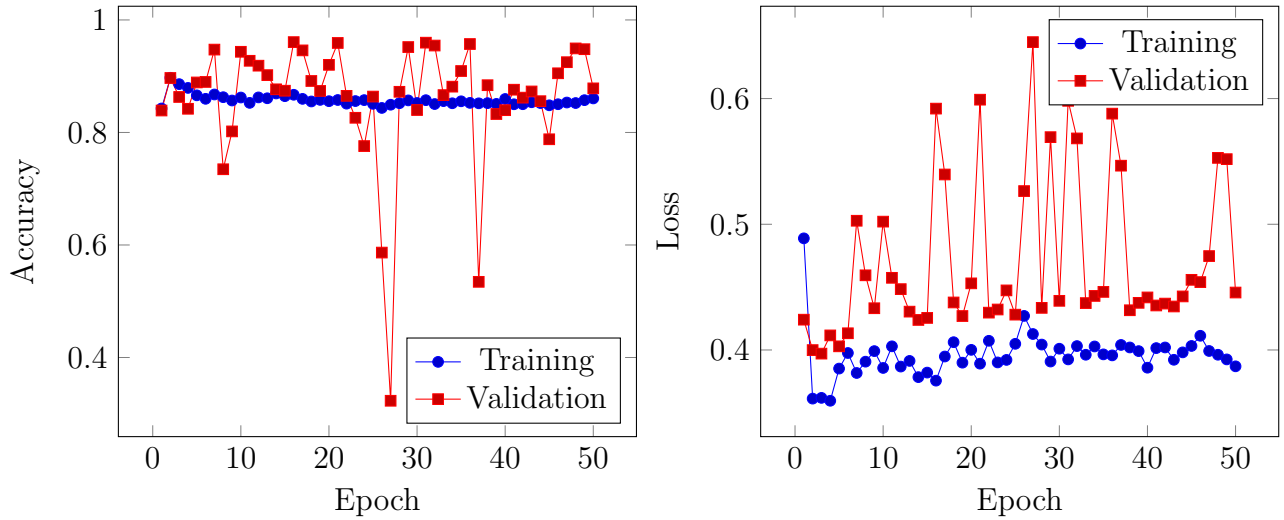


Figure 4.5: Epoch Vs Accuracy & Loss - AspectJ

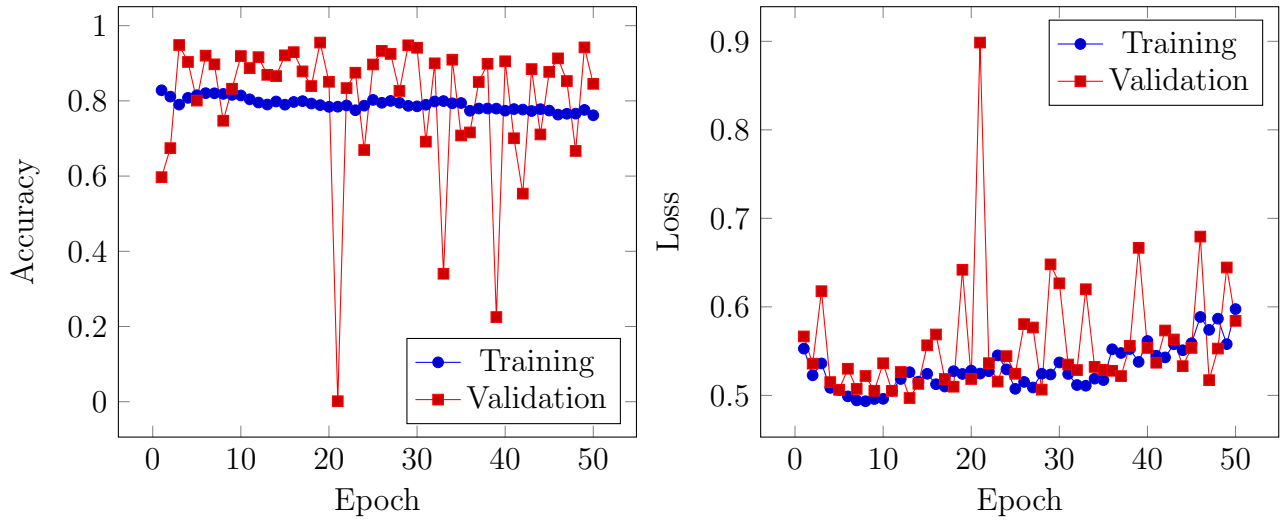


Figure 4.6: Epoch Vs Accuracy & Loss - Tomcat

Parameter	Value
Fixed Iterations (I)	0
Max Boosting Iterations (M)	500
Weight Trimming (beta)	0.0, 0.1

Table 4.6: Hyperparameters - SimpleLogistic Models

4.4.2 SimpleLogistic Model

We use Weka [20] command-line interface to run the SimpleLogistic models on all our datasets. The experimental set up is the same as that used to train the CNN models. In this case, we first convert the string attributes into numeric attributes representing the word occurrence information from the text contained in the strings using Weka's *StringToWordVector* [22]. Next, we apply a *Class Balancer* [19] to the training data, to reweight the instances in the data so that each class has the same total weight. Similar to the previous setup, the validation and the test sets are not balanced. After this step we apply the *SimpleLogistic* [21] models on the dataset with 10-fold cross validation. The hyperparameters used to train the SimpleLogistic models are given in Table 4.6.

In order to reduce the training time for the larger datasets, we follow a heuristic proposed by Sumner et al. [110] which states that weight trimming by a factor β of 0.1 improves the training time without affecting the performance of the model. This means, that only the training instances carrying $100 * (1 - \beta)\%$ of the total weight mass are used for building the simple linear regression model.

4.5 Results

In this section, we give a detailed explanation of the steps followed to compute the metrics mentioned in Chapter 3, i.e., AUC, MAP, MRR, Top-5 Accuracy. All the results in the study are obtained after testing the trained model on the 10% test set. The output of the CNN model is a score, indicating the degree of likeliness that a sample belongs to the positive class (linked). This means that the higher the score the higher is the probability that the particular combination of bug report and source file are linked. The same is the case with SimpleLogistic models.

As mentioned in Chapter 3, the bug localization problem is a Learning-To-Rank

IR problem which is solved using a classification approach. In order to estimate the performance of the model in classifying the instances, we use the AUC metric. For measuring the performance of the model in ranking the relevant source files for a bug report, we use IR metrics like MAP, MRR and Top-5 Accuracy. We do this by first filtering each record in the test set based on the bug report. For each bug report, we sort the source files based on the predicted scores obtained. Once we have the sorted list for all the bug reports in the test set, we calculate the following IR metrics:

- **MAP**: For every bug report, we sort its corresponding source files in the test set by descending order of the predicted scores. After this, we calculate AP as per equations 3.19, 3.20 for each bug report and find the mean of AP to get the MAP score.
- **MRR**: Similar to MAP, after preparing the ranked list of source files in the test set for each bug report, we calculate the RR as per equation 3.21. Next, we compute the mean of RR of each bug report, to get the MRR score.
- **Top-k Rank**: We get the Top-k Accuracy by calculating the percentage of bug reports for which at least a single relevant source file is present in the top k positions of the ranked list of source files present in the test set.

The training time and memory needed to train the models for the three variations of source files for all the datasets are given in Tables 4.7, 4.8, and 4.9, respectively. The AUC, MAP, MRR, and Top-5 Accuracy for all the datasets for the three variations of source files, for both the CNN and SimpleLogistic models are given in Tables 4.10, 4.11, and 4.12, respectively. As the training time and the resources needed to train the models are very difficult to obtain for the CNN models, we computed the metrics on all the 10 folds for only the smaller projects in our study, i.e., AspectJ and Tomcat. For the other projects, we computed the metrics on only the first fold. As for SimpleLogistic models, we could compute the metrics for all the 10 folds and average the values to get the final score.

Dataset	CNN Model			SimpleLogistic Model	
	Memory (GB)	GPUs	Time (days)	Memory (GB)	Time (days)
AspectJ	40	2	21	250	17
Tomcat	80	3	70	550	*
SWT	145	0	150	3000	*
Eclipse	170	0	150	3000	*
JDT	*	*	*	*	*

Table 4.7: Memory And Training Time - 'All Files'

Dataset	CNN Model			SimpleLogistic Model	
	Memory (GB)	GPUs	Time (days)	Memory (GB)	Time (days)
AspectJ	10	1	7	50	4
Tomcat	80	3	24	80	13
SWT	120	3	110	2500	*
Eclipse	120	3	110	2500	*
JDT	*	*	*	*	*

Table 4.8: Memory And Training Time - 'Buggy Files'

Dataset	CNN Model			SimpleLogistic Model	
	Memory (GB)	GPUs	Time (days)	Memory (GB)	Time (days)
AspectJ	10	1	3	50	2
Tomcat	100	3	6	300	4
SWT	120	3	60	800	*
Eclipse	120	3	50	2500	*
JDT	120	3	80	2500	*

Table 4.9: Memory And Training Time - 'Very Buggy Files'

* indicates that the computations could not be carried out, due to lack of availability of resources on Compute Canada clusters

Dataset	CNN Model				SimpleLogistic Model			
	AUC	MAP	MRR	Top-5	AUC	MAP	MRR	Top-5
AspectJ	0.9	0.25	0.27	44%	0.9	0.3	0.33	52%
Tomcat	0.84	0.16	0.17	23%	*	*	*	*
SWT	0.86	0.24	0.26	38%	*	*	*	*
Eclipse	0.6	0.04	0.04	5%	*	*	*	*
JDT	*	*	*	*	*	*	*	*

Table 4.10: Results - 'All Files'

Dataset	CNN Model				SimpleLogistic Model			
	AUC	MAP	MRR	Top-5	AUC	MAP	MRR	Top-5
AspectJ	0.86	0.34	0.38	61%	0.84	0.43	0.47	63%
Tomcat	0.84	0.29	0.32	40%	0.77	0.05	0.05	5%
SWT	0.87	0.3	0.3	51%	*	*	*	*
Eclipse	0.72	0.13	0.14	18%	*	*	*	*
JDT	*	*	*	*	*	*	*	*

Table 4.11: Results - 'Buggy Files'

Dataset	CNN Model				SimpleLogistic Model			
	AUC	MAP	MRR	Top-5	AUC	MAP	MRR	Top-5
AspectJ	0.83	0.44	0.47	64%	0.79	0.16	0.17	24%
Tomcat	0.81	0.35	0.37	52%	0.66	0.25	0.26	36%
SWT	0.82	0.3	0.3	51%	*	*	*	*
Eclipse	0.75	0.16	0.17	23%	*	*	*	*
JDT	0.67	0.11	0.12	17%	*	*	*	*

Table 4.12: Results - 'Very Buggy Files'

* indicates that the computations could not be carried out, due to lack of availability of resources on Compute Canada clusters

4.6 Discussion

4.6.1 Minimizing The Lexical Gap Between Bug Reports And Source Files

As discussed in Chapter 2 of this study, many state-of-the-art models have been proposed in the past to automatically localize bugs. These approaches can be classified

into dynamic and static approaches. The dynamic approaches use the semantics of the program, test case execution traces, whereas the static approaches use pre-defined bug patterns and IR or ML to localize bugs. The IR or ML approaches are the most popular ones used in bug localization research. These models use IR- or ML-based techniques like TF-IDF, LSA, LDA, VSM, rVSM, Naive Bayes, etc. The key drawback of all these approaches is that they are unable to minimize the lexical gap [28, 85, 130] between the natural language corpus in the bug reports and the technical corpus in the source code. Some attempts have been made in the past to overcome this drawback. Ye et al.[130] used additional corpus from the documentation of the APIs used in source files. This approach did not help as the documentation of APIs does not have the buggy information specific to a project, rather they contain information regarding more general tasks. Kim et al. [60] used the names of the previously fixed source files as classification labels on the bug reports instead of the actual source file content. The problem with this approach is that for a new bug report, the model cannot suggest files that have not been buggy or faulty before.

As none of these state-of-the-art models were successful in this task, in recent years some researchers have proposed deep learning-based models which aim to eliminate or reduce the lexical gap between bug reports and source files. Lam et al. [67, 68], Huo et al. [54, 126] proposed deep learning models based on RBM, CNN, LSTM to localize bugs. Their work proved that unlike the other state-of-the-art traditional models, the deep learning models are able to minimize the lexical gap between bug reports and source files. They concluded that DNNs are able to do this, by learning to link high-level, abstract concepts between bugs reports and source code files.

As mentioned in the previous chapters, a simple CNN with little hyper-parameter tuning has been shown to perform significantly well for classifying text [132]. As discussed in Chapter 3, the two key concepts of CNNs are Local Invariance and Compositionality. These key features enable CNNs to able to recognize images with high accuracy [65]. They also enable CNNs to perform well on certain NLP-related tasks like sentiment analysis [132], identifying Parts Of Speech [39], text classification [134], etc. CNNs construct higher-order features (n-grams) from lower-order features and preserve the local information about locality. CNNs lose the global information about locality due to the Local Invariance feature. Nevertheless, this does not affect the performance of the model when performing sentiment analysis or text classification, as in these cases

the ordering of words is not of importance at the document level. For instance, in case of Movie Review sentiment analysis, it is important that the model is able to learn the local order of the words like - "not bad, quite good" which are crucial for identifying the sentiment of the sentence, and not the exact occurrence of these words in the sentence or document. This means that, CNNs are good at identifying the polarity of a sentence but cannot encode long-range dependencies in a sentence. This is the reason why LSTMs are more preferred for tasks like language modelling.

As discussed earlier, bug localization is a Learning-To-Rank IR problem which can be solved using the Pairwise approach, where each sentence is a combination of bug reports and source files and the task is to identify the positive (linked) and negative (non-linked) records. Hence a CNN model, which has been widely used in the past for learning the polarity of a sentence [132] could be a potential model that can correctly classify the corpus related to bug report and source code files, by learning to relate the natural language or domain-related terms/tokens in bug reports and different code tokens in source files.

Conclusion

From the above discussion, we can conclude that a deep learning-based model like CNN could be a potential approach that could eliminate or minimize the lexical gap between bug reports and source code files.

4.6.2 CNN vs SimpleLogistic models

We now compare a non-linear DNN (CNN model) and a traditional ML (SimpleLogistic model) in terms of performance, training time and memory. Tables 4.10, 4.11, and 4.12 compare the two models for using 'All Files', 'Buggy Files', and 'Very Buggy Files', respectively, for all the datasets.

- **Performance**: In the case of AspectJ, the SimpleLogistic model outperforms the CNN model in terms of MAP, MRR, and Top-5 Rank, when considering all the source files in the dataset. The same trend continues even when we train the model on only the buggy source files in the dataset. This trend is reversed when we further reduce the buggy source files. In this case, the CNN model performs significantly

better than the SimpleLogistic model. In the case of Tomcat, the CNN models outperform the SimpleLogistic models in all the three variations of source files, in terms of MAP, MRR and Top-5 Rank.

- **Training Time**: Tables 4.7, 4.8, and 4.9 show that CNN training takes between 3 and 21 days for the smallest project (AspectJ), and between 80 and 110 days for the largest project (JDT). The training time for each dataset, is influenced by the number of GPUs used. The greater is the number of GPUs used, the lesser will be the time required to train the model. For instance, in case of AspectJ for 'All Files' variation, the training time with 1 GPU is 42 days. But as we used 2 GPUs, the training was reduced to 21 days. If we compare the training times across projects, per GPU, then CNN takes between 3 and 42 days for the smallest project and between 150 and 330 days for larger projects like Eclipse. The dataset size decreases as we reduced the buggy files in the project, which is the reason why training times decrease as we move from 'All Files' to 'Very Buggy Files' variation for both CNN and SimpleLogistic models. The CNN models have very high training time compared to the SimpleLogistic models, but if we consider the SimpleLogistic models alone, their training time is still high. This shows that even the traditional linear models like SimpleLogistic take large time to train on bug localization datasets.

- **Memory**: To train, the CNN models need from 10GB to 150GB of memory, while the SimpleLogistic models require 50GB to 3000GB of memory.

Conclusion

From the above discussion, we can conclude that the computation resources in terms the memory or the number of GPUs, required to train the deep learning-based or the traditional ML-based bug localization models are huge. From a practical perspective, obtaining these computation resources is both expensive and challenging for the mainstream software practitioners, who might not have the budget to access such resources.

We can also infer from the above discussion, that both the models have their own merits and demerits. The CNN model outperforms the SimpleLogistic model in most of the cases, but they have their own drawback of high training time. SimpleLogistic

models though fast, need a lot of memory and their performance is not as good as CNN models in most of the cases.

4.6.3 Performance Across Projects

We now discuss the performance of the models across the projects (see Tables 4.10, 4.11, 4.12).

For all the three variations of source files, the AUC for the CNN models for all the datasets is between 0.6 - 0.9. AUC greater than 0.5 is deemed as a good classifier, and as most of the values are around 0.8, we can conclude that the CNN and SimpleLogistic models are fairly good at classifying the samples across any bug localization dataset.

As for the MAP metric, for the 'All Files' variation, the values across the projects range from 0.04 - 0.25. When considering only the buggy files in the dataset, the MAP values range from 0.13 - 0.34. Finally, for the 'Very Buggy Files' variation, the MAP scores across the projects are in the range 0.11 - 0.44. MAP greater than 0.3 means that 3 out of the first 10 predictions are correct or relevant to the search query. That means, higher the MAP, better is the performance of the IR model. The trend in performance of the MAP across all the projects, for all the three variations of the source files, shows that the CNN models perform poorly for larger projects like Eclipse and JDT. This could be attributed to the fact that, for the larger projects, the datasets are more imbalanced as compared to the smaller projects. This makes it even more difficult for the models to identify or learn patterns in the few sets of linked records are present in the dataset. Nevertheless, this may not always be the only factor which affects the performance of the models across the projects. For instance, although Tomcat is a smaller dataset compared to SWT, the CNN models gives a higher MAP score for SWT dataset for two variations. This could be due to the quality of the artifact corpus in the datasets. The datasets which contain bug reports that have more information related to the buggy source files like stack traces, exception messages (showing the buggy source file paths) can be more easily linked to their buggy source files. This is because of the lexical match between bug reports and source code corpus, making it easier for the model to learn their relationship.

The MRR metrics trend is similar to the MAP one. For each of the variations of the source files across all the projects, the MRR scores lie between 0.04 - 0.27, 0.14 - 0.38, and 0.12 - 0.47, respectively. MRR of 0.5 means that, for a search query, its first relevant

or correct item in a ranked list of 10 items is present in the 5th position. This shows that the higher the MRR, the better is the performance of the model. Similar to the previous case of MAP, the CNN models give a lower MRR scores for larger projects, i.e., for Eclipse and JDT. The same explanation as above can be given to this trend.

The Top-5 ranks trends across the projects for all the variations of source files are in the range of 5% - 44%, 18% - 61%, and 17% - 64%, respectively. Top-5 rank of 50% means that for 50% of the bug reports, at least one relevant file is in the 1 to 5 positions of the retrieved ranked list. This clearly shows that a well performing model will have a high Top-5 score. As with MAP, MRR scores, the Top-5 rank is higher for AspectJ and then SWT. Tomcat has lesser Top-5 score compared to these two datasets. The larger projects, Eclipse and JDT, have the lowest Top-5 scores.

Conclusion

We can conclude that the performance of the CNN models is dependent on the size of the dataset, as it performs significantly better on smaller projects compared to the larger projects for all the three variations of source files. This trend is uniform across all the metrics, i.e., MAP, MRR, and Top-5. The only exception to this trend is the SWT dataset. The SWT project, though significantly larger than the Tomcat dataset, has better or similar performance to Tomcat data. As discussed above, this could be due to the quality of the bug report corpus in a dataset. Bug reports which have stack traces and tokens/words related to their buggy source files are more easily traceable, than those which have pure natural language data with no information about their potential buggy files. Hence the dataset size along with the quality of the bug report corpus in the dataset are the major factors that affect the performance of the CNN model for a particular dataset.

4.6.4 Effect Of Varying Buggy Files

We now examine the effect of varying the buggy source files in the datasets. When reducing the number of source files, the performance of the CNN models improve. This can observed from Tables 4.10, 4.11, and 4.12. For all the projects, i.e., AspectJ, Tomcat, SWT, and Eclipse, the MAP scores increase from 0.25 to 0.44, 0.16 to 0.35, 0.26 to 0.3, 0.04 to 0.17, respectively. The same observation can also be made for other metrics, i.e.,

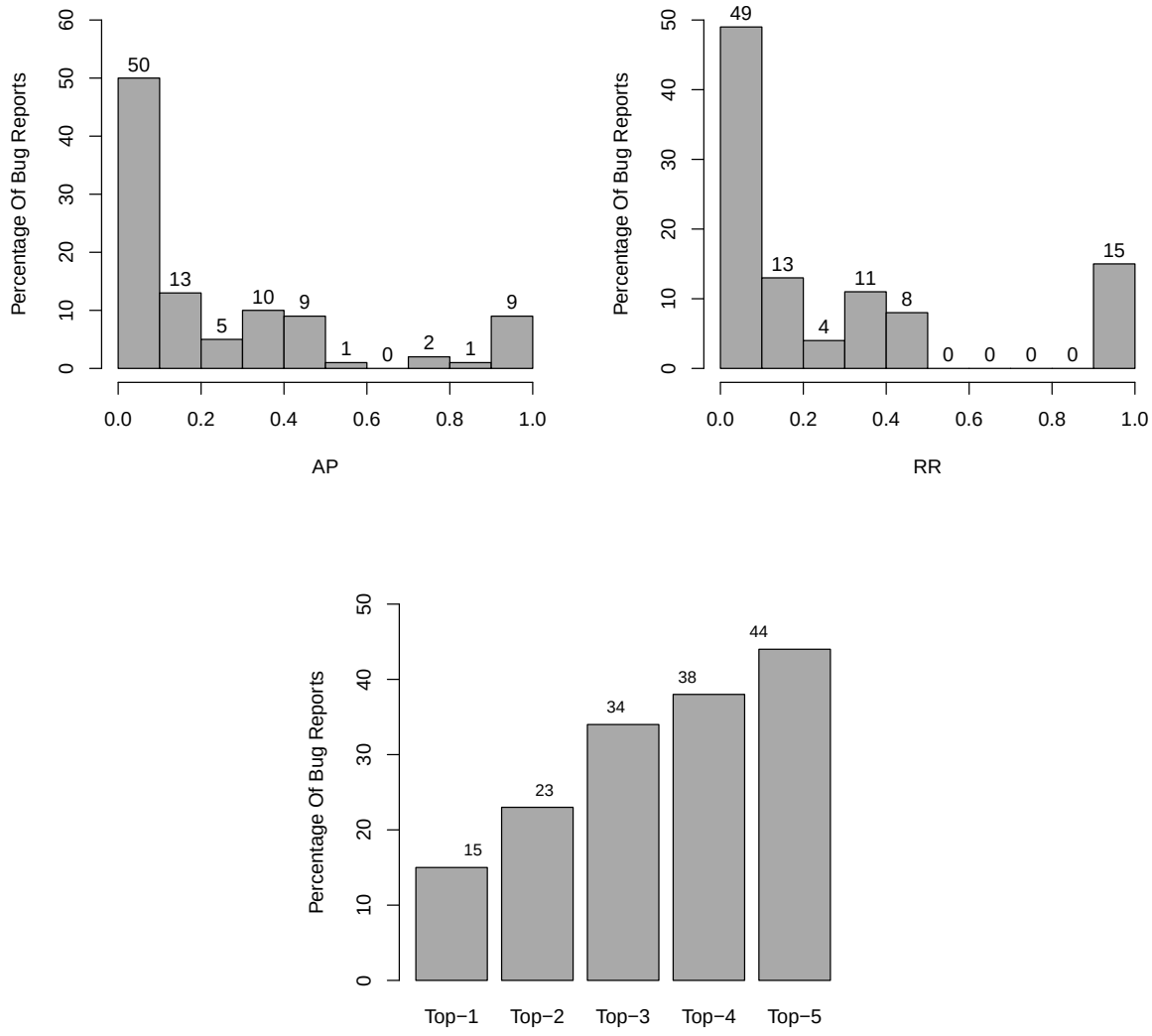


Figure 4.7: Plots For Metrics - CNN Model - AspectJ - 'All Files'

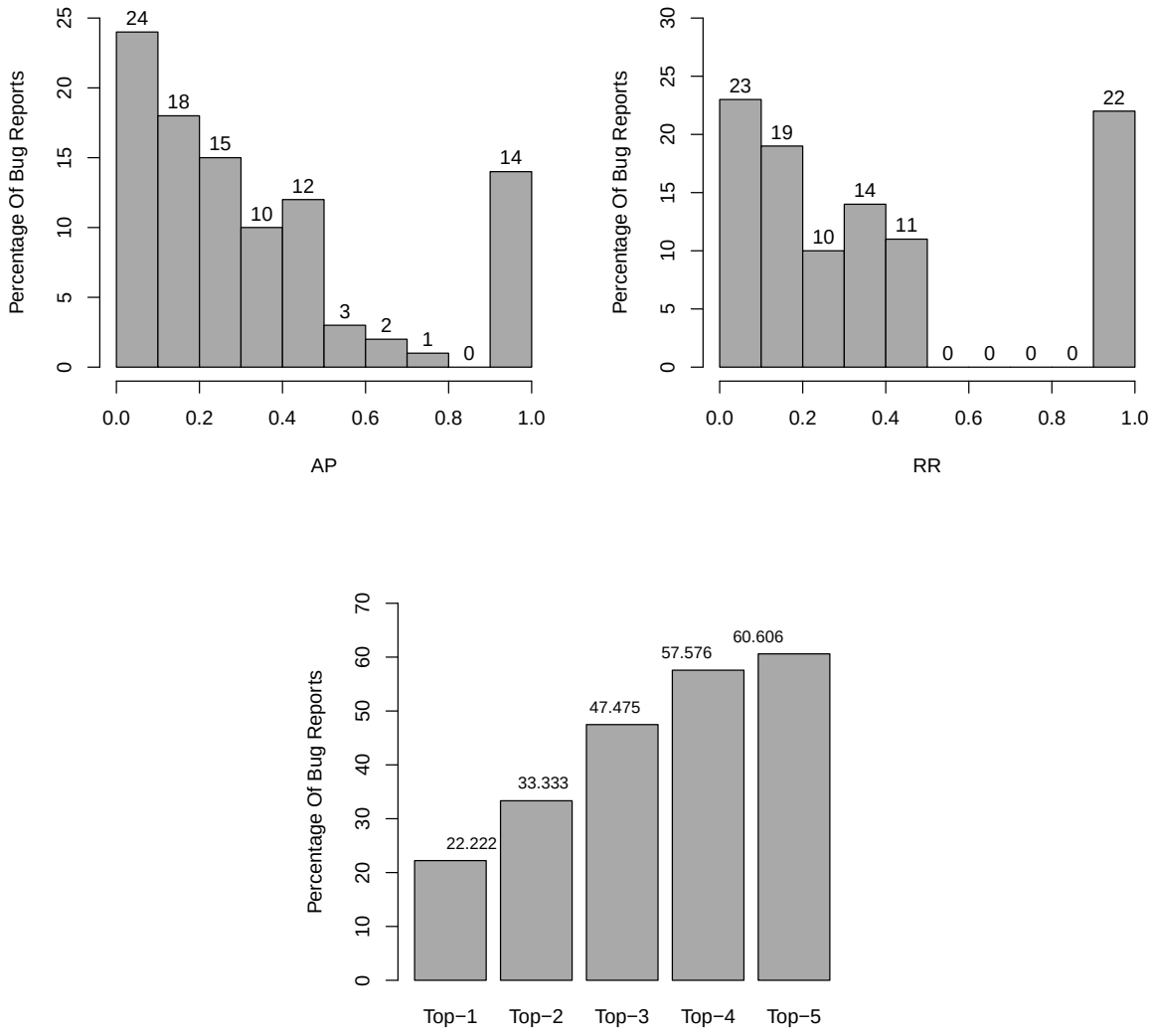


Figure 4.8: Plots For Metrics - CNN Model - AspectJ - 'Buggy Files'

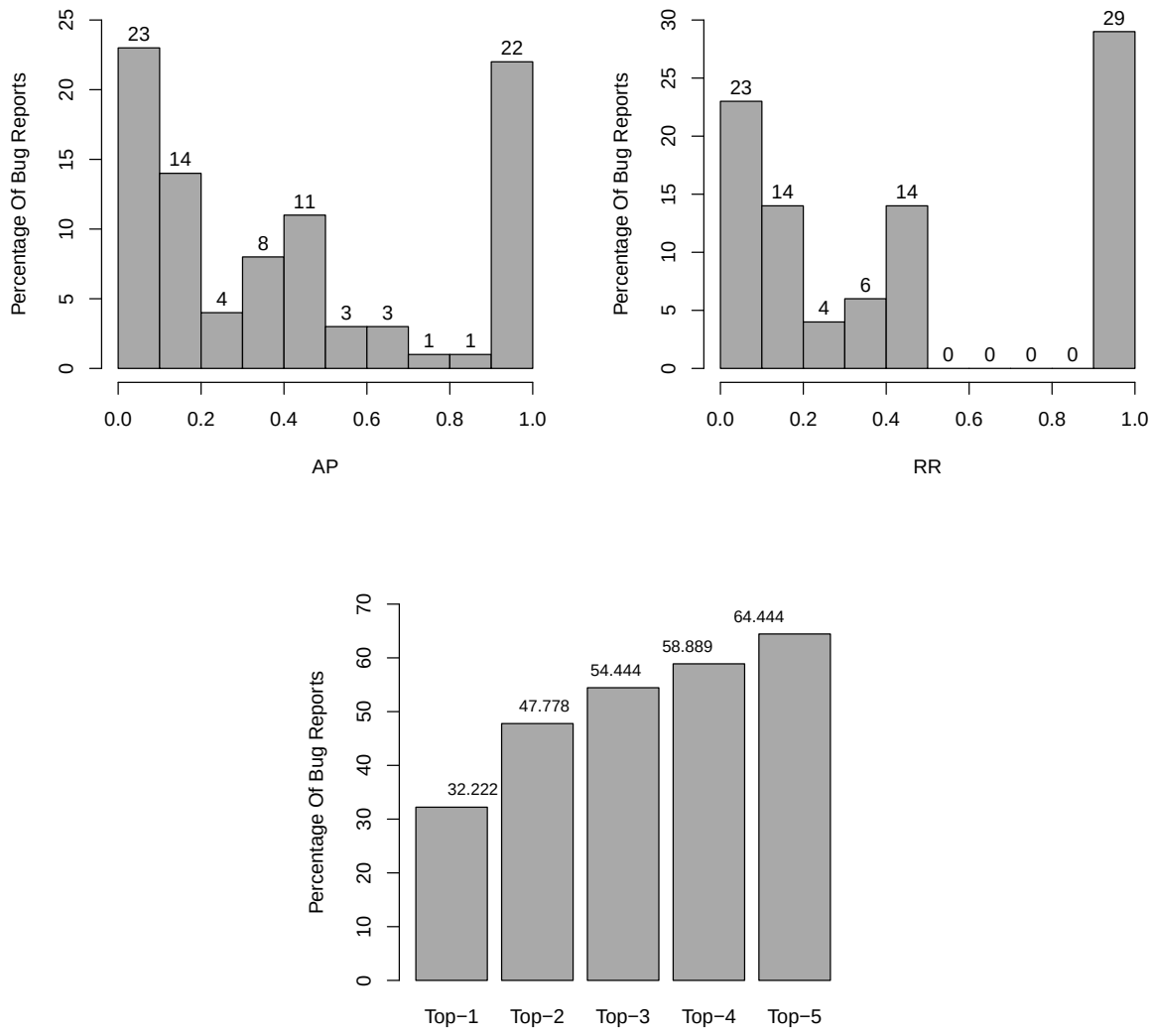


Figure 4.9: Plots For Metrics - CNN Model - AspectJ - 'Very Buggy Files'

MRR and Top-5 Rank. We conjecture that this improvement may be because reducing the number of files is reducing the search space, i.e., reserving more relevant files actually enables the model to easily identify the linked or the relevant files in the dataset.

Figures 4.7, 4.8, and 4.9 illustrate the range of AP, RR, and also the Top-1 to Top-5 ranks across the bug reports for AspectJ. From the bar plots, we can observe that only 9% of the bug reports have AP of 1 when considering all the source files. This score improves to 14% and then to 22% as we reduce the number of buggy source files. Different values of MAP (between 0.1 and 0.7) are considered across different IR systems based on the information needs of the end user [78]. In fields like traceability studies and bug localization, an IR model is considered to be effective if MAP values are greater than 0.5 [48, 126]. MAP equal to 1 means the model is a perfect IR model. We can also observe that only 13% of bug reports have MAP greater than 0.5. This score increases to 20% when considering only the buggy source files in the dataset. It further increases to 30% when using very buggy files to train the model. Similar observations can be made for the RR scores for AspectJ. An RR score of 1 means, the model is a perfect IR model. The percentage of bug reports with RR increases from 15% to 29% when reducing the buggy source files. As for the Top-k rank for AspectJ, for the 'All Files' variation, 16% of the bug reports have at least one relevant file in the first position of the ranked list of source files. This score increases to 23% and 33% as we reduced the number of buggy source files. The same trend can be observed even for Top-2, Top-3, Top-4, and Top-5 ranks. Similar plots have been generated for all the other datasets and are included in Appendix A.

Also, from Tables 4.10, 4.11, and 4.12, we can observe the effect of varying the number of buggy source files on the performance of the model for the other datasets. In case of Tomcat, the Top-5 rank increased from 23% to 40% and when further reducing the relevant buggy files, it went up to 52%. The same trend can also be noticed in case of MAP and MRR for the Tomcat dataset. When considering all the source files in the dataset, the MAP value is 0.16. This score improves to 0.29 for 'Buggy Files' and to 0.35 when further reducing the buggy source files. Similarly, MRR values increased from 0.17 to 0.32 and further to 0.37, as we reduced the buggy source files in the dataset.

Similar experimentation of varying the source files has also been performed on SWT, Eclipse, and JDT datasets. The trends in case of SWT are quite similar to the above trends. The Top-5 rank increases from 38% to 51% when considering only the buggy

files. Similarily, in case of MAP and MRR the scores improve by 0.06 as we reduce the files. But unlike the other datasets, for SWT, the performance is almost similar when further reducing the buggy source files. This means considering 'Very Buggy Files' does not have any affect on the performance of the model, as the MAP, MRR and Top-5 values do not change. Eclipse on the other hand follows the same trends as seen in AspectJ and Tomcat datasets. Though the improvement in performance in terms of MAP, MRR, Top-5 is not very large, there is still some difference in performance when varying the number of buggy source files in the dataset. As for JDT, due to the lack of GPU resources with RAM of more than 500GB, we could not run the models for the 'All Files' and 'Buggy Files' variations.

As for the SimpleLogistic models, the effect of varying source files is not very clear, as in some cases it tends to improve the performance of the model, whereas in some cases it does not.

Conclusion

From the above discussion, we can conclude that considering 'Very Buggy' files improves the performance of the CNN models. An important point to note from the above observation is that, most of the recent deep learning-based models [54, 67, 68, 126] consider only the 'Very Buggy Files' in order to improve the performance of the DNN models. This beats the practical relevance of these models, as the end users would use these models to localize the buggy files from the entire set of source files and not from only the 'Very Buggy Files' present in the project.

4.6.5 Practical Relevance

We now examine the practical relevance of the CNN model and the other state-of-the-art models in bug localization research. In this study we examine the effectiveness of the below models:

- *BugLocator* (BL) [137]: The BugLocator ranks the source files based on the textual similarity between the bug report and the source code using rVSM. It also takes into consideration information about similar bugs that have been fixed before. Zhou et

al. [137] performed experiments on AspectJ, Eclipse, and SWT datasets used in this study.

- *Two-Phase* (NB) [60]: Kim et al. [60] proposed a Two-Phase prediction model for localizing bugs using Naive Bayes approach. Lam et al. [67] and Huo et al. [54] used these models on the same datasets used in this study to compare the performance of their models with the Two-Phase model.

- *Learning-To-Rank* (LR) [130]: Ye et al. [130] introduced an adaptive ranking approach to leverage the domain knowledge of source code files into methods, API descriptions of library components used in the code, the bug-fixing history, and the code change history. When a user queries the model with a bug report, it computes the ranking score of each source file as a weighted combination of an array of features encoding the domain knowledge. The weights are trained automatically on previously solved bug reports using the Learning-To-Rank technique. Lam et al. [67] used this model on the open source bug localization datasets (that we used in this study) to compare the performance of their models with the LR model.

- *BLUiR* (Bug Localization Using Information Retrieval) [96]: Saha et al. [96] utilized the structured code information from bug reports and source files to enable more accurate bug localization. The user needs to first input the source files in which he/she would like to localize the bugs. BLUiR, builds the abstract syntax tree (AST) of each source code file using Eclipse, JDT and traverses the AST to extract different program constructs. Then, it tokenizes all the identifier names and comments in source files. The same is done with the bug reports. Later TF-IDF is applied between the source code and bug report tokens.

- *AmaLgam* [118]: Wang et al. [118] proposed a model which combines version history, similar bug reports and structure information for locating the buggy source code. Their aim to compose various variants of VSM with different weighting schemes, is achieved by a genetic algorithm (GA) based approach which explores the space of possible compositions and outputs a heuristically near-optimal composite model.

- *HyLoc* [67]: Lam et al. [67] used an RBM-based DNN in combination with rVSM

Dataset	Model	Top-5	MRR	MAP
AspectJ	HyLoc	71.2%	0.52	0.32
	LR	45.5%	0.33	0.25
	BL	47.7%	0.32	0.22
	NB	16%	0.1	0.07
Tomcat	HyLoc	72.9%	0.6	0.52
	LR	66.5%	0.55	0.49
	BL	61.8%	0.48	0.43
	NB	9%	0.08	0.07
SWT	HyLoc	69.0%	0.45	0.37
	LR	58.2%	0.41	0.36
	BL	38.3%	0.28	0.25
	NB	19%	0.14	0.11
Eclipse	HyLoc	70.5%	0.51	0.41
	LR	60.1%	0.47	0.40
	BL	49.3%	0.37	0.31
	NB	10.6%	0.07	0.06
JDT	HyLoc	65%	0.45	0.34
	LR	55.2%	0.42	0.34
	BL	40.2%	0.30	0.23
	NB	15%	0.11	0.08

Table 4.13: Comparison Of HyLoc With Other Models [67]

Dataset	Model	Top-10	MAP
AspectJ	BL	62.2%	0.41
	BLUiR	65.9%	0.43
	AmaLgam	69.3%	0.42
	CNN	77.3%	0.51
	NP-CNN	83.6%	0.54
	LSTM	79.2%	0.51
	LSTM+	85.5%	0.54
	LS-CNN	86.9%	0.56
Eclipse	BL	72.7%	0.42
	BLUiR	75.4%	0.44
	AmaLgam	77.3%	0.44
	CNN	82.1%	0.49
	NP-CNN	87.2%	0.54
	LSTM	86.9%	0.52
	LSTM+	87.2%	0.54
	LS-CNN	89.5%	0.56
JDT	BL	70.3%	0.44
	BLUiR	74.5%	0.43
	AmaLgam	75.7%	0.44
	CNN	85.9%	0.51
	NP-CNN	88.2%	0.53
	LSTM	86.8%	0.52
	LSTM+	88.3%	0.54
	LS-CNN	91.7%	0.58

Table 4.14: Comparison Of DNN-based Models With Other Models [126]

to localize bugs. rVSM collects the textual similarity features between bug reports and source files. The RBM-based DNN was used to learn to relate the terms in bug reports to potentially different code tokens and terms in source files. They experimented on the same datasets used in this study and also compared the performance of their model with all the above state-of-the-art models. Their experiments show that HyLoc outperforms the other models. Table 4.13 shows the comparison made by Lam et al [67] using the above state-of-the-art models.

- *NP-CNN* [54] and *LS-CNN* [126]: Huo et al. [54, 126] proposed variants of the CNN and LSTM deep learning models called NP-CNN and LS-CNN. They compared the performance of their models with some of the above state-of-the-art models on open source bug localization datasets. The CNN model used in our study has been previously used as a baseline model by Huo et al. [54], [126]. They proved that their NP-CNN and LS-CNN models outperform the CNN model and also the other state-of-the-art bug localization models. Table 4.14 shows the comparison made by Huo et al [126] against the above models.

From Tables 4.10, 4.14 we can observe that the experiments on the CNN models made by Huo et al. [126] on AspectJ, Eclipse and JDT datasets yielded results which are different from those obtained in our study. This can be attributed to the fact the Huo et al.[126] experimented on the reduced set of source files. They compared their NP-CNN and LS-CNN models with baseline models such as the CNN model proposed by Kim [132] and some of the above state-of-the-art models.

But from the practical relevance perspective, the same CNN model used by Huo et al. [54] does not perform strongly when trained on the entire set of source files from the project repository. This is an important observation because, when a software practitioner uses this model, he/she would expect the model to locate buggy source files from the entire source base. Hence the actual effectiveness of any bug localization model can only be determined when the model is trained on the entire set of source code files available in the project repository. This motivated us to experiment on different variations of the buggy source files in our study, to examine the effect of the size of search space on the effectiveness of CNN models.

We now compare the performance of our CNN model and also each of the above models with the expectations of the software industry. The criteria to be met by any

bug localization model in order to satisfy the software practitioner are given below [64]:

- **Granularity**: All the models work on file- or class-level granularity. But almost 52% of the practitioners prefer method level granularity and only 26% prefer file level granularity. So the above state-of-the-art models are able to satisfy only about 26% of the practitioners in terms of granularity.

- **Success Criteria**: This is defined by the Top-k Rank metric. Almost 75% of the developers gave 100% Top-5 rank as the minimum success criteria for any model. Another 25% are satisfied with a 100% Top-10 Rank. From Table 4.13 we can observe that HyLoc[67] outperforms all the other models like LR [130], BL [137] and NB [60] in terms of Top-5 Rank. But it gives a Top-5 Rank in the range 65% - 73% across different projects. The DNN models trained by Hou et al. [126] in Table 4.14, give a Top-10 Rank in the range 77% - 92% . But these values cannot be used to examine the practical relevance of the DNN models, as they are not trained on the entire set of source files in the datasets. This can supported by the fact that the same CNN architecture used in our study gives Top-5 Rank ranging from 44% - 64% for AspectJ (see Tables 4.10, 4.11, 4.12). For the other projects, like Tomcat, SWT, Eclipse and JDT the Top-5 Rank is 23% - 52%, 38% - 51%, 5% - 23%, 17% respectively. This shows the CNN models can suggest at least one buggy or relevant file in Top-5 positions for less than 50% of bugs. This shows that these models do not meet the the minimum success criteria of the practitioners.

- **Trustworthiness**: If a model meets the above success criteria atleast 50% of the time, then it will achieve 50% of the developer's satisfaction rate. But given that none of the state-of-the-art models or the DNN models meet the success criteria, i.e., the Top-5 Rank of 75% developers, the trustworthiness of these models is low.

- **Scalability**: About 50%, 75%, 90% of satisfaction rates can be achieved if a model is scalable enough to localize bugs in projects containing 10,000 LOC, 100,000 LOC, and 1,000,000 LOC. Ye et al. [130] and Zhou et al. [136] built models that could satisfy about 75% of the practitioners, i.e., they could localize bugs from programs of size greater than 100,000 LOC. Also, Huo et al. [54], Lam et al. [67] experimented on open source projects with programs of size above 100,000 LOC. In our study we have experimented on the same open source datasets each having programs sizes

of 160,000 LOC for AspectJ, 409,021 LOC for Tomcat, 1,057,102 LOC for Eclipse and 745,326 LOC for SWT, 1,523,382 LOC for JDT. This confirms that the models in our study are all scalable and satisfy about 75% (in case of AspectJ, Tomcat, SWT) and 95% (in case of Eclipse, JDT) of the practitioners. The same can also be said for the prior models [54, 67] which used the same datasets.

- **Efficiency**: To achieve a satisfaction rate of at least 50%, the model should localize bugs in less than a minute. About 90% of the practitioners are satisfied with this efficiency threshold. Most researchers when proposing a new bug localization approach, do not state the execution time of their models. Nevertheless, a recent research by Lam et. al [67, 68] state that their HyLoc models takes a maximum of 1.5, 4.1, 3.8, 4.8, 2.4 minutes to predict the relevant buggy source files for one bug report in Tomcat, AspectJ, Eclipse, JDT, SWT respectively. So for the smallest project, i.e., AspectJ with about 593 bug reports, Hyloc takes about 2431 minutes to localize buggy files. As for the CNN models trained in our study, the time taken to predict relevant source files for all the bug reports for AspectJ, Tomcat, SWT and Eclipse and JDT projects is 8, 28, 226, 494, 900 minutes respectively. This shows that the CNN model is more efficient than the HyLoc model in terms of execution time. Nevertheless, neither of these models meet the efficiency criteria of the software practitioner. Potentially, this can be rectified by adding more compute power.

Conclusion

The three important points worth noting from the above discussion are:

- Researchers do not use the entire set of source files from the repository when experimenting on a particular bug localization approach. This will prevent them from evaluating their models from a practical relevance perspective. This is the major setback in the evaluation process of the bug localization models.

- Apart from using IR metrics, to measure the performance of the models, researchers should verify if their models meet the criteria stated by software practitioners in [64], as they are the end-users of these models.

- From the above discussion, it is evident that the software developers should be cautious while using the current bug localization models and should not depend on them blindly to localize bugs, at least till more effective approaches evolve.

4.7 Threats To Validity

In this section we discuss the threats to validity, classified as per [124, 131].

4.7.1 Internal Validity

Interval validity relates to the experimental errors and biases in the study. We mitigate the bias by reusing the bug reports dataset that has been used in prior studies [54, 67, 68, 126, 130, 137]. To reduce the experimental errors, we have carefully checked our implementation to the best of our abilities.

4.7.2 Construct Validity

Construct validity relates to the applicability of the set of evaluation metrics used in this study. The metrics like MAP, MRR and Top-k Rank are well-known information retrieval metrics and have been used before to evaluate many past bug localization approaches [92, 96, 104, 136]. Thus, we believe there is little threat to construct validity.

4.7.3 Conclusion Validity

We made sure the model is generalizable by avoiding overfitting. As the model used in our study is a non-linear model, over-fitting is a common problem. In order to prevent that, we have employed the dropout technique [108], which is simple yet effective. Dropout prevents co-adaptation of hidden units by dropping out values randomly. Moreover, we are performing 10-fold cross validation. Also, the risk of insufficient generalization is reduced by evaluating the models on five open source projects.

4.7.4 External Validity

Software engineering studies suffer from the variability of the real world, and the *generalization* problem cannot be solved completely [123]. Wieringa et al. [123] indicate that in order to build a theory, we need to generalize a theoretical population and have adequate knowledge of the architectural similarity relation that defines the theoretical population. Although we have used five open source projects in this study, our empirical evaluation may not be generalizable to other open source projects or industrial projects. The goal of the study was not to build a new model, but to experiment on the recently proposed deep learning-based bug localization models and examine their relevance in research and industry. The aim of the study was to also point out the pitfalls in the current research in this field and gain a better understanding of the various factors like dataset size and the artifact corpus on the performance of the model. The same empirical examination can also be applied to other software products with well-designed and controlled experiments.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this study we examine the effectiveness of a deep learning-based bug localization model and compare it with a traditional ML model. We also observe the effect of varying the buggy files on the performance of the model. We perform all our experiments on five open source bug localization datasets (AspectJ, Tomcat, Eclipse, SWT, and JDT).

We give a detailed explanation about the lexical gap that exists between bug reports and source code files and also discuss the various attempts made in the past bug localization studies, to reduce this lexical gap. We also explain why we have chosen to experiment on a CNN model and the motivation behind examining the effectiveness of a deep learning-based bug localization model. We also cover the fundamentals of deep learning and logistic models. Then, we explain the architecture of the CNN model used in our study and also the IR metrics used to evaluate the models. Next, we discuss the data extraction process followed to obtain the bug localization data for all the five open source projects. Then, we mention the data preprocessing steps followed to generate the bag-of-words corpus from bug reports and source code files. After this, we go into the details of how we generated the traceability matrix. Next, we give some descriptive statistics of the data and also analyze the linked records in the traceability matrix.

We apply the CNN and SimpleLogistic models on the traceability matrix for each of the five datasets, for all the three variations of buggy source files. We also describe the experimental setup and metric computations for both models. Next, we compare

the CNN model with the SimpleLogistic model in terms of performance, training time, and memory and discuss the merits and demerits of these models. Also, we compare the performance of the CNN model across all the datasets for all three variations of buggy source files: 'All Files', 'Buggy Files', and 'Very Buggy Files'. Then, we observe the trends in the performance of both the CNN and SimpleLogistic models across the three variations of source files.

Finally, we evaluate the CNN model against the expectations of the software practitioner and discuss the drawbacks of the current state-of-the-art bug localization models.

Our first research question (RQ1) was: *How can we minimize the lexical gap between natural language texts in bug reports and technical/domain corpus in source code files in order to automatically localize bugs?* To address this research question, we discuss the existing state-of-the-art traditional and deep learning models and how a deep learning model like CNN could be a potential solution that could minimize the lexical gap between bug reports and source files.

Our next research question (RQ2) was:*How effective are the CNN models in meeting the expectations of the software practitioner?* To address this question, we train a CNN model which has been widely used in the past for text classification and bug localization purposes. We then examine the CNN model and other modern bug localization models against the expectations set by the software practitioner. We found that none of these models meet the practitioner's criteria in terms of granularity, success rate, trustworthiness, and efficiency. Nevertheless, the deep learning-based models could still meet the scalability criteria to some extent.

Our next research question (RQ3) was: *How do the CNN models perform in comparison with the SimpleLogistic models on software bug localization data?* To address this, we calculate metrics for all the five datasets on the predictions made by the CNN and SimpleLogistic models. Then, we compare both models using IR metrics MAP, MRR, and Top-k rank. We found that the CNN models outperform the SimpleLogistic models in most cases. CNN models require higher training times compared to the SimpleLogistic models. Nevertheless, SimpleLogistic models still take more than 100 days to train, especially for large projects. SimpleLogistic models require a very large amount of memory (about 50 - 3000GB) in comparison with the CNN models (10 - 120GB).

Our next research question (RQ4) was: *How do the CNN models perform across different open source software bug localization datasets?* To address this, we train the CNN

model on all the five open source bug localization datasets and compare the performance of the model across the datasets. We found that the performance of the model is dependent on the size of the dataset, as it performs significantly better on smaller projects compared to the larger projects for all the three variations of source files. The model gives the best performance on the smallest dataset (AspectJ) and the worst on the largest dataset (JDT). This trend is uniform across all the metrics, i.e., MAP, MRR, and Top-5. The only exception to this trend is the SWT dataset, which is larger than the Tomcat dataset but yields better or similar performance to Tomcat.

Our next research question (RQ5) was: *How does varying the source files in the dataset, affect the performance of the CNN and SimpleLogistic Models?* To address this, we vary the source files in the dataset based on three variations: 'All Files', 'Buggy Files', and 'Very Buggy Files'. We observe the performance of the CNN model across these three variations for AspectJ, Tomcat, SWT, and Eclipse datasets. Our experiments show that reducing the number of buggy source files improves the performance of the model. This could be due to the fact that reducing the search space enables the model to easily identify the linked or the relevant files in the dataset.

We believe that our experiments highlight the drawbacks of the experimental setup for the modern deep learning models. Recent works consider only a subset of the buggy source files in the repository when training the models. This approach prevents us from knowing the effectiveness of these models from a practical standpoint. This is because, in a real-time scenario, a practitioner would want the model to localize a buggy source file from the entire source base and not just from a subset of files.

Most researchers consider only the IR metrics to evaluate the performance of their proposed approach or model to localize bugs. We believe that in addition to this, the models should be examined to verify if they meet the expectations of a software practitioner. Also, even though deep learning models perform well, compared to the traditional ML models on bug localization data in most cases, they have their own set of demerits such as high training time and large amount of hardware resources, such as GPUs and memory. We believe that our study is of interest to software practitioners, as it provides enough evidence to convince the practitioners that they should be cautious while using the current deep learning models and should not depend on them blindly to localize bugs.

5.2 Future Work

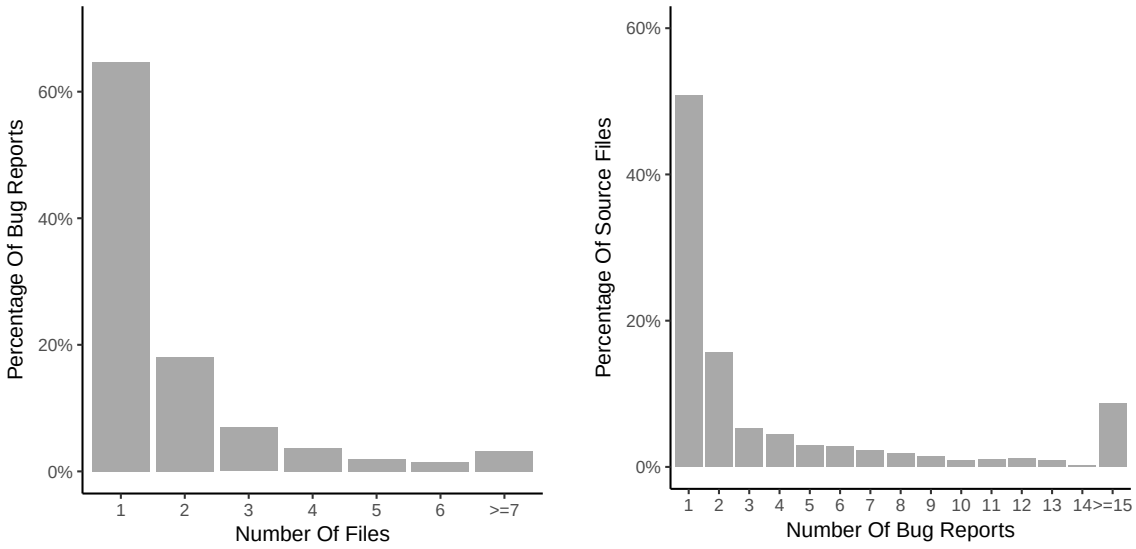
Going forward, we would like to experiment with different traditional ML models and DNN architectures which localize bugs at a more finer granularity level and which are more efficient. We would also like to ensure that we build a model that achieves at least 75% practitioner's satisfaction rate. Also, there are no studies which build scalable bug localization models on industrial datasets. Hence, we will explore options needed to tackle the scalability problem on commercial software.

Though there has been a large number of publications and research in the last 10 - 15 years on bug localization, it is still an evolving area of research. The critical challenge for a researcher in this field would be to build a model that finally meets the adoption thresholds set by the software industry.

Appendices

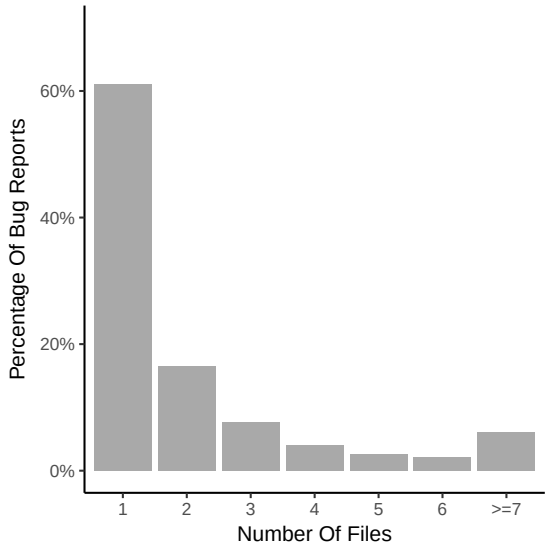
Appendix A

Additional Figures

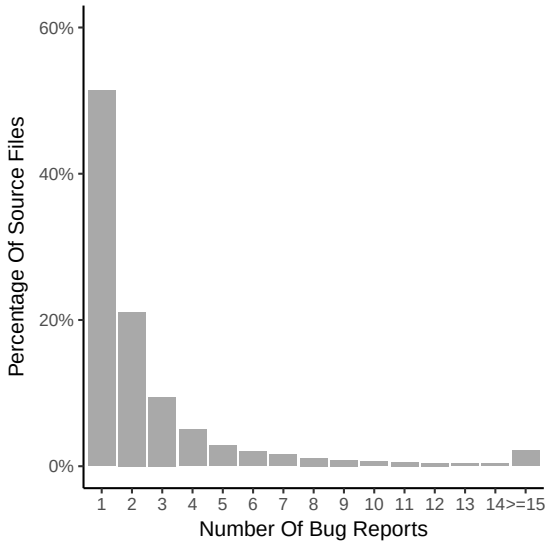


(a) Files Changed Across Bug Reports (b) Bug Reports Associated With Files

Figure A.1: Analysis Of Linked Records - SWT

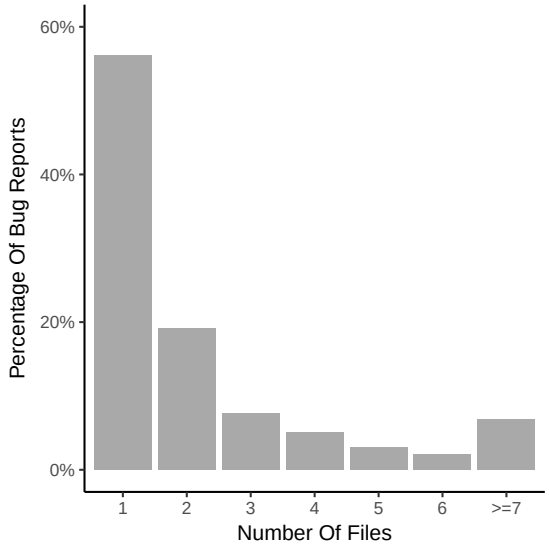


(a) Files Changed Across Bug Reports

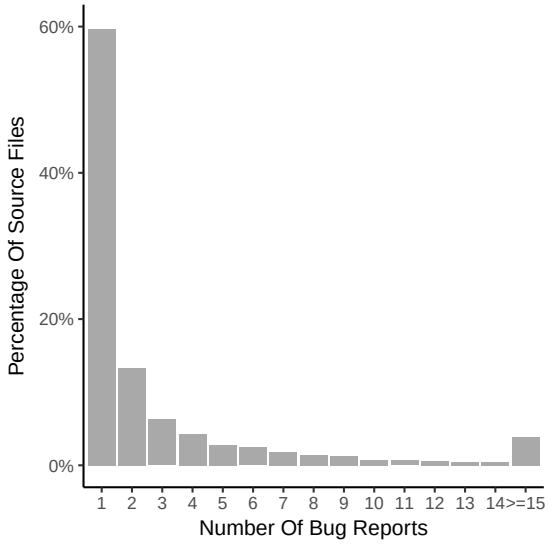


(b) Bug Reports Associated With Files

Figure A.2: Analysis Of Linked Records - Eclipse



(a) Files Changed Across Bug Reports



(b) Bug Reports Associated With Files

Figure A.3: Analysis Of Linked Records - JDT

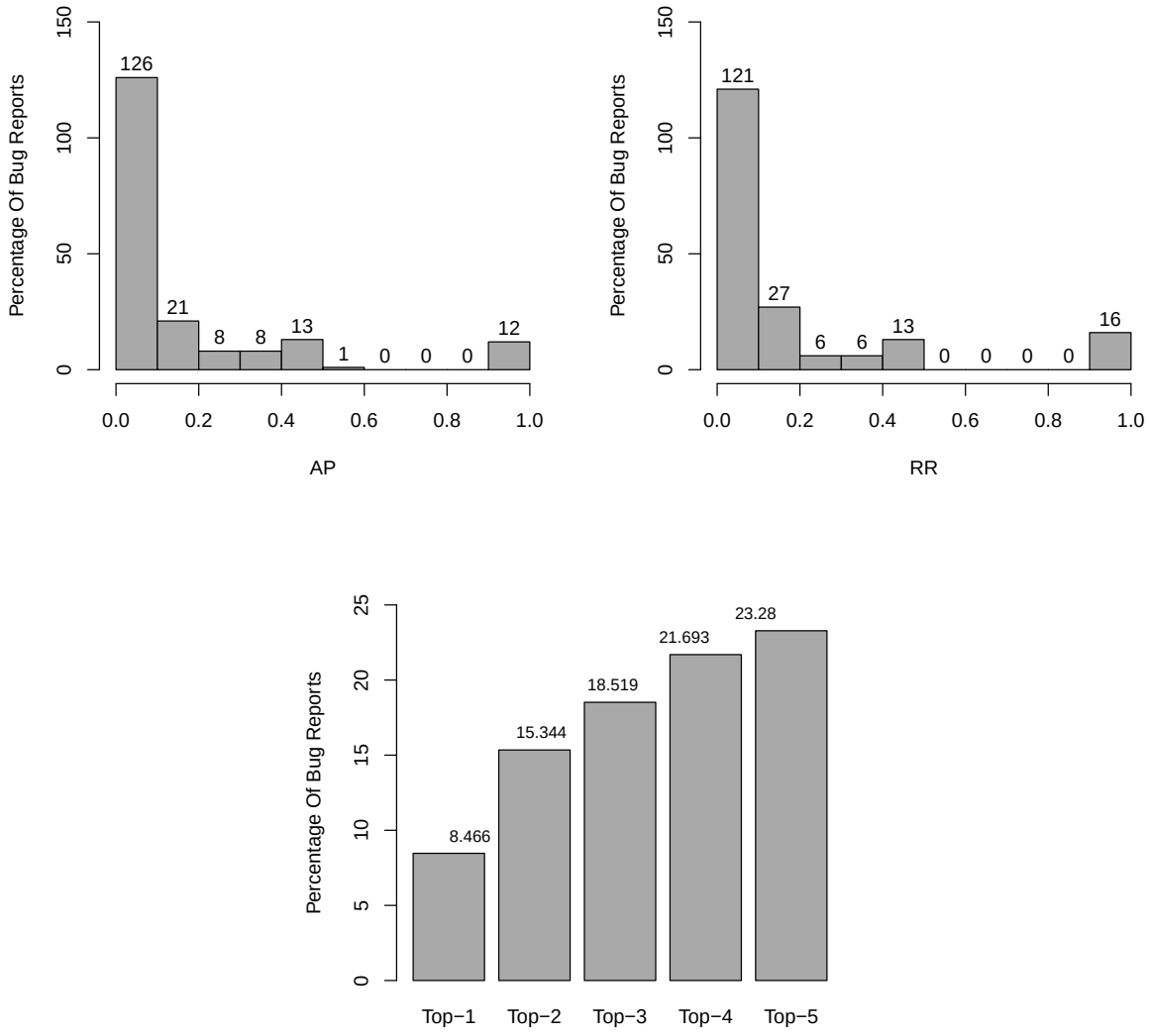


Figure A.4: Plots For Metrics - CNN Model - Tomcat - 'All Files'

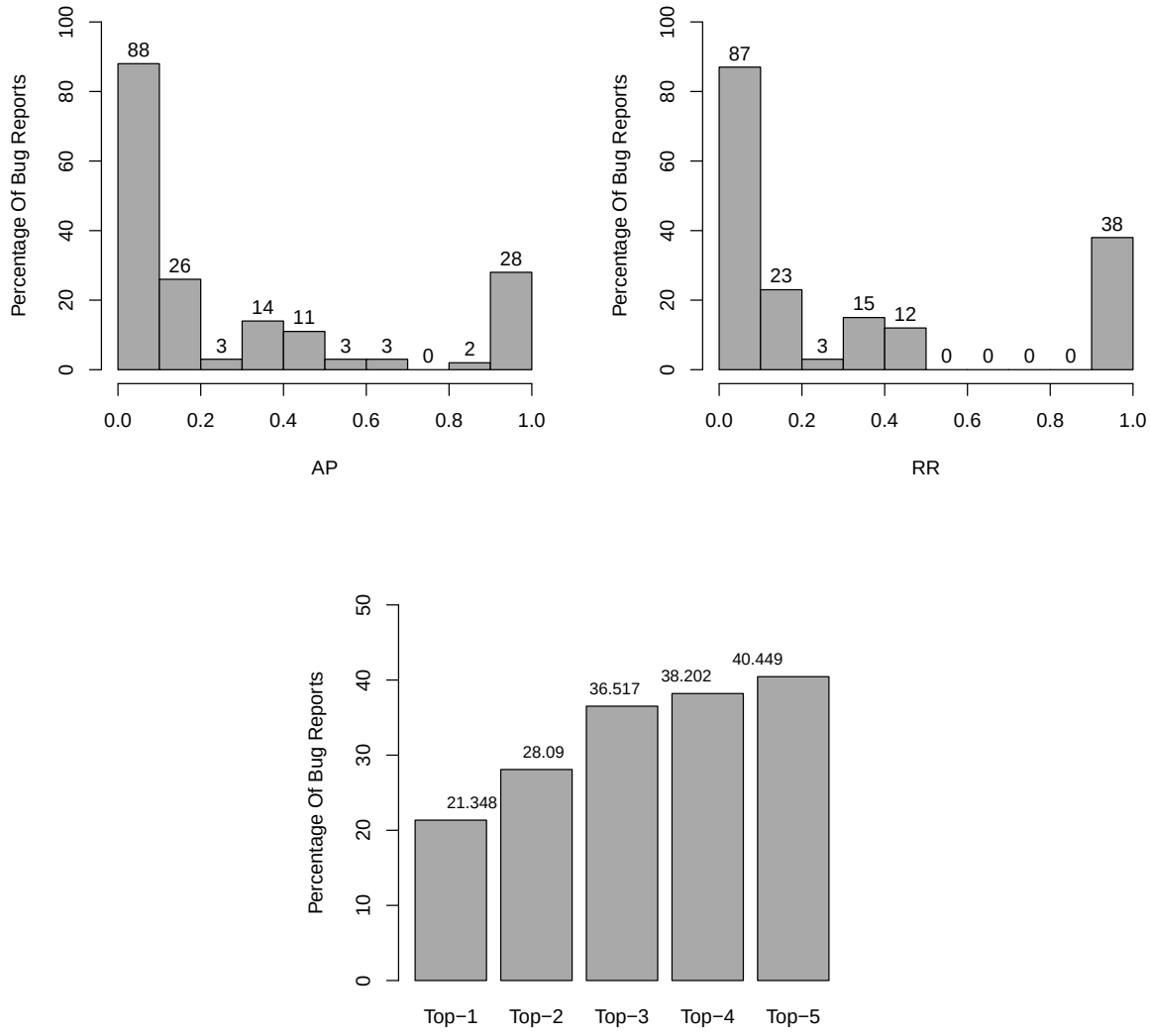


Figure A.5: Plots For Metrics - CNN Model - Tomcat - 'Buggy Files'

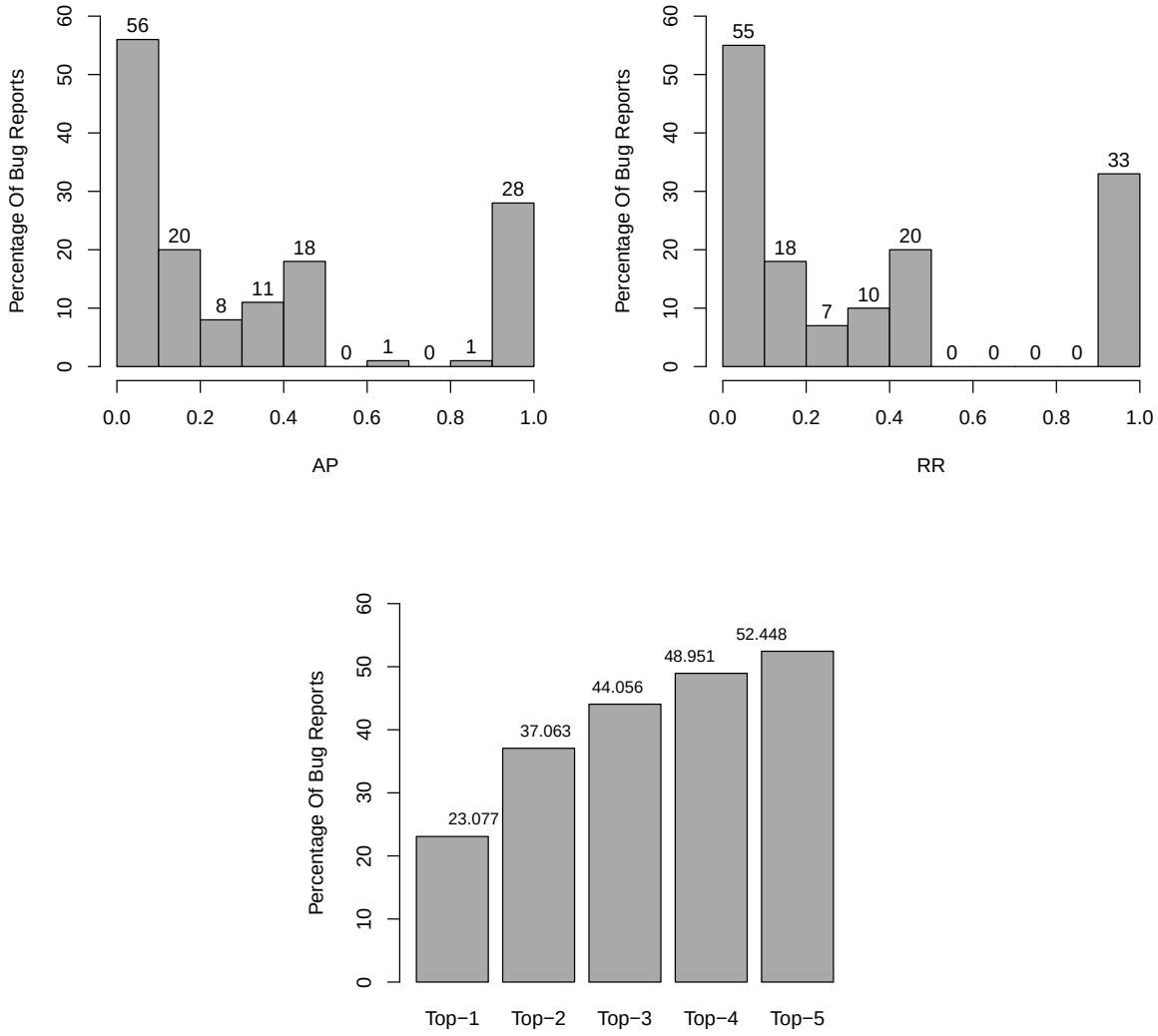


Figure A.6: Plots For Metrics - CNN Model - Tomcat - 'Very Buggy Files'

Appendix B

Data Extraction Scripts

Script to extract source files from git repo based on the bug-commit mapping. We extract the version of the source file before the bug fix (commit) was made for each bug report.

```
1
2  #Script: data.py
3  #Execution : python data.py <missing bug file> <new file>
4  # To filter out only bugs and commits from the other mappings.
5  import csv, sys, os
6  file = os.path.join(sys.path[0], sys.argv[1])
7  newfile = os.path.join(sys.path[0], sys.argv[2])
8  print os.path.join(sys.path[0], file)
9  print os.path.join(sys.path[0], newfile)
10 with open(file, "rb") as csvfile:
11     reader = csv.reader(csvfile, delimiter=",")
12     for row in reader:
13         data = ([row[1]],row[2].split("\r\n"))
14         with open(newfile, "a+") as newcsvfile:
15             writedata = csv.writer(newcsvfile)
16             writedata.writerows(data)
17
18 #Script: filter.sh
19 #Execution : ./filter.sh
20 # To get the head value of each commit.
```

```
while read line;
    do echo $line;
    git checkout $line~1;
    head=$(git rev-parse --short HEAD~1); echo "$head,$line" p0 >>
       head_commit.out; git checkout master; done <
       eclipse_commits_filtered.csv

#Script: diff.sh
#Execution : ./diff.sh
# To get the difference between the head and the commit
#!/bin/bash
while read line
do
echo $line
head=$(echo $line | awk -F\, '{print $1}')
commit=$(echo $line | awk -F\, '{print $2}')
files=()
files=$(git diff --name-status $head $commit | grep ".java$" | grep -E "^
   A|^M|^D")
#echo -ne ${files[@]} >> $commit.out
printf '%s\n' "${files[@]}" >> $commit.out
done < head_commit.out

#Script: extract.sh
#Execution : ./extract.sh
# To extract the files based on the diff information.
#!/bin/bash
while read line
do
    head=$(echo $line | awk -F\, '{print $1}')
    commit=$(echo $line | awk -F\, '{print $2}')
    while read newline
    do
```

```
54          type=$(echo "$newline" | cut -f 1)
55          file=$(echo "$newline" | cut -f 2)
56          if [[ $type == "A" ]]; then
57              git checkout $commit
58              if [ -f "$file" ] && [ -e "$file" ]; then
59                  folder=${file%/*}
60                  [ ! -d "$HOME/Sravya_Backup/Eclipse_Repo_NewMap/$folder" ]
61                      && \
62                      mkdir -p "$HOME/Sravya_Backup/Eclipse_Repo_NewMap/
63                         $folder"
64                  echo "$file exists in $commit"
65                  cp "$file" "$HOME/Sravya_Backup/Eclipse_Repo_NewMap/
66                     $folder"
67              else
68                  echo "$file doesn't exist in $commit"
69              fi
70              git fetch --all && git reset --hard origin/master && git
71                 checkout master
72
73          elif [[ $type == "D" ]] || [[ $type == "M" ]]; then
74              git checkout $head
75              if [ -f "$file" ] && [ -e "$file" ]; then
76                  folder=${file%/*}
77                  [ ! -d "$HOME/Sravya_Backup/Eclipse_Repo_NewMap/$folder" ]
78                      && \
79                      mkdir -p "$HOME/Sravya_Backup/Eclipse_Repo_NewMap/
80                         $folder"
81                  echo "$file exists in $commit"
82                  cp "$file" "$HOME/Sravya_Backup/Eclipse_Repo_NewMap/
83                     $folder"
84              else
85                  echo "$file doesn't exist in $commit"
86              fi
```

```
87              git fetch --all && git reset --hard origin/master && git
88                  checkout master
89          fi
90      done < $commit.out
91  done < head_commit.out
92
```

Appendix C

Source Code Parser Scripts

The below shell script is used to parse the java source code files to extract source code corpus. To Execute the below shell scripts, run the below commands on Linux shell:

```
for file in ${find . -name \*.java\*};
    do data=${../java-code.sh $file | ../code-to-words.sh
    -k ../java-keywords | awk '!x[$0]++' |
    paste -sd ' '}; echo -e $file '\t' $data;
    done > AspectJ_file_data.txt


# Script: code.words.sh
#!/bin/bash
set -e
basedir=${dirname "$0"}

keyword_opts=()
stopword_opts=()

while getopts k:s:h opt
do
        case $opt in
                k)
```

```
                        keyword_opts+=( "-f" "$OPTARG" )
                        ;;
                s)
                        stopword_opts+=( "-f" "$OPTARG" )
                        ;;
                h)
                        echo "$0 [[-k <keywords-file>]|[-s <stopwords-file
                           >]]* [<input-file>]*"
                        exit
                        ;;
        esac
done

# Select sed
if [ command -v gsed >/dev/null 2>&1 ]; then
  SED=gsed
else
  SED=sed
fi

# Discard punctuation & numeric literals.
function extract_identifiers() {
    $SED -e 's/0[xX][[:alnum:]]\+//g' -e 's/[^[:alpha:]_]\+/\n/g' | grep
       -v '^$'

}

# Split camel case into individual words, taking into account all-caps
# abbreviations, such as XML or JPEG, and split at underscores
function split_words() {
    $SED -e 's/\([[:lower:]]\)\([[:upper:]]\)/\1\n\2/g' \
         -e 's/\([[:upper:]]\+\)\([[:upper:]][[:lower:]]\)/\1\n\2/g' \
            -e 's/_\+/\n/g'
```

```
54 }
55
56 function ignore_keywords() {
57         grep -vw "${keyword_opts[@]}" -e "^$"
58 }
59
60 function lowercase() {
61         tr [:upper:] [:lower:]
62 }
63
64 extract_identifiers | split_words | lowercase | ignore_keywords
65
66
67 # Script: java-code.sh
68 #!/bin/bash
69 set -e
70
71 if [ $# -eq 0 ]
72 then
73     srcs="."
74 else
75     srcs="$@"
76 fi
77
78 function ignore_comments() {
79     cpp | grep -Evx '#.+'
80 }
81
82 # Ignore import and package statements.
83 function ignore_package_names() {
84     grep -Evx '[[:space:]]*(import|package)[[:space:]].+'
85 }
86
```

```
87 # -exec cat works with paths that contain spaces,
88 while pipe through xargs cat does not
89 find $srcs -name '*.java' -exec cat {} \; | ignore_comments |
90    ignore_package_names
91 #find "$srcs" -name '*.java' -exec cat {} \;
92
93 #java-keywords
94 list
95 serializable
96 observable
97 throwable
98 int
99 double
100 float
101 string
102 boolean
103 constructor
104 exception
105 null
106 throwable
107 declare
108 java
109 abstract
110 continue
111 for
112 new
113 switch
114 assert
115 default
116 goto
117 package
118 synchronized
119 do
```

```
120 if
121 private
122 this
123 break
124 implements
125 protected
126 throw
127 else
128 import
129 public
130 throws
131 case
132 enum
133 instanceof
134 return
135 transient
136 catch
137 extends
138 try
139 final
140 interface
141 static
142 void
143 class
144 finally
145 strictfp
146 volatile
147 const
148 native
149 super
150 while
151
```

Appendix D

Data Preprocessing Scripts

The below R script is used to preprocess the data and generate train, test data.

```
1
2  #rt-polarity.pos : Linked records (bug report text and source file text
3      combined)
4  #rt-polarity.neg : Non-Linked records(bug report text and source file
5      text combined)
6
7  library(tm)
8  library(hash)
9  library(SnowballC)
10 library(textstem)
11 library(data.table)
12
13 tomcat <-dbConnect(MySQL(), user='root', password='', dbname='tomcat',
14     host='localhost')
15 results<-dbSendQuery(tomcat,
16     "SELECT bug_id,summary,description,files
17     FROM bug_and_files
18     ORDER BY report_time DESC")
19 res<-dbFetch(results,n=-1)
20 Tomcat<-as.data.frame(res)
21 Tomcat["combined_text"]<-paste(Tomcat$summary,Tomcat$description,sep=" ")
```

```
## expanding the files columns to make sure there is one file per row
Tomcat$files<-gsub("\\.java","\\.java<delim>",Tomcat$files)
s <- strsplit(Tomcat$files, split = "<delim>")
Tomcat<-data.frame(bug_id= rep(Tomcat$bug_id, sapply(s, length)),
combined_text = rep(Tomcat$combined_text, sapply(s, length)),
    files = unlist(s))
Tomcat$files<-trimws(Tomcat$file,which="both")

# Linked Files
Tomcat_Linked_Files<-read.csv(
    file="/Volumes/CORSAIR/Results/Tomcat/Tomcat_Linked_file_data.txt",
    sep="\t",header = FALSE,
    col.names=c("file","file_text"))
Tomcat_Linked_Files$file<-gsub("\\./","",Tomcat_Linked_Files$file)
Tomcat_Linked_Files$file<-trimws(Tomcat_Linked_Files$file,
    which="both")

# Removing all files which do not have any content in them.
Tomcat_Linked_Files<-Tomcat_Linked_Files[!Tomcat_Linked_Files
$file_text=="",]

Tomcat_All_Files<-read.csv(
    file="/Volumes/CORSAIR/Results/Tomcat/Tomcat_file_data.txt",
    sep="\t",header = FALSE,
    col.names=c("file","file_text"))

Tomcat_All_Files$file<-gsub("\\./","",Tomcat_All_Files$file)
Tomcat_All_Files$file<-trimws(Tomcat_All_Files$file,which="both")
Tomcat_All_Files$file_text<-as.character(Tomcat_All_Files$file_text)
Tomcat_All_Files<-Tomcat_All_Files[!Tomcat_All_Files$file_text=="",]

# Separate Non-Linked Files
Tomcat_Non_Linked<-Tomcat_All_Files[! Tomcat_All_Files$file %in%
```

```
55 Tomcat_Linked_Files$file,]
56
57 # Combining Linked and Non-Linked files
58 # This is the complete set of files in AspectJ repo.
59 Tomcat_Files<-rbind(Tomcat_Linked_Files,Tomcat_Non_Linked)
60
61 ## remove all test case related files
62 Tomcat_Files<-Tomcat_Files[!grepl("tests\\/", Tomcat_Files$file),]
63 Tomcat_Files<-Tomcat_Files[!grepl("testdata\\/", Tomcat_Files$file),]
64
65 # Text Pre-Processing
66 # Normalizing the source code text
67 Tomcat_Files$file_text<-stem_strings(Tomcat_Files$file_text)
68 Tomcat_Files$file_text<- gsub('\\s+', ' ', Tomcat_Files$file_text)
69 Tomcat_Files$file_text<- gsub('^\\s+', '', Tomcat_Files$file_text)
70 Tomcat_Files$file_text<-trimws(Tomcat_Files$file_text,which="both")
71
72 # Normalizing the bug report text
73 Tomcat$combined_text<-gsub("[[:punct:]]"," ",Tomcat$combined_text)
74 Tomcat$combined_text<-gsub("[[:digit:]]"," ",Tomcat$combined_text)
75 Tomcat$combined_text<-gsub("([A-Z])", " \\1", Tomcat$combined_text)
76 Tomcat$combined_text<-
77     gsub("(?<=\\b\\w)\\s(?=\\w\\b)", "",Tomcat$combined_text,perl=T)
78 Tomcat$combined_text<-tolower(Tomcat$combined_text)
79 Tomcat$combined_text<-stem_strings(Tomcat$combined_text)
80 Tomcat$combined_text<- gsub('\\s+', ' ', Tomcat$combined_text)
81 Tomcat$combined_text<- gsub('^\\s+', '', Tomcat$combined_text)
82
83 # Bug Ids and Bug Reports
84 dfBRRandom<-unique(Tomcat[sample(nrow(Tomcat)),c("bug_id","combined_text
85     ")])
86
87 # Dump bug reports to file
```

```
write.table(dfBRRandom,
    file="/Volumes/CORSAIR/Results/Tomcat/Tomcat_dfBR_Long.txt",
    sep="\t", row.names = FALSE, col.names = FALSE)

# Shell script keeps only unique words in the bug reports.

dfBRRandom<-read.csv(
    file="/Volumes/CORSAIR/Results/Tomcat/Tomcat_dfBR_Short.txt",
    sep="\t",header = FALSE,col.names = c("bug_id","combined_text"))

# Files and the extracted textual content from File
dfFileRandom<-Tomcat_Files[sample(nrow(Tomcat_Files)),]

distinct_file_names<-data.frame(
    files=dfFileRandom$file,file_text=dfFileRandom$file_text)

distinct_bug_reports<-data.frame(unique(dfBRRandom))

count_of_distinct_bug_reports <- length(unique(
   distinct_bug_reports$bug_id))

count_of_distinct_file_names <- nrow(distinct_file_names)

linked_data_tbl <- hash(keys = paste(Tomcat$files,
    Tomcat$bug_id,sep="#"), values = 0)

print(paste("Total Number of bug reports:",count_of_distinct_bug_reports)
    )

print(paste("Total Number of source files:",count_of_distinct_file_names)
    )

write.table(data.frame(),file="/Volumes/CORSAIR/Results/Tomcat/
```

```
121     All_Files/rt-polarity.neg",col.names = FALSE,row.names = FALSE,quote
122         = FALSE)
123
124 write.table(data.frame(),file="/Volumes/CORSAIR/Results/Tomcat/
125     All_Files/rt-polarity.pos",col.names = FALSE,row.names = FALSE,quote
126         = FALSE)
127
128 write.table(data.frame(bug_id=as.integer(),sentence=as.character()),
129     file="/Volumes/CORSAIR/Results/Tomcat/All_Files/bug_id_sentence",
130     col.names = FALSE,row.names = FALSE,quote = FALSE)
131
132 cnt <- 1
133
134 for(i in 1:count_of_distinct_bug_reports){
135   cat(paste("Processing bug report:",i,"\n"))
136   bug_report_id<-distinct_bug_reports[i,1]
137   bug_report_text<-distinct_bug_reports[i,2]
138   start_id <- cnt
139   end_id <- cnt + nrow(distinct_file_names) - 1
140   dat <- data.frame(row_id = seq(start_id, end_id), bug_id =
141     bug_report_id, bug_report_text = bug_report_text, file =
142     distinct_file_names$files,
143     file_text=distinct_file_names$file_text,class_label = "neg")
144
145   #update class of linked records
146   linked_rows_ids <- which(has.key(paste(dat$file,
147     bug_report_id,sep = "#" ), linked_data_tbl))
148
149   dat$class_label<-as.character(dat$class_label)
150   dat$class_label[linked_rows_ids] = "pos"
151   dat$sentence<-paste(dat$bug_report_text,dat$file_text,sep=" ")
152   cnt <- end_id + 1
153
```

```
154   #save data to positive and negative files
155   pos_rows_to_append<-dat[dat$class_label=="pos","sentence"]
156   neg_rows_to_append <-dat[dat$class_label=="neg","sentence"]
157   tot_rows_to_append <-dat[,c("bug_id","sentence")]
158
159   pos_file_path <- paste("/Volumes/CORSAIR/Results/Tomcat/All_Files/",
160     "rt-polarity.pos", sep="")
161
162   neg_file_path <- paste("/Volumes/CORSAIR/Results/Tomcat/All_Files/",
163     "rt-polarity.neg", sep="")
164
165   tot_file_path <- paste("/Volumes/CORSAIR/Results/Tomcat/All_Files/",
166     "bug_id_sentence", sep="")
167
168   write.table(pos_rows_to_append,file=pos_file_path,
169     append=TRUE,row.names=FALSE,quote = FALSE,col.names =FALSE)
170
171   write.table(neg_rows_to_append,file=neg_file_path,
172     append=TRUE,row.names=FALSE,quote = FALSE,col.names =FALSE)
173
174   write.table(tot_rows_to_append,file=tot_file_path,
175     append=TRUE,row.names=FALSE,quote = FALSE,col.names =FALSE,sep="\t")
176
177 }
178
179 ### LMT Stratified 10 fold Sampling (9 fold into train and 1 fold into
180    test)
181
182 library(data.table)
183 library(caret)
184
185 pos.data<-read.csv(
186     file="/home/spoliset/parse_source_code
```

```
    /Eclipse/bag_of_words/All_Files/rt-polarity.pos",
    header = FALSE,col.names="sentence")
pos.data["true_label"]<-1
neg.data<-fread(
    file="/home/spoliset/parse_source_code
    /Eclipse/bag_of_words/All_Files/rt-polarity.neg",
    sep=",",header = FALSE,quote = "")
colnames(neg.data)<-"sentence"
neg.data<-as.data.frame(neg.data)
neg.data["true_label"]<-0
full.data<-rbind(pos.data,neg.data)
train.index <- createDataPartition(full.data$true_label, p = .9, list =
   FALSE)
train <- full.data[ train.index,]
test <- full.data[-train.index,]
nb.pos.train<-train[train$true_label==1,]
nb.neg.train<-train[!train$true_label==1,]
nb.pos.test<-test[test$true_label==1,]
nb.neg.test<-test[!test$true_label==1,]
write.table(data.frame(),
    file="/home/spoliset/LMT_bow/Eclipse/All_Files/rt-polarity.pos.train
    ", col.names = FALSE,row.names = FALSE,
    quote = FALSE)
write.table(data.frame(),
    file="/home/spoliset/LMT_bow/Eclipse/All_Files/rt-polarity.neg.train
    ", col.names = FALSE,row.names = FALSE,
    quote = FALSE)
write.table(data.frame(),
    file="/home/spoliset/LMT_bow/Eclipse
    /All_Files/rt-polarity.neg.test
    ", col.names = FALSE,row.names = FALSE,
    quote = FALSE)
write.table(data.frame(),
```

```
220     file="/home/spoliset/LMT_bow/Eclipse
221     /All_Files/rt-polarity.pos.test
222     ", col.names = FALSE,row.names = FALSE,
223     quote = FALSE)
224 write.table(nb.pos.train[,"sentence"],
225     file="/home/spoliset/LMT_bow/Eclipse/All_Files/rt-polarity.pos.train
226     ", append=TRUE,row.names=FALSE,
227     quote = FALSE,col.names =FALSE)
228 write.table(nb.neg.train[,"sentence"],
229     file="/home/spoliset/LMT_bow/Eclipse/All_Files/rt-polarity.neg.train
230     ", append=TRUE,row.names=FALSE,
231     quote = FALSE,col.names =FALSE)
232 write.table(nb.pos.test[,"sentence"],
233     file="/home/spoliset/LMT_bow/Eclipse/All_Files/rt-polarity.pos.test
234     ", append=TRUE,row.names=FALSE,
235     quote = FALSE,col.names =FALSE)
236 write.table(nb.neg.test[,"sentence"],
237     file="/home/spoliset/LMT_bow/Eclipse/All_Files/rt-polarity.neg.test
238     ", append=TRUE,row.names=FALSE, quote = FALSE,col.names =FALSE)
239
```

Appendix E

Scripts For CNN Model

The below scripts classify the combined records of bug reports and source files into linked and non-linked records using Convolution Neural Nets. *data_helpers.py* script is used for reading the data, building the vocabulary and padding the input sentences. *model.py* script is used to train and test the CNN model.

```
1
2  # Script: data_helpers.py
3  # Dependencies: Keras with Tensorflow backend
4
5  import numpy as np
6  import re
7  import itertools
8  from collections import Counter
9  import sys
10 from scipy import sparse
11 from keras.preprocessing.text import Tokenizer
12
13 PAD = "<PAD/>"
14 def load_labels(pos_file_name, neg_file_name):
15     #TODO update comments
16     """
17     Loads polarity data from files, splits the data into words and
18        generates labels.
```

```
    Returns split sentences and labels.
    """
    pos_labels_cnt = 0
    neg_labels_cnt = 0

    with open(pos_file_name) as f:
        for line in f:
            pos_labels_cnt = pos_labels_cnt + 1

    with open(neg_file_name) as f:
        for line in f:
            neg_labels_cnt = neg_labels_cnt + 1

    y = np.concatenate([[1] * pos_labels_cnt, [0] * neg_labels_cnt], 0)

    return y


def build_vocab_new(files):
    """
    Builds a vocabulary mapping from word to index based on the sentences
        .
    Returns vocabulary mapping and inverse vocabulary mapping.
    """
    vocabulary = {}
    word_id_cnt = 1
    max_words_in_sentence = 0
    for file in files:
        with open(file) as f:
            for line in f:
                words = line.strip().split(" ")

                #comput maximum number of words in a sentence
```

```
52                if len(words) > max_words_in_sentence:
53                    max_words_in_sentence = len(words)
54
55                #build a mapping between word and numeric index
56                for word in words:
57                    if word not in vocabulary:
58                        vocabulary[word] = word_id_cnt
59                        word_id_cnt = word_id_cnt + 1
60
61    #Add the padding word to the dictionary
62    vocabulary[PAD] = 0
63
64    #build an inverse vocabulary
65    vocabulary_inv = {v: k for k, v in vocabulary.iteritems()}
66
67    #print(vocabulary_inv)
68
69    return [vocabulary, vocabulary_inv, max_words_in_sentence]
70
71 def build_input_data(files,labels, vocabulary, max_words_in_sentence):
72
73    labels_cnt = len(labels)
74    x = np.zeros((labels_cnt, max_words_in_sentence), dtype = np.int32)
75    sentence_ind = 0
76    for file in files:
77        with open(file) as f:
78            for line in f:
79                words = line.strip().split(" ")
80                word_ind = 0
81                for word in words:
82                    x[sentence_ind, word_ind] = vocabulary[word]
83                    word_ind = word_ind + 1
84                sentence_ind = sentence_ind +1
```

```
85
86      return x
87
88  def load_data():
89      """
90      Loads and preprocessed data for the dataset.
91      Returns input vectors, labels, vocabulary, and inverse vocabulary.
92      """
93      vocabulary, vocabulary_inv, max_words_in_sentence =
94      build_vocab_new(["./data/rt-polarity.pos", "./data/rt-polarity.neg"
95          ])
96      y = load_labels("./data/rt-polarity.pos", "./data/rt-polarity.neg")
97      x = build_input_data(["./data/rt-polarity.pos",
98      "./data/rt-polarity.neg"],y, vocabulary, max_words_in_sentence)
99
100     return [x, y, vocabulary, vocabulary_inv]
101
102 # Script model.py script
103
104 from keras.layers import Input, Dense, Embedding, merge,
105 Convolution2D, MaxPooling2D, Dropout, Lambda
106 from sklearn.cross_validation import train_test_split
107 from keras.layers.core import Reshape, Flatten
108 from keras.callbacks import ModelCheckpoint
109 from data_helpers import load_data
110 from keras.optimizers import Adam
111 from keras.models import Model
112 from sklearn.model_selection import StratifiedKFold
113 from keras import backend as K
114 import numpy as np
115 import h5py
116 import tensorflow as tf
117 from sklearn import metrics
```

```
118 from keras import metrics as k_metrics
119 import functools
120 from keras.utils import multi_gpu_model
121 from sklearn.utils import class_weight
122
123 seed = 5
124 np.random.seed(seed)
125
126 print('Loading data')
127 x, y, vocabulary, vocabulary_inv = load_data()
128
129 shuffle_indices = np.random.permutation(np.arange(len(y)))
130 x = x[shuffle_indices]
131 y = y[shuffle_indices]
132
133 sequence_length = x.shape[1]
134
135 vocabulary_size = len(vocabulary_inv)
136 print("Vocabulary Size: {:d}".format(len(vocabulary_inv)))
137
138 nb_classes=vocabulary_size
139 filter_sizes = [2,3,4,5]
140 num_filters = 100
141 drop = 0.5
142
143 epochs = 25
144 batch_size = 64
145
146 hidden_dims = 100
147
148
149 class_weight = class_weight.compute_class_weight('balanced', np.unique(y)
150     , y)
```

```
151
152 class_weight = {0: class_weight[0], 1: class_weight[1]}
153
154 print("Class Weight:",class_weight)
155
156 kfold = StratifiedKFold(n_splits=10, shuffle=False, random_state=seed)
157
158 cvscores = []
159 auc_scores = []
160
161 fold_counter = 0
162
163 for train, test in kfold.split(x, y):
164
165        fold_counter = fold_counter + 1
166
167        print("Current Fold : ", fold_counter)
168
169        # to save test data of the current fold to file
170
171        test_file_name = "test_data_" +  str(fold_counter)
172        true_labels_file_name = "true_labels_" + str(fold_counter)
173        train_lables_file_name = "train_labels" + str(fold_counter)
174        pred_file_name = "predictions_" + str(fold_counter)
175
176        test_indices = test.flatten().tolist()
177        #np.set_printoptions(threshold='nan')
178        file_test = open(test_file_name, 'a')
179        file_labels = open(true_labels_file_name, 'a')
180
181        for i in test_indices:
182               sent_text = np.vectorize(vocabulary_inv.get)(x[i]).tolist
183                   ()
```

```
184             new_sent_text = [j for j in sent_text if j != '<PAD/>']
185             final_sent = ' '.join(new_sent_text) + "\n"
186             file_test.write(final_sent)
187             file_labels.write('%d\n' % y[i])
188
189     file_test.close()
190     file_labels.close()
191
192     np.savetxt(train_lables_file_name,y[train],fmt="%1.2f")
193
194     input_shape = (sequence_length,)
195     output_shape = (input_shape[0], nb_classes)
196     inputs = Input(shape=input_shape,dtype='int32')
197
198
199     # One-Hot Encoding Layer
200
201     ohe=Lambda(K.one_hot,
202     arguments={'num_classes':nb_classes},
203     output_shape=output_shape)(inputs)
204
205     reshape = Reshape((sequence_length,nb_classes,1))(ohe)
206
207     conv_0 = Convolution2D(num_filters, filter_sizes[0], nb_classes,
208     border_mode='valid',
209     init='normal', activation='relu', dim_ordering='tf')(reshape)
210
211     conv_1 = Convolution2D(num_filters, filter_sizes[1], nb_classes,
212     border_mode='valid',
213     init='normal', activation='relu', dim_ordering='tf')(reshape)
214
215     conv_2 = Convolution2D(num_filters, filter_sizes[2], nb_classes,
216     border_mode='valid',
```

```
217         init='normal', activation='relu', dim_ordering='tf')(reshape)
218
219         conv_3 = Convolution2D(num_filters, filter_sizes[3], nb_classes,
220         border_mode='valid',
221         init='normal', activation='relu', dim_ordering='tf')(reshape)
222
223         maxpool_0 = MaxPooling2D(pool_size=(sequence_length -
224            filter_sizes[0] + 1, 1),
225         strides=(1,1), border_mode='valid', dim_ordering='tf')(conv_0)
226
227         maxpool_1 = MaxPooling2D(pool_size=(sequence_length -
228            filter_sizes[1] + 1, 1),
229         strides=(1,1), border_mode='valid', dim_ordering='tf')(conv_1)
230
231         maxpool_2 = MaxPooling2D(pool_size=(sequence_length -
232         filter_sizes[2] + 1, 1),
233         strides=(1,1), border_mode='valid', dim_ordering='tf')(conv_2)
234
235         maxpool_3 = MaxPooling2D(pool_size=(sequence_length -
236         filter_sizes[3] + 1, 1),
237         strides=(1,1), border_mode='valid', dim_ordering='tf')(conv_3)
238
239         merged_tensor = merge([maxpool_0, maxpool_1, maxpool_2, maxpool_3
240            ],
241         mode='concat', concat_axis=1)
242
243         flatten = Flatten()(merged_tensor)
244
245         dropout = Dropout(drop)(flatten)
246
247         dense = Dense(hidden_dims, activation="relu")(dropout)
248
249         output = Dense(1, activation="sigmoid")(dense)
```

```
250
251
252         # this creates a model that includes all the above layers.
253
254         with tf.device('/cpu:0'):
255             model = Model(input=inputs, output=output)
256
257         checkpoint = ModelCheckpoint('weights.best.hdf5', monitor='
258            val_acc',
259         verbose=1, save_best_only=True, mode='auto')
260
261         callbacks_list = [checkpoint]
262
263         adam = Adam(lr=1e-4, beta_1=0.9, beta_2=0.999, epsilon=1e-08)
264
265         parallel_model = multi_gpu_model(model, gpus=4)
266
267         parallel_model.compile(optimizer=adam,
268         loss='binary_crossentropy', metrics=['accuracy'])
269
270         parallel_model.__setattr__('callback_model',model)
271
272         parallel_model.fit(x[train], y[train], batch_size=batch_size,
273            epochs=epochs,
274         verbose=1, validation_split=0.1, class_weight = class_weight,
275         callbacks=callbacks_list) # starts training
276
277         # Load weights which gave best val accuracy and compile the model
278            .
279         Evaluate on test set and calculate metrics
280
281         print('Loading Best Weights..')
282
```

```
283         model.load_weights("weights.best.hdf5")
284 
285         model.compile(optimizer=adam, loss='binary_crossentropy', metrics
286             =['accuracy'])
287 
288         print('Evaluating on Best Weights..')
289 
290         scores = model.evaluate(x[test], y[test], verbose=0)
291 
292         yp = model.predict(x[test], batch_size=50, verbose=1)
293 
294         np.savetxt(pred_file_name,yp,fmt="%1.2f")
295 
296         auc = metrics.roc_auc_score(y[test],yp)
297 
298         print("AUC",auc)
299 
300         print("%s: %.2f%%" % ('Binary Classification Accuracy', scores
301             [1]*100))
302 
303         cvscores.append(scores[1] * 100)
304 
305         auc_scores.append(auc)
306 
307 
308 print("%.2f%% (+/- %.2f%%)" % (np.mean(cvscores), np.std(cvscores)))
309 
310 print("%.2f (+/- %.2f)" % (np.mean(auc_scores), np.std(auc_scores)))
311
```

Appendix F

Scripts For LMT Model

The below commands classify the combined records of bug reports and source files into linked and non-linked records using LMT Models. For both the train and test sets, we first convert the sentences into word vectors. For training set, we balance the positive and negative classes and then apply the LMT Models with 10-fold cross validation. We save the trained model and test it on the test set. The predictions on the test set are saved in the *Results_LR.txt* file.

```
1
2  module load java
3  export CLASSPATH=$CLASSPATH:/home/spoliset/tools/weka-3-8-2/weka.jar
4
5  java -Xmx200g weka.core.converters.TextDirectoryLoader
6  -dir /home/spoliset/weka_CLI_sample/Training >
7  /home/spoliset/weka_CLI_sample/Training/input_train.arff
8
9  java -Xmx200g weka.filters.unsupervised.attribute.StringToWordVector
10 -tokenizer "weka.core.tokenizers.WordTokenizer
11 -delimiters \" \\r\\n\\t.,;:\\\'\\\"()?\!\""
12 -W 10000 -i /home/spoliset/weka_CLI_sample/Training/input_train.arff
13 -o /home/spoliset/weka_CLI_sample/Training/output_train.arff
14
15 java -Xmx200g weka.filters.supervised.instance.ClassBalancer
16 -num-intervals 10
```

```
-i /home/spoliset/weka_CLI_sample/Training/output_train.arff
-o /home/spoliset/weka_CLI_sample/Training/output_train_cb.arff
-c first

java -Xms200g -Xmx500g weka.classifiers.functions.SimpleLogistic -I 0
-M 500 -H 50
-W 0.5
-t home/spoliset/weka_CLI_sample/Training/output_train_cb.arff
-x 10
-d /home/spoliset/weka_CLI_sample
/Training/LR.model
-c first 2>&1 | tee
/home/spoliset/weka_CLI_sample/Training/Results_LR.txt

java -Xmx200g weka.core.converters.TextDirectoryLoader
-dir /home/spoliset/project/spoliset/weka_CLI_sample/Testing >
/home/spoliset/weka_CLI_sample/Testing/input_test.arff

java -Xmx200g weka.filters.unsupervised.attribute.StringToWordVector
-tokenizer "weka.core.tokenizers.WordTokenizer
-delimiters \" \\r\\n\\t.,;:\\\'\\\"()?\!\""
-W 10000 -i /home/spoliset/weka_CLI_sample/Testing/input_test.arff
-o /home/spoliset/weka_CLI_sample/Testing/output_test.arff

java -Xms200g -Xmx500g weka.classifiers.misc.InputMappedClassifier
-L /home/spoliset/bow_LR/Eclipse/Buggy_Files_Reduced/Training/LR.model
-t /home/spoliset/weka_CLI_sample/Training/output_train_cb.arff
-T /home/spoliset/weka_CLI_sample/Testing/output_test.arff -M -v
-classifications "weka.classifiers.evaluation.output.prediction.CSV" >
/home/spoliset/weka_CLI_sample/Testing/Results_LR.txt
```

Appendix G

Evaluation Metrics Scripts

Scripts used to calculate AUC and other IR metrics: MAP, MRR, Top 5 for both CNN and LMT Models

```
1
2  # Functions For calculating Performance Metrics.
3  # Data should have the linked and non-linked records for a single
4  bug/test case and the actual labels.
5  # 1 is linked and 0 is not linked.
6
7  AP<-function(data){
8    relevant_record_indexes <- which(data[,2] == 1)
9    relevant_record_indexes_count <- length(relevant_record_indexes)
10   ap <- ifelse( relevant_record_indexes_count == 0, 0,
11         mean(seq(1:relevant_record_indexes_count)
12                /relevant_record_indexes))
13   return(ap)
14
15 }
16
17 ap_lbls<-function(labels, positive_lbl = 2){
18   relevant_record_indexes <- which(labels == positive_lbl)
19   relevant_record_indexes_count <- length(relevant_record_indexes)
20   ap <- ifelse( relevant_record_indexes_count == 0, 0,
```

```
              mean( seq(1:relevant_record_indexes_count) /
                 relevant_record_indexes ))
}

RR<-function(data){
  reciprocal_rank<-1/(which(data[,2]== 1)[1])
  return(reciprocal_rank)
}

TOP_10<-function(data){
  # Hits/Total Suggestions
  top_10_relevant_record_indexes <- which(data[1:10,2] == 1)
  top_10_relevant_record_indexes_count <- length(
     top_10_relevant_record_indexes)
  return(top_10_relevant_record_indexes_count)
}

MAP<-function(aps){
  return(mean(aps[aps!=0]))
}

MRR<-function(rrs){
  return(mean(rrs[!is.na(rrs)]))
}

# bug reports which have atleast 1 relevant file in its top 10 results of
source files divided by no of bugs which have relevant files in the test
   set
TOP10<-function(top10,n){
  return(length(top10[top10 > 0])/n)
}


```

```
# To calculate MAP, MRR and Top 10 Rank for the CNN model
# 1 Linked, 0 Not Linked
# Files needed:
  # the test set with the actual combined sentences of bug report and
     source file
  # the total file with bug id and combined sentences generated in data
     generation.
  # the true labels of the records in the test set.
  # the predicitions made by the model on the test set.

library(data.table)

df.total.set <- fread(
file="/Volumes/CORSAIR/CNN-Lesser-Source-Files/
bug_id_sentence",sep="\t",
col.names = c("bug_id","sentence"))

df.test_data <- read.csv(
file="~/Downloads/test_data_1",header = FALSE,
col.names = "sentence")

df.test_data$sentence<-as.character(df.test_data$sentence)
df.true_labels <-read.csv(
file="~/Downloads/true_labels_1",
header = FALSE,col.names="relatedness_score")

df.preds<-read.csv(
file="~/Downloads/predictions_1"
,header = FALSE,col.names="pobability_linked")

# Add bug ids to the test data to identify the combined records
pertaining to a bug id.
df.test_data$bug_id <-
```

```
87  df.total.set$bug_id[match(df.test_data$sentence, df.total.set$sentence)]
88
89  # cbind all the above to get the bugid, probability_linked,
90     relatedness_score
91  df.MAP_file<-cbind(df.test_data[2],df.preds,df.true_labels)
92  df.MAP_file <- df.MAP_file[order(df.MAP_file$bug_id),]
93
94  # Save File
95  # This file has the bug id, prediction score, true label
96  write.table(df.MAP_file,file="~/Downloads/test_map",
97  col.names = FALSE,sep=",",row.names = FALSE)
98  Test_Case_Preds="~/Downloads/test_map"
99
100 ap<-function(data,average_prec_all,cnt_sentences){
101   data<-data[rowSums(is.na(data))!=ncol(data), ]
102   data <- data[order(-data[,1]),]
103   aps<-AP(data)
104   cat(paste('AP:',aps,"\n"))
105   average_prec_all[cnt_sentences]<-aps
106   return(average_prec_all)
107 }
108
109 rr<-function(data,rr_all,cnt_sentences){
110   dat<-dat[rowSums(is.na(dat))!=ncol(dat), ]
111   dat <- dat[order(-dat[,1]), ]
112   rrs<-RR(dat)
113   cat(paste('RR:',rrs,"\n"))
114   rr_all[cnt_sentences]<-rrs
115   return(rr_all)
116
117 }
118
119 top_10<-function(data,top_10_all,cnt_sentences,bugs_with_rel_files){
```

```
  dat<-dat[rowSums(is.na(dat))!=ncol(dat), ]
  dat <- dat[order(-dat[,1]), ]
  top_tens<-TOP_10(dat)
  cat(paste('Top 10:',top_tens,"\n"))
  top_10_all[cnt_sentences]<-top_tens
  return(top_10_all)
}
average_prec_all<-vector(mode="numeric")
rr_all<-vector(mode="numeric")
top_10_all<-vector(mode="numeric")

prev_sentence<-'dummy'
dat<-matrix(nrow=1000000,ncol=2)
con = file(Test_Case_Preds, "r")
cnt_sentences<-0
cnt<-0
line_no<-0
while ( TRUE ) {
  line = readLines(con, n = 1)
  line_no<-line_no+1
  cat(paste('Reading Line:',line_no,"\n"))
  cnt<-cnt+1
  if ( length(line) == 0 ) {
    break
  }
  line_split<-unlist(strsplit(line, ","))
  current_sentence<-line_split[1]
  if(prev_sentence != current_sentence){
    cnt_sentences<-cnt_sentences+1
    cat(paste('This is a New Bug:',cnt_sentences,"\n"))
    # get the prev sentence and calculate AP
    if(!all(is.na(dat))){
      cat(paste('Calculating MAP and MRR of sentence:',
```

```
      (cnt_sentences-1),"\n"))
      average_prec_all<-ap(dat,average_prec_all,cnt_sentences-1)
      rr_all<-rr(dat,rr_all,cnt_sentences-1)
      top_10_all<-top_10(dat,top_10_all,cnt_sentences-1)
      dat<-matrix(nrow=1000000,ncol=2)
      cnt<-1
    }
  }
  prev_sentence<-current_sentence
  dat[cnt,1]<-as.numeric(line_split[2])
  dat[cnt,2]<-as.numeric(line_split[3])

}
close(con)

# For calculating the AP for the last sentence in the data
average_prec_all<-ap(dat,average_prec_all,cnt_sentences)

# For calculating the RR for the last sentence in the data
rr_all<-rr(dat,rr_all,cnt_sentences)

# For calculating the Top 10 for the last sentence in the data
top_10_all<-top_10(dat,top_10_all,cnt_sentences)

# Final MAP
mean_average_precision<-MAP(average_prec_all)
round(mean_average_precision,5)

# Final MRR
mean_reciprocal_rank<-MRR(rr_all)
round(mean_reciprocal_rank,5)

# Final Top 10 Rank
```

```
no_bugs_rel<-length(average_prec_all[average_prec_all!=0])
top_10_rank<-TOP10(top_10_all,no_bugs_rel)
round(top_10_rank,5)

# Calculate AUC
df.true_labels$relatedness_score<-as.factor(df.
   true_labels$relatedness_score)
auc(roc(df.preds$pobability_linked,df.true_labels$relatedness_score))

# histogram of AP distibution
hist(average_prec_all[average_prec_all!=0],
main = "Distribution of AP",xlab="AP",ylab="Bug Reports")

# histogram of Top 10 distirbution
hist(top_10_all[which(average_prec_all!=0)],
main = "Distribution of Top 10 Rank",
xlab="Number of Relevant Files In Top-10",ylab="Bug Reports")

# histogram of RR distribution
hist(rr_all[which(average_prec_all!=0)],
main = "Distribution of RR",xlab="RR",ylab="Bug Reports")


## R Script to convert the predicted values to probability linked scores
library(tidyr)
df.preds<-read.csv(file="/Volumes/CORSAIR/Results/LR/AspectJ/All_Files/
   Results_LR.txt", header = TRUE,sep = ",")

df.preds["probability_linked"]<--1

# Updat the above column based on the below logic:
# Actual,Predicted,Probability_Belongs_To_Predicted_Class,
   Probability_Belongs_To_Positive_Class
```

```
# pos neg p 1-p
# neg pos p p
# pos pos p p
# neg neg p 1-p

df.preds$prediction<-as.numeric(df.preds$prediction)
df.preds$actual<-as.character(df.preds$actual)
df.preds$predicted<-as.character(df.preds$predicted)

df.preds[df.preds$actual=="2:neg" & df.preds$predicted=="1:pos" & df.
    preds$prediction< 1.000,"probability_linked"]<-
  df.preds[df.preds$actual=="2:neg" & df.preds$predicted=="1:pos" & df.
      preds$prediction< 1.000,"prediction"]

df.preds[df.preds$actual=="1:pos" & df.preds$predicted=="2:neg" & df.
    preds$prediction< 1.000,"probability_linked"]<-
  1-(df.preds[df.preds$actual=="1:pos" & df.preds$predicted=="2:neg" & df
      .preds$prediction< 1.000,"prediction"])

df.preds[df.preds$actual=="2:neg" & df.preds$predicted=="2:neg" & df.
    preds$prediction< 1.000,"probability_linked"]<-
  1-(df.preds[df.preds$actual=="2:neg" & df.preds$predicted=="2:neg" & df
      .preds$prediction< 1.000,"prediction"])

df.preds[df.preds$actual=="1:pos" & df.preds$predicted=="1:pos" & df.
    preds$prediction< 1.000,"probability_linked"]<-
  df.preds[df.preds$actual=="1:pos" & df.preds$predicted=="1:pos" & df.
      preds$prediction< 1.000,"prediction"]


write.table(df.preds[,"probability_linked"],file="/Volumes/CORSAIR/
    Results/LR/AspectJ/All_Files/derived_predictions_LR.csv",col.names =
    FALSE,row.names=FALSE,sep=",")
```

```
252
253 ##########################################################################
254 # To calculate MAP, MRR and Top 10 Rank for the LMT model
255 # 1 Linked, 0 Not Linked
256 # Files needed:
257 # (1) The test set with the actual combined sentences of bug report and
258 source file.
259 # (2) The total file with bug id and combined sentences generated during
260 data generation in R.
261 # (3) The true labels of the records in the test set.
262 # (4) The predictions made by the model on the test set.
263 ##########################################################################
264 library(data.table)
265 library(stringr)
266 library(tm)
267 library(AUC)
268 # For bag of words files
269 df.total.set<-
270     fread(file="/Volumes/CORSAIR/Results/
271     AspectJ/bag_of_words/All_Files/bug_id_sentence",
272     sep="\t",col.names = "sentence")
273 df.total.set$bug_id <- word(df.total.set$sentence,1)
274 df.total.set$sentence<-removeNumbers(df.total.set$sentence)
275 df.total.set$sentence<-trimws(df.total.set$sentence,which="both")
276
277 df.pos.test_data<-
278     read.csv(file="/Volumes/CORSAIR/Results/
279     LR/AspectJ/All_Files/rt-polarity.pos.test",header = FALSE,
280     col.names = "sentence")
281 df.pos.test_data["true_labels"]<-1
282
283 df.neg.test_data<-
284     read.csv(file="/Volumes/CORSAIR/Results/
```

```
285     LR/AspectJ/All_Files/rt-polarity.neg.test",header = FALSE,
286     col.names = "sentence")
287 df.neg.test_data["true_labels"]<-0
288
289 df.test_data<-rbind(df.pos.test_data,df.neg.test_data)
290 df.true_labels<-as.data.frame(df.test_data[,"true_labels"])
291 colnames(df.true_labels)<-"relatedness_score"
292 df.test_data$sentence<-as.character(df.test_data$sentence)
293 df.test_data$sentence<-trimws(df.test_data$sentence,which="both")
294
295 # Probabilities derived after converting the model predictions
296 # to prob_linked
297 df.preds<-
298     read.csv(file="/Volumes/CORSAIR/Results/
299     LR/AspectJ/All_Files/derived_predictions_LR.csv",
300     header = FALSE,
301     col.names = "probability_linked")
302
303 # Add bug ids to the test data to identify the combined records
304    pertaining
305 # to a bug id.
306 df.test_data$bug_id<-df.total.set$bug_id[match(df.test_data$sentence,
307     df.total.set$sentence)]
308
309 # cbind all the above to get the bugid, true_labels, prediction
310
311 df.MAP_file<-cbind(df.test_data[,c("bug_id","true_labels")],df.preds)
312 df.MAP_file <- df.MAP_file[order(df.MAP_file$bug_id),]
313
314 # Re-ordering the columns : bug id, prediction score, true label
315 df.MAP_file<-df.MAP_file[c(1,3,2)]
316
317 # Save File. This file has the bug id, prediction score, true label
```

```
318 write.table(df.MAP_file,
319     file="/Volumes/CORSAIR/Results/LR/AspectJ/All_Files/test_map_LR",
320     col.names = FALSE,sep=",",
321     row.names = FALSE)
322 Test_Case_Preds="/Volumes/CORSAIR/Results/LR/AspectJ/All_Files/
323     test_map_LR"
324
325 ap<-function(data,average_prec_all,cnt_sentences){
326   data<-data[rowSums(is.na(data))!=ncol(data), ]
327   data <- data[order(-data[,1]),]
328   aps<-AP(data)
329   cat(paste('AP:',aps,"\n"))
330   average_prec_all[cnt_sentences]<-aps
331   return(average_prec_all)
332 }
333
334 rr<-function(data,rr_all,cnt_sentences){
335   dat<-dat[rowSums(is.na(dat))!=ncol(dat), ]
336   dat <- dat[order(-dat[,1]), ]
337   rrs<-RR(dat)
338   cat(paste('RR:',rrs,"\n"))
339   rr_all[cnt_sentences]<-rrs
340   return(rr_all)
341
342 }
343
344 top_5<-function(data,top_5_all,cnt_sentences,bugs_with_rel_files){
345   dat<-dat[rowSums(is.na(dat))!=ncol(dat), ]
346   dat <- dat[order(-dat[,1]), ]
347   top_fives<-TOP_5(dat)
348   cat(paste('Top 5:',top_fives,"\n"))
349   top_5_all[cnt_sentences]<-top_fives
350   return(top_5_all)
```

```
}
average_prec_all<-vector(mode="numeric")
rr_all<-vector(mode="numeric")
top_5_all<-vector(mode="numeric")
prev_sentence<-'dummy'
dat<-matrix(nrow=1000000,ncol=2)
con = file(Test_Case_Preds, "r")
cnt_sentences<-0
cnt<-0
line_no<-0
while ( TRUE ) {
  line = readLines(con, n = 1)
  line_no<-line_no+1
  cat(paste('Reading Line:',line_no,"\n"))
  cnt<-cnt+1
  if ( length(line) == 0 ) {
    break
  }
  line_split<-unlist(strsplit(line, ","))
  current_sentence<-line_split[1]
  if(prev_sentence != current_sentence){
    cnt_sentences<-cnt_sentences+1
    cat(paste('This is a New Bug:',cnt_sentences,"\n"))
    # get the prev sentence and calculate AP
    if(!all(is.na(dat))){
      cat(paste('Calculating MAP and MRR of sentence:'
      ,(cnt_sentences-1),"\n"))
      average_prec_all<-ap(dat,average_prec_all,cnt_sentences-1)
      rr_all<-rr(dat,rr_all,cnt_sentences-1)
      top_5_all<-top_5(dat,top_5_all,cnt_sentences-1)
      dat<-matrix(nrow=1000000,ncol=2)
      cnt<-1
    }
```

```
  }
  prev_sentence<-current_sentence
  dat[cnt,1]<-as.numeric(line_split[2])
  dat[cnt,2]<-as.numeric(line_split[3])

}
close(con)

# For calculating the AP for the last sentence in the data
average_prec_all<-ap(dat,average_prec_all,cnt_sentences)

# For calculating the RR for the last sentence in the data
rr_all<-rr(dat,rr_all,cnt_sentences)

# For calculating the Top 10 for the last sentence in the data
top_5_all<-top_5(dat,top_5_all,cnt_sentences)
# Final MAP
mean_average_precision<-MAP(average_prec_all)
round(mean_average_precision,5)

# Final MRR
mean_reciprocal_rank<-MRR(rr_all)
round(mean_reciprocal_rank,5)

# Final Top 5 Rank
no_bugs_rel<-length(average_prec_all[average_prec_all!=0])
top_5_rank<-TOP5(top_5_all,no_bugs_rel)
round(top_5_rank,5)

df.true_labels$relatedness_score<-as.factor(df.
   true_labels$relatedness_score
   )auc(roc(df.preds$probability_linked,df.true_labels$relatedness_score
      ))
```

```
417 # histogram of AP distribution
418 hist(average_prec_all[average_prec_all!=0],
419     main = "Distribution of AP",xlab="AP",
420     ylab="Bug Reports")
421
422 # histogram of Top 5 distribution
423 hist(top_5_all[which(average_prec_all!=0)],
424     main = "Distribution of Top 5 Rank",
425     xlab="Number of Relevant Files In Top-5",
426     ylab="Bug Reports")
427
428 # histogram of RR distribution
429 hist(rr_all[which(average_prec_all!=0)],
430     main = "Distribution of RR",
431     xlab="RR", ylab="Bug Reports")
432
```

References

[1] Apache Tomcat. http://tomcat.apache.org/.

[2] Bugzilla. https://bugs.eclipse.org/bugs/.

[3] Caffe. https://github.com/BVLC/caffe/blob/master/docs/multigpu.md.

[4] Drawing Convolution Neural Nets. https://github.com/gwding/draw_convnet.

[5] Eclipse Foundation. https://github.com/eclipse.

[6] Eclipse foundation — the eclipse foundation. https://www.eclipse.org/org/foundation/. (Accessed on 06/30/2018).

[7] Eclipse Java development tools (JDT). http://www.eclipse.org/jdt/.

[8] Enabling open innovation & collaboration — the eclipse foundation. https://www.eclipse.org/. (Accessed on 06/30/2018).

[9] GPU vs CPU? What is GPU Computing?—NVIDIA. http://www.nvidia.ca/object/what-is-gpu-computing.html.

[10] hub manual. https://hub.github.com/hub.1.html.

[11] Jira — Issue & Project Tracking Software — Atlassian. https://www.atlassian.com/software/jira.

[12] Platform UI. http://projects.eclipse.org/projects/eclipse.platform.ui.

[13] SWT: The Standard Widget Toolkit. http://www.eclipse.org/swt/.

[14] The AspectJ Project — The Eclipse Foundation. https://www.eclipse.org/aspectj/.

[15] The Eclipse Project. http://www.eclipse.org/eclipse/.

[16] Using GPUs - TensorFlow. https://www.tensorflow.org/programmers_guide/using_gpu.

[17] Using GPUs — Keras. https://keras.io/getting-started/faq/#how-can-i-run-a-keras-model-on-multiple-gpus.

[18] Using GPUs — Theano. http://deeplearning.net/software/theano/tutorial/using_multi_gpu.html.

[19] Weka - ClassBalancer. http://weka.sourceforge.net/doc.dev/weka/filters/supervised/instance/ClassBalancer.html. (Accessed on 05/16/2018).

[20] Weka - Running from the command line. https://www.cs.waikato.ac.nz/~remco/weka_bn/node13.html. (Accessed on 05/15/2018).

[21] Weka - SimpleLogistic. http://weka.sourceforge.net/doc.dev/weka/classifiers/functions/SimpleLogistic.html. (Accessed on 05/15/2018).

[22] Weka - StringToWordVector. http://weka.sourceforge.net/doc.dev/weka/filters/unsupervised/attribute/StringToWordVector.html. (Accessed on 05/15/2018).

[23] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. JC. Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[24] J. Anvik. Automating bug report assignment. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 937–940, New York, NY, USA, 2006. ACM.

[25] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[26] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu. Advances in optimizing recurrent networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8624–8628, May 2013.

[27] Y. Bengio et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[28] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318. ACM, 2008.

[29] D. M. Blei. Probabilistic topic models. *Commun. ACM*, 55(4):77–84, April 2012.

[30] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[31] Y. Boureau, J. Ponce, and Y. LeCun. A theoretical analysis of feature pooling in visual recognition. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 111–118, 2010.

[32] E.J. Braude and M.E. Bernstein. *Software Engineering: Modern Approaches, Second Edition*. John Wiley, 2016.

[33] G. Candea, S. Bucur, and C. Zamfir. Automated software testing as a service. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 155–160. ACM, 2010.

[34] K. Chang, V. Bertacco, and I. L. Markov. Simulation-based bug trace minimization with bmc-based refinement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(1):152–165, 2007.

[35] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 160–167, New York, NY, USA, 2008. ACM.

[36] R. Collobert, J. Weston, Léon L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537, November 2011.

[37] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[38] D. Curro, K. G. Derpanis, and A. V. Miranskyy. Building usage profiles using deep neural nets. In *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track*, ICSE-NIER '17, pages 43–46, Piscataway, NJ, USA, 2017. IEEE Press.

[39] C. Nogueira D. Santos and B. Zadrozny. Learning character-level representations for part-of-speech tagging. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages II–1818–II–1826. JMLR.org, 2014.

[40] G. E. Dahl, T. N. Sainath, and G. E. Hinton. Improving deep neural networks for lvcsr using rectified linear units and dropout. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8609–8613, May 2013.

[41] L. Deng, D. Yu, et al. Deep learning: methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014.

[42] C. Dugas, Y. Bengio, F. Bélisle, C. Nadeau, and R. Garcia. Incorporating second-order functional knowledge for better option pricing. In *Advances in neural information processing systems*, pages 472–478, 2001.

[43] S. T. Dumais. Latent semantic analysis. *Annual review of information science and technology*, 38(1):188–230, 2004.

[44] A. Feldman and A. van Gemund. A two-step hierarchical algorithm for model-based diagnosis. 2006.

[45] J. Friedman, T. Hastie, R. Tibshirani, et al. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2):337–407, 2000.

[46] K. Ganesan, C. Zhai, and J. Han. Opinosis: a graph-based approach to abstractive summarization of highly redundant opinions. In *Proceedings of the 23rd international conference on computational linguistics*, pages 340–348. Association for Computational Linguistics, 2010.

[47] J. Gao, P. Pantel, M. Gamon, X. He, and L. Deng. Modeling interestingness with deep neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2–13, 2014.

[48] J. Guo, J. Cheng, and J. Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 3–14, Piscataway, NJ, USA, 2017. IEEE Press.

[49] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[50] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

[51] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[52] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

[53] M. Hu and B. Liu. Mining and summarizing customer reviews. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 168–177, New York, NY, USA, 2004. ACM.

[54] X. Huo, M. Li, and Z. Zhou. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 1606–1612. AAAI Press, 2016.

[55] A. G. Ivakhnenko. Polynomial theory of complex systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-1(4):364–378, Oct 1971.

[56] R. Johnson and T. Zhang. Effective use of word order for text categorization with convolutional neural networks. *CoRR*, abs/1412.1058, 2014.

[57] R. Johnson and T. Zhang. Semi-supervised convolutional neural networks for text categorization via region embedding. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, pages 919–927, Cambridge, MA, USA, 2015. MIT Press.

[58] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282. ACM, 2005.

[59] N. Kalchbrenner, E. Grefenstette, and P. Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.

[60] D. Kim, Y. Tao, S. Kim, and A. Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE Transactions on Software Engineering*, 39(11):1597–1610, Nov 2013.

[61] Y. Kim, Y. Jernite, D. Sontag, and A. M. Rush. Character-aware neural language models. *CoRR*, abs/1508.06615, 2015.

[62] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[63] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. An empirical study of adoption of software testing in open source projects. In *Quality Software (QSIC), 2013 13th International Conference on*, pages 103–112. IEEE, 2013.

[64] P. S. Kochhar, X. Xia, D. Lo, and S. Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 165–176, New York, NY, USA, 2016. ACM.

[65] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[66] S. Lal and A. Sureka. A static technique for fault localization using character n-gram based information retrieval model. In *Proceedings of the 5th India Software Engineering Conference*, ISEC '12, pages 109–118, New York, NY, USA, 2012. ACM.

[67] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 476–481, Nov 2015.

[68] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 218–229, May 2017.

[69] N. Landwehr, M. Hall, and E. Frank. Logistic model trees. *Machine Learning*, 59(1):161–205, May 2005.

[70] T. B. Le, R. J. Oentaryo, and D. Lo. Information retrieval and spectrum based bug localization: better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 579–590. ACM, 2015.

[71] Y. LeCun et al. Lenet-5, convolutional neural networks.

[72] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive datasets*. Cambridge university press, 2014.

[73] K. M. Leung. Naive bayesian classifier. *Polytechnic University Department of Computer Science/Finance and Risk Engineering*, 2007.

[74] X. Li and D. Roth. Learning question classifiers. In *Proceedings of the 19th international conference on Computational linguistics-Volume 1*, pages 1–7. Association for Computational Linguistics, 2002.

[75] X. Li and L. Zhang. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):92, 2017.

[76] T. Liu et al. Learning to rank for information retrieval. *Foundations and Trends® in Information Retrieval*, 3(3):225–331, 2009.

[77] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Source code retrieval for bug localization using latent dirichlet allocation. In *2008 15th Working Conference on Reverse Engineering*, pages 155–164, Oct 2008.

[78] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[79] W. Mayer and M. Stumptner. Model-based debugging–state of the art and future challenges. *Electronic Notes in Theoretical Computer Science*, 174(4):61–82, 2007.

[80] M. A. Mazurowski, P. A. Habas, J. M. Zurada, J. Y. Lo, J. A. Baker, and G. D. Tourassi. Training neural network classifiers for medical decision making: The effects of imbalanced datasets on classification performance. *Neural networks*, 21(2-3):427–436, 2008.

[81] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[82] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[83] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 1287–1293. AAAI Press, 2016.

[84] S. S. Murtaza, A. Hamou-Lhadj, N. H. Madhavji, and M. Gittens. An empirical study on the use of mutant traces for diagnosis of faults in deployed systems. *J. Syst. Softw.*, 90:29–44, April 2014.

[85] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 263–272, Nov 2011.

[86] R. Pal. *Predictive Modeling of Drug Sensitivity*. Academic Press, 2016.

[87] B. Pang, L. Lee, and S. Vaithyanathan. Thumbs up? sentiment classification using machine learning techniques. In *Proceedings of EMNLP*, pages 79–86, 2002.

[88] J. Pennington, R. Socher, and C. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[89] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[90] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: An approach to debugging evolving programs. *ACM Trans. Softw. Eng. Methodol.*, 21(3):19:1–19:29, July 2012.

[91] D. R. Radev, H. Qi, H. Wu, and W. Fan. Evaluating web-based question answering systems. *Ann Arbor*, 1001:48109.

[92] S. Rao and A. Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 43–52, New York, NY, USA, 2011. ACM.

[93] S. Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[94] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[95] D. Saha, M. G. Nanda, P. Dhoolia, V. K. Nandivada, V. Sinha, and S. Chandra. Fault localization for data-centric programs. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 157–167. ACM, 2011.

[96] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355, Nov 2013.

[97] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, November 1975.

[98] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[99] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[100] H. Schütze, C. D. Manning, and P. Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press, 2008.

[101] A. Shaffy. One-Hot Encoding of Text. `https://medium.com/@athif.shaffy/one-hot-encoding-of-text-b69124bef0a7`.

[102] Y. Shen, X. He, J. Gao, L. Deng, and G. Mesnil. A latent semantic model with convolutional-pooling structure for information retrieval. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 101–110. ACM, 2014.

[103] A. Singhal et al. Modern information retrieval: A brief overview. 2001.

[104] B. Sisman and A. C. Kak. Incorporating version histories in information retrieval based bug localization. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 50–59. IEEE, 2012.

[105] P. Smolensky. Information processing in dynamical systems: Foundations of harmony theory. Technical report, COLORADO UNIV AT BOULDER DEPT OF COMPUTER SCIENCE, 1986.

[106] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[107] I. Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, USA, 9th edition, 2010.

[108] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[109] M. Sumner, E. Frank, and M. Hall. Speeding up logistic model tree induction. In *Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases*, PKDD'05, pages 675–683, Berlin, Heidelberg, 2005. Springer-Verlag.

[110] M. Sumner, E. Frank, and M. Hall. Speeding up logistic model tree induction. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 675–683. Springer, 2005.

[111] Y. Sun, L. Lin, D. Tang, N. Yang, Z. Ji, and X. Wang. Modeling mention, context and entity with neural networks for entity disambiguation. In *IJCAI*, pages 1333–1339, 2015.

[112] C. Szegedy, A. Toshev, and D. Erhan. Deep neural networks for object detection. In *Advances in neural information processing systems*, pages 2553–2561, 2013.

[113] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[114] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59, Sept 2012.

[115] M. van Gerven and S. Bohte. *Artificial neural networks as models of neural information processing*. Frontiers Media SA, 2018.

[116] E. M. Voorhees and D. Harman. Overview of the sixth text retrieval conference (trec-6). *Information Processing & Management*, 36(1):3–35, 2000.

[117] B. C. Wallace, L. Kertz, E. Charniak, et al. Humans require context to infer ironic intent (so computers probably do, too). In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 512–516, 2014.

[118] S. Wang, D. Lo, and J. Lawall. Compositional vector space models for improved bug localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 171–180, Sept 2014.

[119] J. Wegener, K. Grimm, M. Grochtmann, H. Sthamer, and B. Jones. Systematic testing of real-time systems. In *4th International Conference on Software Testing Analysis and Review (EuroSTAR 96)*, 1996.

[120] J. Weston, S. Chopra, and K. Adams. Semantic embeddings from hashtags. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1822–1827, 2014.

[121] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 334–345, Piscataway, NJ, USA, 2015. IEEE Press.

[122] J. Wiebe, T. Wilson, and C. Cardie. Annotating expressions of opinions and emotions in language. *Language resources and evaluation*, 39(2-3):165–210, 2005.

[123] R. J. Wieringa and M. Daneva. Six strategies for generalizing software engineering theories. *Science of computer programming*, 101:136–152, 4 2015. eemcs-eprint-25555.

[124] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Computer Science. Springer Berlin Heidelberg, 2012.

[125] T. Wu, S. Liu, J. Zhang, and Y. Xiang. Twitter spam detection based on deep learning. In *Proceedings of the Australasian Computer Science Week Multiconference*, page 3. ACM, 2017.

[126] M. Li X. Huo. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 1909–1915, 2017.

[127] F. Xia, T. Liu, J. Wang, W. Zhang, and H. Li. Listwise approach to learning to rank: theory and algorithm. In *Proceedings of the 25th international conference on Machine learning*, pages 1192–1199. ACM, 2008.

[128] Z. Xiang and Y. LeCun. Text Understanding From Scratch. *CoRR*, abs/1502.01710, 2015.

[129] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*, pages 17–26. IEEE, 2015.

[130] X. Ye, R. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 689–699, New York, NY, USA, 2014. ACM.

[131] R.K. Yin. *Case Study Research: Design and Methods*. Applied Social Research Methods. SAGE Publications, 2009.

[132] K. Yoon. Convolutional Neural Networks for Sentence Classification. *CoRR*, abs/1408.5882, 2014.

[133] D. Zeng, K. Liu, S. Lai, G. Zhou, and J. Zhao. Relation classification via convolutional deep neural network. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 2335–2344, 2014.

[134] X. Zhang, J. J. Zhao, and Y. LeCun. Character-level convolutional networks for text classification. *CoRR*, abs/1509.01626, 2015.

[135] Y. Zhang and B. Wallace. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *arXiv preprint arXiv:1510.03820*, 2015.

[136] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 14–24, June 2012.

[137] Z. Zhou and X. Liu. Training cost-sensitive neural networks with methods addressing the class imbalance problem. *IEEE Transactions on Knowledge and Data Engineering*, 18(1):63–77, Jan 2006.