



Run Run Shaw Library

香港城市大學
City University of Hong Kong

Copyright Warning

Use of this thesis/dissertation/project is for the purpose of private study or scholarly research only. ***Users must comply with the Copyright Ordinance.***

Anyone who consults this thesis/dissertation/project is understood to recognise that its copyright rests with its author and that no part of it may be reproduced without the author's prior written consent.

CITY UNIVERSITY OF HONG KONG
香港城市大學

Lexical Mismatch Analysis for Bug Localization
along with Extracting Semantic and
Structural Features
錯誤定位的詞法不匹配分析以及語義和結構
特徵的提取

Submitted to
Department of Computer Science
電腦科學系
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
哲學博士學位

by

Xiao Yan
肖艷

August 2019
二零一九年八月

Abstract

Automatic localization of buggy files can speed up the process of bug fixing to improve the efficiency and productivity of software quality assurance teams. The challenge of locating bugs in mostly large-scale software systems has led to the development of bug localization techniques. Most prevalent approaches based on information retrieval, machine learning and deep learning techniques achieve encouraging results. However, several challenges still lie in the field of bug localization given bug reports. First, bug reports are written in natural languages, while source files are written in code tokens. The lexical mismatch between bug reports and source codes degrades the performances of existing information retrieval or machine learning-based approaches. Furthermore, both bug reports and source files are written in words by humans, where useful semantic information is available, but it is usually underutilized by existing bug localization approaches. On the other hand, compared to natural languages, programs contain more stringent structural information. Correctly localizing buggy files for bug reports together with their semantic and structural information is a crucial task, which would essentially improve the accuracy of bug localization techniques.

In this thesis, we introduce several models to address the aforementioned problems. First, to bridge the lexical gap and improve the effectiveness of localizing buggy files by leveraging the extracted semantic information from bug reports and source code, we present BugTranslator, a novel deep learning-based machine trans-

lation technique composed of an attention-based recurrent neural network (RNN) Encoder-Decoder with long short-term memory cells. One RNN encodes bug reports into several context vectors that are decoded by another RNN into code tokens of buggy files. The technique studies and adopts the relevance between the extracted semantic information from bug reports and source files. An enhanced version using character-level convolutional neural network (CNN) is then discussed to improve the accuracy of bug localization for bug reports by expressing them in character and analyzing them with a language model. The proposed model is composed of two main parts: character-level CNN and RNN language model. Both bug reports and source files are expressed in a character level and then input into a CNN, whose output is given to an RNN encoder-decoder architecture. Bug-Translator and the enhanced version distinguish bug reports and source code into different symbolic classes and then extracts deep semantic similarity and relevance between bug reports and the corresponding buggy files to bridge the lexical gap at its source, thereby further improving the accuracy of bug localization.

Second, to preserve semantic information, we adapt word embedding techniques to transform words in bug reports and source files into word vectors. Meanwhile, to parse semantics from word vectors, we enhance CNN by making use of the important bug-fixing experience (bug-fixing recency and frequency) besides the semantic information that can be extracted by conventional CNN from bug reports and source files. Based on these, DeepLocator and DeepLoc, two deep learning-based models are proposed to improve the accuracy of bug localization by making full use of semantic information. DeepLoc is capable of automatically connecting bug reports to the corresponding buggy files and achieves better performance than four other existing approaches based on a deep understanding of semantics in bug reports and source code.

Third, to empirically evaluate and demonstrate the effects of both semantic and

structural information in bug reports and source files on improving the accuracy of bug localization, CNN_Forest is proposed, which involves CNN and an ensemble of random forests that have excellent performance in the tasks of semantic parsing and structural information extraction. CNN_Forest uses CNN with multiple filters and an ensemble of random forests with multi-grained scanning to extract semantic and structural features from the word vectors derived from bug reports and source files. And a subsequent cascade forest (a cascade of ensembles of random forests) is used to further extract deeper features and observe the correlated relationships between bug reports and source files. CNN_Forest is capable of defining the correlated relationships between bug reports and source files, and we empirically show that semantic and structural information in bug reports and source files are crucial in improving bug localization.

Our experiments indicate that the proposed models can increase the accuracy of bug localization and empirically demonstrate the significance of including both semantic and structural information in bug localization and bridging the lexical gap between bug reports and source files. We hope that the promising results will interest and encourage more in-depth research in these problems to further improve the accuracy of bug localization.

CITY UNIVERSITY OF HONG KONG
Qualifying Panel and Examination Panel

Surname: XIAO
First Name: Yan
Degree: PhD
College/Department: Department of Computer Science

The Qualifying Panel of the above student is composed of:

Supervisor(s)

Dr. KEUNG Wai Jacky	Department of Computer Science City University of Hong Kong
---------------------	--

Qualifying Panel Member(s)

Dr. LI Minming	Department of Computer Science City University of Hong Kong
----------------	--

Prof. ZHANG Qingfu	Department of Computer Science City University of Hong Kong
--------------------	--

This thesis has been examined and approved by the following examiners:

Dr. KWOK Lam For	Department of Computer Science City University of Hong Kong
------------------	--

Dr. YU Yuen Tak	Department of Computer Science City University of Hong Kong
-----------------	--

Dr. KEUNG Wai Jacky	Department of Computer Science City University of Hong Kong
---------------------	--

Dr. LAI Richard	Department of Computer Science and Information Technology La Trobe University
-----------------	---

Acknowledgements

First of all, I would like to express my greatest sincere gratitude to my supervisor, Dr. KEUNG Wai Jacky, for his patience, guidance, financial support and encouragement during my Ph.D. pursuit. His practical research skills have been a blessing to me as I have been able to learn from his thinking and writing skills. I am also very grateful to him for allowing me the opportunity to travel for research internships and for introducing me to experienced software engineering researchers across the globe. These are priceless opportunities which will be with me for the rest of my life and have been key to the completion of my Ph.D. research studies. His guidance and invaluable advice helped me in all the time of research and writing of this thesis.

I would also like to thank my Ph.D. qualifying panel members, Prof. ZHANG Qingfu and Dr. LI Minming, not only for their precious time and insightful comments on my research reports, but also for the hard questions which inspired me to widen my research from various perspectives.

I am very grateful to my dissertation examiners, Dr. KWOK Lam For, Dr. YU Yuen Tak, and Dr. LAI Richard Y, for their invaluable time spent on carefully reading my thesis, and for providing me with precious suggestions and constructive comments to help improve the quality of this thesis.

My sincere thanks also go to my friends and lab-mates at City University of Hong Kong, Ms. Miao Zhang, Ms. Qing Mi, Mr. Kwabena Ebo Bennin, Mr.

Shuo Feng, and Mr. Xiupei Mei, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last three years.

Last but not least, I would like to thank my family for supporting me spiritually throughout writing this thesis. Without their precious understanding and encouragement, it would not be possible for me to complete my dissertation.

Contents

Abstract	i
Acknowledgements	vii
1 Introduction	1
1.1 Background	1
1.2 Problem Statement	3
1.2.1 Lexical Mismatch Problem	3
1.2.2 Semantic Information Omission	4
1.2.3 Structural Information Omission	5
1.3 Thesis Contributions	6
1.4 Thesis Organization	7
2 Literature Review	9
2.1 Bug Localization	9
2.1.1 Spectrum-based Techniques	10
2.1.2 IR-based Techniques	11
2.1.3 ML-based Techniques	14
2.1.4 DL-based Techniques	16
2.1.5 Additional Information Used in Bug Localization	17
2.2 Lexical Mismatch Problem	19

2.2.1	Lexical Mismatch Problem in Bug Localization	20
2.3	Deep Learning in Software Engineering	21
2.3.1	Recurrent Neural Network	24
2.3.2	Convolutional Neural Network	24
2.3.3	Cascade Forest	25
3	Bridging Lexical Gap for Bug Localization	27
3.1	Preliminaries	29
3.1.1	Recurrent Neural Network (RNN)	29
3.1.1.1	Long Short-Term Memory	30
3.2	BugTranslator	32
3.2.1	Model Design	32
3.2.1.1	Generating Training Instances	32
3.2.1.2	Attention-Based RNN Encoder-Decoder with LSTM Cells	33
3.2.1.3	Translation and Generation of Scores	35
3.2.2	Experiments	36
3.2.2.1	Preparation of Experiments	36
3.2.2.2	Experimental Results and Discussions	37
3.2.3	Discussion	38
3.3	Character-level Convolutional Neural Network and Recurrent Neu- ral Network	39
3.3.1	The Proposed Model	40
3.3.1.1	Data Preprocessing	40
3.3.1.2	Character-level CNN	41
3.3.1.3	RNN Encoder-Decoder	41
3.3.2	The Preliminary Results	41
3.4	Chapter Summary	42

4	Semantic Information Extraction for Bug Localization	45
4.1	Preliminaries	46
4.1.1	Term Frequency-Inverse Document Frequency (TF-IDF) . .	46
4.1.2	Word Embedding	48
4.1.3	Convolutional Neural Network (CNN)	50
4.1.4	Batch Normalization	51
4.2	Enhanced CNN	52
4.3	DeepLocator	54
4.3.1	Model Design	55
4.3.1.1	Data Preprocessing	56
4.3.1.2	Word Vectors Obtained by word2vec	57
4.3.1.3	Enhanced CNN Training and Prediction	57
4.3.2	Experiments	59
4.3.2.1	Experimental Settings and Evaluation Metrics . .	59
4.3.2.2	Research Questions	61
4.3.2.3	Results and Analyses	62
4.3.3	Discussion	65
4.3.3.1	Effectiveness of rTF-IDuF	65
4.3.3.2	Why does DeepLocator work?	67
4.3.3.3	Threats to Validity	68
4.4	DeepLoc	69
4.4.1	Model Design	70
4.4.1.1	Data Pre-processing	71
4.4.1.2	Word Vectors Generation	72
4.4.1.3	Feature Detecting	74
4.4.1.4	Enhanced CNN Training and Prediction	75
4.4.2	Experiments	77

4.4.2.1	Experimental Settings and Evaluation Metrics . . .	77
4.4.2.2	Research Questions	80
4.4.2.3	Results and Analyses	81
4.4.3	Discussion	87
4.4.4	Threats to Validity	89
4.4.4.1	Internal Validity	89
4.4.4.2	Construct Validity	90
4.4.4.3	External Validity	90
4.4.4.4	Statistical Conclusion Validity	91
4.5	Chapter Summary	91
5	Structural Information Extraction for Bug Localization	93
5.1	Preliminaries	93
5.1.1	Random Forest	93
5.2	CNN_Forest	95
5.2.1	Model Design	96
5.2.1.1	Data Preprocessing	97
5.2.1.2	Word Embedding	98
5.2.1.3	Feature Extraction	98
5.2.1.4	Cascade Forest	101
5.2.2	Experiments	103
5.2.2.1	Experimental Preparations	103
5.2.2.2	Results and Analyses	106
5.2.3	Discussion	109
5.2.3.1	Why does CNN_Forest Work?	109
5.2.3.2	Threats to Validity	110
5.3	Chapter Summary	112

CONTENTS

6	Conclusions and Future Work	113
6.1	Conclusions	113
6.2	Future Work	115
6.2.1	Future Work towards Proposed Techniques	116
6.2.2	Future Research towards Bug Localization	117
	Bibliography	119

List of Tables

1.1	Percentage of same words between bug reports and source files in subject projects.	4
3.1	Results of four models.	37
3.2	Results of two models.	42
4.1	Subject projects. [133]	60
4.2	Accuracy and time consumption on different number of layers. . . .	64
4.3	Accuracy of two models.	65
4.4	MAP and time consumption for different numbers of layers.	82
4.5	Prediction time (seconds) per bug report for three models.	83
4.6	Performance comparison.	85
5.1	Subject projects.	104
5.2	Performance comparison for intrinsic evaluation.	107
5.3	Performance comparison with the state-of-the-art techniques.	108
6.1	Performance Comparison between proposed models and the State-of-the-art Techniques.	115

List of Figures

3.1	Vanilla architecture of RNN.	29
3.2	Four main types of RNN architectures. T_x and T_y denote the length of the input and output.	30
3.3	Overall workflow of attention-based RNN Encoder-Decoder with LSTM cells.	33
3.4	The overview of the proposed model.	40
4.1	The architecture of the Skip-gram model.	48
4.2	The architecture of Sen2Vec [87].	49
4.3	The process of convolution and subsampling of a CNN.	50
4.4	The net structure of a CNN.	51
4.5	The overall workflow of DeepLocator.	56
4.6	The architecture of CNN for a sentence. n is the number of words in the sentence and k is the vector dimension of each word that is tuned experimentally (details are shown in Figure 4.7).	58
4.7	Performance of word2vec and random initialization along the dimension of word embedding (k).	62
4.8	Performance of different filter sizes.	62
4.9	Performance of different training set sizes.	66

4.10 Bug reports that contain "aspect" in Project AspectJ and Project Eclipse UI.	67
4.11 The overall workflow of DeepLoc.	71
4.12 An example of transforming descriptions in bug reports into vectors using Sent2Vec.	73
4.13 The architecture of feature detection.	76
4.14 The concatenation process.	77
4.15 Performance of different filter sizes.	81
4.16 Performance of number of filters.	81
4.17 MAP and MRR of conventional CNN and enhanced CNN for five projects.	83
4.18 Accuracy@k of the four models on Project SWT.	84
4.19 Visualization of the learned embeddings of 500 bug reports from Project SWT using t-SNE.	87
5.1 The illustration of random forest.	94
5.2 The overall workflow of CNN_Forest.	97
5.3 The feature extraction of bug reports using CNN.	99
5.4 The feature extraction of source files using ensemble of random forests.	100
5.5 Cascade forest to further extract features and learn the correlated relationships between bug reports and source files.	102

Chapter 1

Introduction

1.1 Background

Software bugs are coding mistakes that tend to induce unexpected or abnormal behaviors in a software project [13]. Once discovered, these bugs are described in bug reports or issue reports submitted by developers, testers, or users of the project system. A bug report provides information about the scenarios in which the software does not behave as expected and how to reproduce this abnormal behavior [58]. Bug reports are the main source of information for developers to understand and analyze bugs. When a bug report is received and assigned to a developer for fixing, the developer needs to investigate the contents in the bug report and then search through the source files to locate the potential buggy files that are relevant to the bug. This process is called bug localization [84].

Bug localization is a significant task during software maintenance. To locate a newly reported bug, developers must carefully analyze the bug report and review numerous source code files. Thus, bug resolution activity usually requires considerable time and effort [58] to improve software quality and ensure software integrity [13]. However, manually localizing buggy files is painstaking for devel-

opers if there are tens of thousands of source files. To reduce maintenance costs and substantially improve the efficiency and productivity of the whole software team, researchers have proposed several automated bug localization approaches and tools for localization of potential buggy files, especially when the project is large and complex and involves thousands of source files.

The existing bug localization techniques can be categorized into four main groups. The first group comprises approaches that use traditional features related to program analysis information [46], such as passing or failing execution information with test cases. Spectrum-based fault localization techniques [67, 1] are typical examples of these approaches. These methods extract static features from the source code or execution information, which is a time-consuming process. The second group comprises the information retrieval (IR)-based approaches, which search and rank buggy files for a given bug report. These approaches measure the textual similarity between bug reports and the names of classes or methods in source code files. This kind of similarity focuses mainly on the term weights as used in the IR field [133]. In addition, machine learning (ML)-based approaches have been proposed. These approaches adopt ML models that are trained to match the topics of bug reports with those of source files [84] or classify source files into multiple classes using formerly fixed files [50]. The last group is based on deep learning techniques [56, 57]. However, the recently proposed deep learning based model for bug localization combines deep neural networks (DNNs) and IR techniques. The final results are influenced by the performance of IR to some extent. Their experiments showed that only using DNN achieved poor performance and most improvements still benefited from IR techniques, which implied that DNN in their model was a subsidiary.

1.2 Problem Statement

1.2.1 Lexical Mismatch Problem

The lexical mismatch problem lies in the different expressions with similar or even same meanings. Significantly, a machine translation-based technique trained on the pairs of API annotations and corresponding API sequences is able to generate API sequences given a natural language query similar to the annotations [32]. A natural language model is applied in this paper to recognize deep semantics of the API sequences and natural language queries by considering sequences of words and APIs instead of bags-of-words used in existing information retrieval models [12, 91].

Bug reports are written in natural language, whereas source files are represented by code tokens. The differences between them in expression and representation lead to a lexical mismatch problem, which stifles the effectiveness and accuracy of existing bug localization techniques in detecting buggy files [57, 133, 134]. To explicitly show the mismatch degree between natural languages in bug reports and code tokens in source files, the following formulation is used to calculate the percentage of same words:

$$\frac{N_{same}}{N_B + N_S - N_{same}} \quad (1.1)$$

where N_{same} represents the number of same words that are in both bug reports and source files, N_B denotes the number of words in bug reports and N_S stands for the number of words in source files. As shown in Table 1.1, the average percentage of the same words is 5.1% - 6.9%, which indicates that bug reports and source files share few words with each other. Even though the maximum percentages of same words are 45.0% - 69.0% that seem very high, there are only 2 pairs of bug reports and corresponding buggy files in Project Eclipse UI whose percentages are higher than 40.0%, similarly, 3 in Project JDT whose percentages are higher than 50.0%

and only 1 in Project SWT whose percentages are higher than 40.0%. Significantly, there are 113 - 654 pairs whose percentage is 0.0%, which is a really high number. This numeric evidence indicates the importance of bridging the lexical gap for bug localization.

Table 1.1: Percentage of same words between bug reports and source files in subject projects.

Project	Time Range	Min % of Same words	Max % of Same words	Avg. % of Same words
Eclipse UI	10/2001-01/2014	0.0%	45.0%	6.3%
JDT	10/2001-01/2014	0.0%	69.0%	6.9%
SWT	02/2002-01/2014	0.0%	49.0%	5.1%

To improve the accuracy of bug localization, recent techniques [57, 133] include the similarity between bug reports and application programming interface (API) entities (class and interface names) to bridge the lexical gap. Ye et al. [134] applied word embedding to obtain word vectors of bug reports and source code in a shared representation space. The existing approaches regard the code tokens in source files as the same natural languages used in bug reports, which fails to effectively suppress the effects of lexical mismatch on bug localization. We thus propose approaches to distinguish bug reports and source code into different symbolic classes and then extract deep semantic similarity and relevance between bug reports and the corresponding buggy files to bridge the lexical gap at its source, thereby further improving the performance of bug localization.

1.2.2 Semantic Information Omission

The approaches in the aforementioned second and third groups (Section 1.1) focus on the term weights of natural language texts and do not consider source code semantics during the bug localization process. These bug localization techniques

1.2. PROBLEM STATEMENT

transform the terms in bug reports and source files into textual representations and adopt some models to measure the textual similarity between them. For example, Latent Dirichlet Allocation (LDA)-based techniques [10, 69] were used to represent bug reports and source files, and BugLocator proposed by Zhou et al. [145] applied revised Vector Space Model (VSM) [28] to calculate the textual similarity. These techniques focused more on textual similarity than the semantics of the terms in bug reports and source files.

According to prior research, semantic information is important for code suggestion [83] and code completion [35] and is also useful for bug localization [134]. To bridge the semantic gap and the lexical gap, deep learning has been introduced into the domain of bug localization for semantic parsing. Deep learning is known to have excellent performance in natural language processing (NLP) and image processing [21, 55]. The recently proposed deep learning-based model for bug localization combines deep neural networks (DNNs) and IR techniques [56, 57]. Its results are influenced to some extent by the performance of IR techniques. Most importantly, many DNN models are used together, which makes it very complex and difficult to accurately adjust the parameters of the model. Therefore, this thesis introduces new approaches to extract the underlying semantic information from bug reports and source code.

1.2.3 Structural Information Omission

Current state-of-the-art techniques have acknowledged the lexical mismatch problem between the texts in bug reports and code tokens in source files caused by the ignorance of the semantic information in them. Several studies [56, 57, 133, 134] try to bridge the lexical gap by including semantic information of the bug reports and source files. Ye et al. [134] applied word embedding techniques to obtain vectors of words in bug reports and source files and measured the similarity between

them. Lam et al. [56, 57] employed deep neural networks to correlate the frequently occurred terms in bug reports and source files. These techniques considered the semantics but inadvertently ignored the structural information of both bug reports and source files. NP-CNN proposed by Huo et al. [39] employed convolutional neural networks (CNNs) to extract structural features from both bug reports and source files and used a fully-connected network to fuse features. However, the model did not consider the semantic information in the bug reports and source files.

The existing challenge is not only how to make effective use of the semantic information but also how to extract the structural information from bug reports and source files. Additionally, there are obvious differences between natural languages and programming languages [88] and these differences can stifle the localization process. Compared to natural languages, programs contain more stringent structural information [35, 81]. Therefore, we propose to use the techniques based on tree-structure that have shown excellent performance on extracting the structural and syntactic information in dealing with source code [65, 81, 83, 121].

1.3 Thesis Contributions

In this thesis, we investigate three problems lying in the field of bug localization as described in Section 1.2. Several models are built to solve the aforementioned problems and corresponding evaluations are conducted to analyze the problems as well as the performance of proposed models. We provide several contributions to the field of bug localization. The key contributions are highlighted as follows:

- We investigate the impact of three main problems on the performance of bug localization. Firstly, how to bridge the lexical gap caused by the different expressions of bug reports and source files? Secondly, how to preserve the

semantic information of both bug reports and source files? Thirdly, besides semantics, how to extract structural information in source code?

- We introduce a machine translation-based technique to the area of bug localization. BugTranslator and an enhanced version based on recurrent neural network encoder-decoder architectures are proposed to bridge the lexical gap between bug reports and source files.
- We empirically demonstrate that the word embedding techniques can help preserve the semantic information and CNN is good at semantic parsing in bug reports and source files. An enhanced CNN is proposed to include the important bug-fixing experience (bug-fixing recency and frequency) besides the semantic information that can be extracted by conventional CNN from bug reports and source files. Meanwhile, two novel deep learning models (DeepLocator and DeepLoc) based on the proposed enhanced CNN are built for bug localization.
- We leverage the ensemble of random forests to detect the structural information from the source code. Two different techniques (CNN with multiple filters and an ensemble of random forests with multi-grained scanning) are applied to extract features respectively from bug reports and source files according to the differences between natural languages and programming languages.

1.4 Thesis Organization

The thesis is structured as follows. Chapter 2 summarizes several related works of bug localization, the lexical mismatch problem, deep learning in software engineering, and other related techniques, and extracts main problems with existing

studies. The lexical mismatch problem in bug localization is discussed in Chapter 3, along with two models (BugTranslator and an enhanced version) proposed to bridge the lexical gap between bug reports and source files. In Chapter 4, an enhanced CNN is proposed to include the important bug-fixing experience to further strengthen the ability of CNN for bug localization. Two models (DeepLocator and DeepLoc) are then described to extract semantic information from bug reports and source files, and the corresponding experimental results and discussions are presented. Chapter 5 discusses CNN_Forest to extract both semantic and structural information from bug reports and source files, along with the experimental results and discussions. We conclude the thesis in Chapter 6 with a discussion of findings from all the experiments conducted from the previous chapters and highlight some promising future directions of localizing bugs.

Chapter 2

Literature Review

In this chapter, we first review the existing bug localization techniques given bug reports and then conclude other related information, including code structure, similar bug reports, code version history, stack traces, and so on, used in the existing techniques. The lexical mismatch problem, especially in bug localization, is subsequently discussed in Section 2.2. We also survey several deep learning techniques used in software engineering and other studies related to the main techniques used in this thesis.

2.1 Bug Localization

Software systems are often plagued with bugs, which are a serious concern for developers and maintainers [115]. To improve the reliability and stability of systems, developers often allow testers or users to submit bug reports to bug tracking systems. For each of these reports, developers need to locate the code units that need to be modified to fix bugs, which is called the bug localization that can be a tedious task in large-scale software projects having a large code base with thousands or even millions of files [69]. Bug localization task is a crucial process in

the development of software systems, while it is time- and effort-consuming. It is widely known that the later a failure is detected, the higher the cost to fix it. Therefore, effective methods for automatically locating bugs from bug reports are highly desirable, as they would shorten the bug-fixing time, consequently reducing software maintenance costs. Given the ever-increasing size and complexity of software projects, manual bug localization is a lengthy and challenging task, which motivates the research for automated bug localization techniques. There are several automated bug localization techniques proposed in the literature to help developers localize buggy files given bug reports. In this section, we review the existing automated bug localization techniques. The commonly used techniques are either based on spectrum methods, information retrieval (IR) methods, machine learning (ML) methods, or deep learning (DL) methods. Additional related information besides bug reports is concluded subsequently.

2.1.1 Spectrum-based Techniques

Spectrum-based bug localization techniques analyze passed and failed runtime execution traces of source code files mostly given test cases. These approaches instrument source code to record the execution information when running source files. The counts of each program element (e.g., statements, branches, methods or classes) are then used to formulate different suspiciousness scores, such as Tarantula [46], Ample [23], Ochiai [2], and MULTRIC [131]. Their basic idea is that a program element is considered to be more suspicious if it appears more frequently in failed executions than in passed ones. Given that spectrum-based techniques are costly and lack scalability [113], most researchers recently combine spectrum-based methods with other techniques, such as IR-based techniques, to overcome the limitation of each individual method (e.g., lack of test cases or bug reports) and further improve the accuracy of bug localization [60, 36]. In this thesis, we

mainly focus on how to improve the accuracy of bug localization given bug reports.

2.1.2 IR-based Techniques

IR-based techniques formulate information tokens extracted from a given requirement (e.g., feature description, bug report, etc.) as a query to be matched in a document search space built from the collections of source files and indexed through code tokens (e.g., files, classes, functions, or methods) extracted from source code artifacts to retrieve a list of code documents relevant to the query. The IR-based techniques then rank the source code documents based on the relevance to a query that is measured as a textual similarity score. The higher the similarity score, the more likely the document is to contain the buggy code.

The following studies focused on the IR techniques themselves on the task of bug localization. Chaparro et al. [15] reformulated queries that contain only terms describing observed behaviors by manually reducing noisy and ineffective queries, and empirically demonstrated that combining the bug report title and the observed behavior perform best across various IR-based techniques. Khatiwada et al. [49] proposed a new paradigm of information-theoretic IR methods, including Pointwise Mutual Information and Normalized Google Distance, and investigated their performance in bug localization tasks.

We then discuss several methods frequently-used in the IR domain for the bug localization tasks. Gay et al. [28] transformed bug reports and source files into feature vectors based on Vector Space Model (VSM) and then measured the similarity between them. Zhou et al. [145] proposed BugLocator, which was based on a revised Vector Space Model (rVSM), to measure the textual similarity between bug reports and source files using information from similar bugs that had previously been fixed. The weighted sum of two rankings (ranking based on source code and ranking based on similar bugs) was used to locate the relevant files for a

bug.

Scanniello et al. [96, 97] and Zhang et al. [142, 143] used both textual information from source code and bug reports and syntactic information from source code for concept location. Since BugLocator was a famous technique that attempted to rank relevant source files for bug reports, Wong et al. [124] proposed segmentation and stack-trace analysis techniques, namely Brtracer, on top of BugLocator [145], to boost fault localization. They divided a source code file into a series of code snippets and used the code snippet that was most similar to the bug report to represent the source code file. They then analyzed the similarity between the stack traces in the bug report and the source code file. Saha et al. [95] regarded the summary and description of bug reports as two different query fields, combined respectively with classes, methods, variables, and comments extracted from source files. However, these eight features were considered equally important. Moreover, they used the fixed version of the project in the evaluation, which led to inconsistency when considering future bug-fixing information.

To address the issues mentioned above, Ye et al. [133] developed an adaptive ranking model supported by a learning-to-rank (LR) technique to rank relevant files for bug reports. Six features were used in this paper: surface lexical similarity, application programming interface (API)-enriched lexical similarity, collaborative filtering score, class name similarity, bug-fixing recency, and bug-fixing frequency. In the literature [134], LR+WE was proposed to enhance LR, in which bug reports and source code were represented by word embeddings (WE) and the similarities between them were added as additional features. The weight of each feature was then trained using the LR technique based on the previously fixed bugs. However, this model used a hill-climbing algorithm with a linear weighted sum of features, which might ignore nonlinear relationships. These IR-based techniques rely on the textual similarities between bug reports and source files, which do not bridge the

2.1. BUG LOCALIZATION

semantic gap.

Amalgam [114] put together version history, similar bug reports, and code structure by integrating bug prediction algorithm of Google [63] to analyze source code version history, BugLocator proposed by Zhou et al. [145] to analyze similar bug reports that had previously been fixed in the bug reports system, and BLUiR proposed by Saha et al. [95] to analyze the structured information of the source code. Version history information used in Amalgam was built on the intuition that bugs are likely to occur again in source files that have been fixed many times. On top of the aforementioned three features (source code version history, similar bug reports, and structured information), two additional information including stack traces and reporter information was included by Amalgam+ [115] for bug localization, which gained 6% improvement. However, these sources of information were composed by a weighted sum that didn't consider non-linear relationships between them.

Hoang et al. [36] proposed a multi-modal technique considering both bug reports and methods to localize bugs by tuning various weights learned from a set of training bug reports relevant to the new report and program spectra. This study also incorporated the network Lasso regularization on both bug report and method similarity networks to enforce similar bug reports (and methods) to have similar (or even identical) model parameters. Their approach created a bug-specific model to map a particular bug to its possible location and introduced the concept of suspicious words associated with a specific bug. A method level bug localization using minimized search space (MBuM) proposed by Rahman et al [93] identified buggy locations by discarding irrelevant source code. It extracted the relevant search space for a bug by using the execution trace of the source code and then used the modified VSM to measure the textual similarity between the bug report and the source code methods. However, in most cases, it is very difficult to reproduce

the bug and trace all the executed methods by running the whole software system, which might not be commonly-used at any stage of software development and maintenance.

Zhang et al. [143] proposed a method called MULAB for multi-abstraction concern localization, which represented a code unit and a textual description at multiple abstraction levels. The MULAB approach leveraged multiple topic models to capture representations of documents at different abstraction levels by employing an adaptive LDA with genetic algorithm (LDA-GA) to infer semi-optimal topic model configuration. These representations were then used to compute the similarity between bug reports and source code units by VSM at multiple abstraction levels. Afterward, a data fusion method called COMBMNZ-DEF was applied to fuse the data from the bug tracking system and the source code version control system. Youm et al. [136] proposed BLIA for file-level and method-level software bug localization. The BLIA approach integrated the information from texts, stack traces, and comments in bug reports, structured information of source files, and the source code change history. The ranked suspicious files were obtained from the weighted sum of the above integrated information, whose scores were combined with the similarity between methods in the files and bug reports to rank the suspicious methods. This paper also used structural parts to improve the method-level bug location, which is of limited capacity to improve the performance.

2.1.3 ML-based Techniques

Kim et al. [50] applied a Naive Bayes classifier to build a two-phase recommendation model. Phase 1 was used to decide whether the bug was predictable or not. If not, there was no further prediction behavior. If it was predictable, Phase 2 applied a Bayes model to recommend a set of files to fix. However, the model mainly focused on the names of the fixed files. Thus, it was difficult for

2.1. BUG LOCALIZATION

the model to recommend a file that had never been fixed before. Moser et al. [80] built a model based on three common machine learners to find defective files from Eclipse projects considering code changes. Thomas et al. [107] empirically studied the impact of classifier configurations, including bug report representation, source code entity representations, preprocessing steps, IR model parameters and entity metrics, in bug localization task to investigate the performance of the underlying IR-based methods and showed that the classifier combination could help improve bug localization performance. Markov logic was used in [139] by combining statement coverage, static program structure information, and prior bug information.

Topic-based models were used to extract topics for many tasks in the field of software engineering [85, 130]. Poshyvanyk et al. [90] applied Latent Semantic Indexing (LSI) to represent code and queries as vectors, and estimated the similarity between their vector representations using the cosine similarity. Their experiments on Mozilla Firefox project demonstrated that PROMESIR could significantly improve the performance of bug localization in comparison with the methods using either LSI or probability retrieval model independently. Lukins et al. [69] proposed a Latent Dirichlet Allocation (LDA)-based approach that measured the similarity between the descriptions of bug reports and the topics of source files estimated by LDA. The STMLocator [117] was built on top of LDA for supervised topic modeling considering word co-occurrence and the phenomenon of longer source files tend to contain more bugs. However, these approaches were not automatic because of the tuning process of the right number of topics. BugSout, a topic-based machine learning model proposed by Nguyen et al. [84], used an extended LDA [10]. The bugs were related to their corresponding buggy files by their shared topics. However, a tuning process was needed to find the right number of topics for different projects, which meant that the model was not automated. In addition, there was a strong assumption in this paper that the textual contents of bug reports shared

some technical aspects with the textual contents of the corresponding buggy files. However, natural language texts used in bug reports likely differ from code tokens in the source code files of real-world projects, which implies that this assumption is insufficient. Moreover, LSI and LDA-based techniques have been experimentally validated worse than the Vector Space Model-based techniques [10, 69].

2.1.4 DL-based Techniques

Deep learning is a relatively new concept, and there is little prior work about deep learning-based bug localization. Lam et al. [56, 57] built a model named HyLoc that combined deep learning with the IR technique. About six deep neural networks (DNNs) were used in this model: two for feature extraction, two for projection, one for relevancy estimation, and one for feature combination. However, using so many kinds of DNNs posed a risk of complex weight adjustment and high training and learning costs. Their experiments showed that using DNNs alone achieved poor performance and that most improvements still benefited from IR techniques, which implied that DNNs in their model was a subsidiary. Moreover, the approach did not consider the structural information of the bug reports and source files. Huo et al. [39] proposed NP-CNN, a convolutional neural network (CNN)-based model, to learn the unified features from bug reports and source files written in natural and programming languages. In their work, CNN was employed to extract features from both bug reports and source files. Different from their model, our proposed CNN_Forest that will be discussed in Chapter 5 adopted CNN with multiple filters and the ensemble of random forests with multi-grained scanning to respectively extract structural information from bug reports and source files. The experimental results validated that this different feature extraction strategies for bug reports and source files indeed make sense. Furthermore, we employed word embedding techniques to preserve the semantics

of feature vectors before feeding them into the CNN and random forests, which was not included in NP-CNN.

Previous studies have proposed several features to represent the program source code. However, few methods try to extract both semantic and structural information from bug reports and source files. Deep learning can help to bridge this gap.

2.1.5 Additional Information Used in Bug Localization

In existing bug localization techniques, there is additional information related to bug reports and source code that is leveraged to adjust the ranking of relevant code files, including stack traces [79, 124, 115, 136, 101], code structure [114, 115, 95, 136, 3, 106], code version history [102, 114, 115, 136, 120], similar bug reports [145, 114, 115, 95, 136, 25, 124, 94], or combinations of the above [114, 136, 95, 124, 99, 24].

Stack traces as promising information in bug localization are sometimes included in bug reports, which are also used to alter the text-based ranking. Wong et al. [124] proposed a Brtracer which further considered stack traces in addition to SimiScore [145]. Moreno et al. [79] proposed Lobster to combine the stack traces and textual contents in bug reports to match code elements in source code files for bug localization using the system’s call graph. A traceability information graph [94] was built to calculate trace score incorporating requirements artifacts. These techniques are mainly based on an assumption that the stack traces may directly reference the classes or methods of buggy code files [79, 124, 115, 136, 101]. The set of suspicious classes/files is expanded by identifying artifacts (in)directly referenced in the code of the ones found in the stack trace.

The information of code structure is often combined with other features to boost the bug localization process. Youm et al. [136] combined code structural

information with other scores to improve the accuracy of bug localization. Takahashi et al. [106] presented a preliminary study using code smells to improve bug localization performance. Code smells were detected along with their severity to form a severity score that was combined with the textual similarity while ranking code documents.

Software history information is used to enhance the accuracy of bug localization mainly based on the version control records. Two main history information leveraged in the existing bug localization techniques is bug-fixing recency [102, 115, 136, 120] and frequency [102, 114, 120]. They imply that source files fixed recently are more likely to be buggy than those fixed long ago (bug-fixing recency) and that source files that have been fixed many times are more likely to still contain bugs than those seldom fixed or even never fixed (bug-fixing frequency). The bug fixing history is also used to complement the textual similarity. A set of previously-fixed bug reports is kept, along with its corresponding fixed buggy-set: the set of code documents that were modified in order to fix each bug. Given a bug report, it is compared to each previously-fixed bug report. The documents in each fixed buggy-set are boosted according to the textual similarity of the fixed bug with the bug report [145, 114, 115, 95, 136, 25, 124, 133, 134].

In addition, several approaches use the combination of the above information or even existing techniques to improve the performance of bug localization. Amalgam [114] took advantage of the bug prediction algorithm of Google [63] to analyze source code version history, BugLocator [145] to analyze similar bug reports that had previously been fixed in the bug reports system, and BLUiR [95] to analyze the structured information of the source code. BILA [135] put together source code entities [95], SimiScore [145], stack trace score [124], and version history score to improve bug localization performance. Locus [120] incorporated token, code entity, and history scores to compute the final similarity score by using commit

logs as well as change hunks in revision history to improve similarity measures. FineLocator [140] expanded query scores by semantic similarity score, temporal proximity score and call dependency score to localize bugs in method-level. The representation of a method with short length was augmented by elements of its neighboring methods with query expansion. FineLocator retrieved the methods in source code by similarity ranking on the bug report and the augmented methods for bug localization.

2.2 Lexical Mismatch Problem

The lexical mismatch means that a similar meaning can be expressed by different vocabulary or languages. The lexical gap exists in several areas, such as machine translation [76], community question answering [138, 144] and so on. To solve the lexical mismatch problem, several techniques have been proposed, such as topic-based model [138], continuous word embedding representations-based methods [76], and translation-based models [144].

In the field of community question answering, probabilistic topic models have been proposed to learn the latent topics aligned across the question-answer pairs to bridge the lexical gap, on the assumption that question and its answer should share a similar topic distribution [14, 44]. However, questions and answers are heterogeneous in many aspects, they do not share the same topic distribution in practice. The word embedding-based methods have been proved to be efficient and simple against other remaining methods. However, they still suffer from the polysemous problem that means a same word has different semantic meanings and word order problem. Recently, some methods have been proposed to handle the polysemous problem [82, 16]. Neelakantan et al. [82] focused on the creation of sense specific embedding. Cheng et al. [16] introduced the concept information to

handle this problem, which obtained better results against other methods. Besides, other methods [37, 59, 98] have been proposed to obtain the sentence or query embedding representation directly so that the syntactic information can be kept. Much importance has been attached to statistical translation in the literature. The IBM translation model was used [43, 132] to learn the translation probabilities between question and answer words. Apart from word-level translation, phrase-level translation for question and answer retrieval has also achieved promising results [14]. Experimental results show that translation models obtain state-of-the-art performance for question retrieval in community question answering.

2.2.1 Lexical Mismatch Problem in Bug Localization

In the field of bug localization, bug reports and source code represent similar operations with different expressions. This lexical mismatch challenge also limits the performance of existing bug localization techniques [57, 133].

Lam et al. [56] attempted to bridge the lexical gap by combining deep neural networks (DNNs) with IR technique to localize buggy files for bug reports. They trained DNNs to relate the terms in bug reports and tokens in source files if they appeared frequently enough in pairs during training phrase. DNNs in their model were regarded as black boxes and they didn't extract semantic information from bug reports and source files. What's more, their experimental results showed that DNNs without the combination of IR techniques gained very poor performance. Word embeddings were used by Ye et al. [134] to obtain document similarities as two new features added into their previously proposed linear ranking model [133]. The natural language in bug reports and code snippets in source files were projected by word embeddings into vectors based on the hypothesis [34] that words prone to have similar meanings if they appear in the same context. Their model contained the semantic similarity between the two bags-of-words of bug reports

and source code. But the model only considered the information of API documentations that supported more general tasks without fully utilizing the information of the project itself. And if the pairs of words never appear in same context, word embeddings have low possibility to give them similar word vectors, which also limited the performance of the model.

Significantly, the approaches that attempted to bridge the lexical gap were experimentally validated to outperform those that ignored the lexical mismatch, which also revealed the existing challenge caused by lexical mismatch.

2.3 Deep Learning in Software Engineering

In recent years, many deep learning techniques have been introduced to software engineering research, such as code suggestion [122, 83], code summarization [4, 41], code generation [66], API suggestion [32], defect prediction [116], effort estimation [18], test deep learning systems [108, 70, 33], program classification [81], bug localization [128, 126], and so on.

White et al. [122] proposed a deep architecture to model software language specific to sequential data and suggested many avenues for future work, especially for code suggestion. Besides, GraLan [83] was proposed to develop an API suggestion engine, which was a graph-based statistical language model learned from a source code corpus that computed the appearance probabilities of usage graphs given the observed (sub)graphs. A convolutional attention network employing convolution on the input tokens to detect local time-invariant and long-range topical attention features in a context-dependent way [4], and CODE-NN using long short-term memory (LSTM) networks with attention to produce sentences that describe C# and SQL code snippets [41] were built to summarize source code.

A latent prediction network was proposed as a generative model for code gen-

eration [66]. It combined a character level softmax to generate language-specific tokens and multiple pointer networks to copy keywords from the input, along with attention-based neural models to attend over structured inputs and a code compression method to reduce the size of the code. A novel tree-based convolutional neural network (TBCNN) was built by Mou et al. [81] for programming language processing. Wang et al. [116] used a deep belief network to detect features from tokens extracted from an AST of source code to find defective files, but it was insufficient for mapping the tokens to continuous integers.

There are several studies on testing deep neural network. Tian et al. proposed DeepTest [108], a systematic testing tool specific for deep neural network (DNN)-driven vehicles to detect errors. It leveraged real-world changes in driving conditions to automatically generate test cases that maximized the numbers of activated neurons. The authors demonstrated that different realistic image transformations (e.g., scaling, shearing, and rotation) are useful to detect defects in DNN-driven autonomous cars. Ma et al. [70] proposed DeepGauge, a set of coverage criteria for DNNs to effectively measure testing adequacy. The testing criteria lie in two main levels: neuron-level and layer-level. More specifically, they defined neuron-level coverage criteria for k-multisection neuron coverage, neuron boundary coverage and strong neuron activation coverage, and layer-level coverage criteria for top-k neuron coverage and top-k neuron patterns. These criteria monitor and gauge the neuron activities and intrinsic network connectivity at various levels of granularity, which is a tremendous process considering the large number of neurons and huge layers in most DNNs. DLFuzz proposed by Guo et al. [33] used differential fuzz testing to maximize neuron coverage by continually minutely mutating the input, and generating more adversarial inputs for a given DL system. To cover more logic, DLFuzz needs to iteratively select neurons worthy to activate. Besides communicating with thousands of neurons, it applied minute perturbations on the

2.3. DEEP LEARNING IN SOFTWARE ENGINEERING

test inputs and kept the mutated inputs that increased the neuron coverage for subsequent fuzzing. The prediction results before and after the mutation were ensured to be the same by the minute perturbation. If the prediction result of the mutated input was not the same as the original input, then an error triggered.

Choetkiertikul et al. [18] proposed a deep learning-based prediction system for estimating story points based on Long Short-Term Memory (to learn a vector representation for issue reports) and Recurrent Highway Network (to build a deep representation). DeepAPI was proposed by Gu et al. [32] to find API usage sequences given an API-related natural language query. The model adopted the attention-based recurrent neural network (RNN) Encoder-Decoder model considering API importance by using inverse document frequency-based weighting to train API usage sequences and their corresponding annotations. The training time of this model was very long, which is also a drawback of RNN. Gu et al. [31] built an embedding neural network to jointly transform code snippets and natural language descriptions into vectors. With the vector representations, the code snippets could be searched out based on similarity given a natural language query.

In the software engineering field, many files are written in natural languages, such as bug reports, API descriptions, and annotations. Even source codes themselves are similar to natural language to some extent. Thus, many mature techniques used in natural language processing can be applied to some issues in software engineering to extract semantic information.

In the next subsections, we will discuss in detail the literature of the main deep learning-related techniques used in this thesis, including recurrent neural network, convolutional neural network, and cascade forest.

2.3.1 Recurrent Neural Network

Deep learning has shown great success in the area of machine translation. Recurrent neural network (RNN) language model proposed by Mikolov et al. [77] re-scored the n-best lists given by previous statistical machine translation (SMT) system to improve the quality of translation. Cho et al. [17] proposed an RNN Encoder-Decoder model consisting of two RNNs to encode sentences into vectors and then decode. The conditional probabilities of a target phrase given a source phrase computed by RNN Encoder-Decoder were integrated into the existing SMT system. Sutskever et al. [104] used RNN with long short-term memory (LSTM) units for machine translation and achieved comparable results with the conventional SMT system. Moreover, the LSTM learned different sentence representations that were sensitive to the order of words. Aforementioned Encoder-Decoders encoded the source sentences into a fixed-length vector and then decoded this vector into the target sentences, which was experimentally revealed by Bahdanau et al. [6] to be a limitation of basic RNN Encoder-Decoder model. Their proposed enhanced architecture could especially improve the performance on long sentences, which will be adopted in Chapter 3.

2.3.2 Convolutional Neural Network

Convolutional neural network (CNN) has excellent performance in the field of image processing [54, 105] and natural language processing [20, 21]. Kim [51] conducted a series of experiments for sentence classification tasks using CNN with the pre-trained word vectors obtained by the word embedding technique and achieved excellent results on several benchmarks. Johnson et al. [45] studied CNN on text categorization (sentiment classification and topic classification) to make use of the internal structure of text data. Besides NP-CNN discussed previously for

bug localization, Mou et al. [81] proposed a CNN over tree structures for programming language processing in the field of software engineering. This model contained vector representation layer, coding layer, tree-based convolutional layer whose convolution kernel was designed over programs' abstract syntax tree and dynamic pooling layer, as well as a fully-connected hidden layer to detect structural information from programs. It was superior in terms of program classification and code pattern detection.

2.3.3 Cascade Forest

The cascade forest is a cascade of ensembles of random forests that are the ensemble of decision trees. The ensemble methods combine multiple learners and are powerful on many tasks [146]. The cascade procedure is able to automatically determine the number of cascade levels and thus have a control on its model complexity, which is related to boosting procedure [26, 119, 112]. Zhou et al. [147] proposed deep forest based on the cascade structure for many tasks including image categorization, face recognition, sentence classification, and achieved comparable or even better results. They analyzed that this cascade structure behaved like the layer structure in deep learning. Inspired by their work, we adopted the cascade forest in Chapter 5 to further extract the structural information from bug reports and source files, and learn the correlated relationships between them.

Chapter 3

Bridging Lexical Gap for Bug Localization

The high cost of manual *bug localization*, especially for large software systems, has instigated the design of automated techniques to help developers prioritize and focus on potentially buggy files based on bug reports. However, bug reports are written in natural language, whereas source files are represented by code tokens. The differences between them in expression and representation lead to a *lexical mismatch* problem, which stifles the effectiveness and accuracy of proposed bug localization techniques in detecting buggy files [57, 133, 134].

To improve the accuracy of bug localization, recent techniques [57, 133] include the similarity between bug reports and application programming interface (API) entities (class and interface names) to bridge the lexical gap. Ye et al. [134] applied word embedding (WE) to obtain word vectors of bug reports and source code in a shared representation space. These approaches regard the code tokens in source files as the same natural languages used in bug reports, which fails to effectively suppress the effects of lexical mismatch on bug localization.

To address the above issue of lexical mismatch and thereby further improve

the performance of bug localization, we distinguish bug reports and source files into different symbolic classes and formulate the bug localization problem as a machine translation problem. For example, during the machine translation process of an English sentence into a French sentence, the two sentences are represented in different languages (symbols), but they represent similar meaning. Likewise, the pairs of API description and API sequence denote similar operations by different representations. Significantly, a machine translation technique achieves outstanding performance in the generation of API sequences given a natural language query [32]. Motivated by this, we propose two novel bug localization models, BugTranslator and an enhanced version, based on a recurrent neural network (RNN) Encoder-Decoder with long short-term memory (LSTM) cells by absorbing useful modules from famous machine translation models [6, 17, 104].

The main contributions of this chapter are:

- To the best of our knowledge, we are the first to introduce the machine translation technique to the area of bug localization and to propose a novel method of bridging the lexical gap radically.
- Empirically validate the effectiveness of BugTranslator and the enhanced version using character-level Convolutional Neural Network in overcoming the lexical mismatch challenge.

BugTranslator [126] and the enhanced version [125] have been published in Information & Software Technology and APSEC 2018 respectively. The remainder of the chapter is structured as follows. Section 3.1 discusses preliminaries for this chapter. Two models based on RNN Encoder-Decoder architecture are presented in Section 3.2 and 3.3. Finally, this chapter is summarized in the last section.

3.1 Preliminaries

3.1.1 Recurrent Neural Network (RNN)

In practice, when predicting the content of the next frames in the video, it is obvious that the former input content is correlated with the prediction output. For these scenarios with context features, the used net must have memory capacity. Compared with the feed-forward neural network, there is a closed loop added in the hidden layers of RNN [74] that enables it have memory capacity, just as Figure 3.1. Given an input sequence $x_1, \dots, x_{t-1}, x_t, x_{t+1}, \dots, x_T$, in a language model, x_t represents a word, the entire sequence means a sentence, the hidden state is $h_1, \dots, h_t, \dots, h_T$, the output sequence is $y_1, \dots, y_t, \dots, y_T$. W_{hx}, W_{hh}, W_{yh} represent the weight matrices between input layer and hidden layer, hidden layer and hidden layer, output layer and hidden layer.

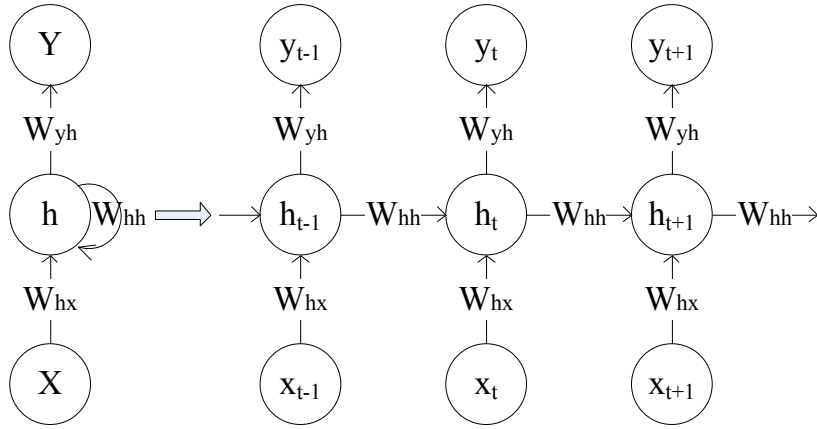


Figure 3.1: Vanilla architecture of RNN.

There are four main types of RNN architectures that are one-to-one, one-to-many, many-to-one and many-to-many respectively. The blank boxes in Figure 3.2 represent different cells (e.g., Long Short-Term Memory (LSTM) [29] and Gated Recurrent Unit (GRU) [19]) used in RNNs. This thesis designed a variant of

the many-to-many type for bug localization. The details will be introduced in subsequent sections.

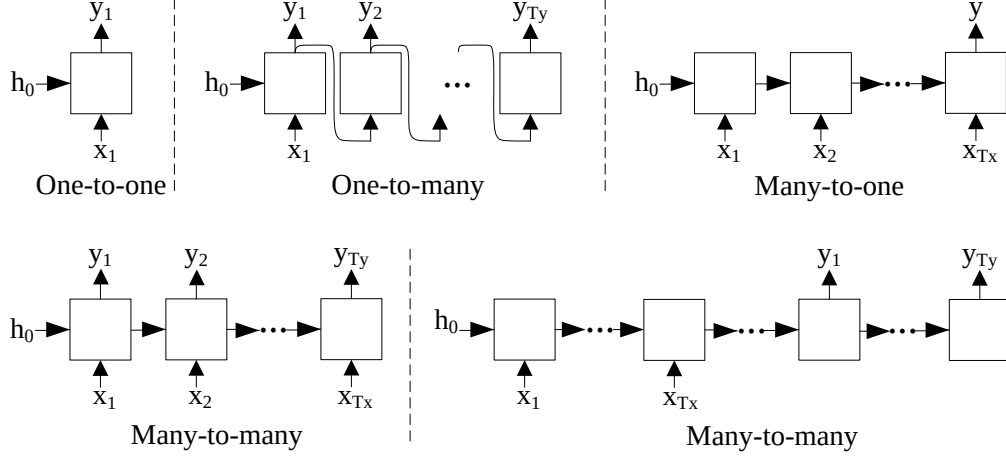


Figure 3.2: Four main types of RNN architectures. T_x and T_y denote the length of the input and output.

3.1.1.1 Long Short-Term Memory

More recently, deep RNNs using LSTM cells (which we will consistently refer to as LSTMs) have been empirically showed to outperform the conventional RNNs especially in the field of machine translation [6, 104] and speech recognition [75]. LSTMs are effective in modeling long-term temporal dependency without suffering from the optimization hurdles that plague conventional RNNs [8].

At the time step t , the vector formulas of the computation can be described as follows:

LSTM first updates the information in the input gate i_t :

$$i_t = \theta(W_i[h_{t-1}, x_t] + b_i) \quad (3.1)$$

It secondly decides which information should be deleted from the cell state. This

3.1. PRELIMINARIES

decision is controlled by the forget gate f_t :

$$f_t = \theta(W_f[h_{t-1}, x_t] + b_f) \quad (3.2)$$

Thirdly, LSTM determines which parts of new information should be stored in the memory cell c_t . A new candidate $\hat{c}_t$ is added into the memory cell c_t :

$$\hat{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c) \quad (3.3)$$

$$c_t = i_t * \hat{c}_t + f_t * c_{t-1} \quad (3.4)$$

Finally, the output gate will choose the output information:

$$o_t = \theta(W_o[h_{t-1}, x_t] + b_o) \quad (3.5)$$

$$h_t = o_t * \tanh(c_t) \quad (3.6)$$

Where i , f , and o respectively represent the input gate, forget gate, and output gate. The weight matrices (W_i , W_f , W_c , and W_o) are learned during training. The operation $[\ , \]$ concatenates h_{t-1} and x_t and the operation $*$ represents elementwise multiplication of vectors. The functions θ is the logistic sigmoid function.

Furthermore, the LSTM can be enriched with peephole connections [29] that link the memory cells to the gates to learn the precise timing of the outputs. It means that the gate values may depend not just on h_{t-1} and on x_t , but also on the previous memory cell value c_{t-1} , and the peephole connection can go into all three of these gates' computations that are i_t , f_t , and o_t .

3.2 BugTranslator

In this section, we describe the proposed model BugTranslator in detail. It adopts attention-based RNN Encoder-Decoder with LSTM cells for the task of bug localization. Two RNNs are used in BugTranslator, one for encoding and the other one for decoding [17]. This translation model is designed to learn how to translate natural languages to code tokens based on the deep understanding of semantics.

3.2.1 Model Design

3.2.1.1 Generating Training Instances

We first prepare the training set for BugTranslator: API documents, project-specific documents, and older bug reports with corresponding buggy files. We attempt to translate bug reports into corresponding buggy files based on the deep semantic similarity and relevance between them. Thus, the first training instances are the pairs of older bug reports and abstract syntax tree (AST) nodes parsed from corresponding buggy files. During testing, some out-of-vocabulary words never appear in older bug reports and their corresponding buggy files, and this is known to decrease the accuracy of most translation models [17]. In addition to older bug reports and corresponding buggy files, API documents and project-specific documents are included in the training set to enrich the vocabulary and detect some comprehensive information.

The API annotations and corresponding API sequences from API documents in Java SE 7 are extracted as noted in the literature [32]. The source code is parsed into AST nodes that include field declarations and type bindings of all classes and methods. In addition, the method-level code summaries are extracted as corresponding annotations. The project-specific documents are also included in

3.2. BUGTRANSLATOR

addition to the API documents that are generally invoked by all projects. Paired with annotations of classes, methods, and fields, the source code is parsed into AST nodes of declarations, method invocations, and class instance creations.

3.2.1.2 Attention-Based RNN Encoder-Decoder with LSTM Cells

To learn how to translate natural languages into code tokens, we build an attention-based RNN Encoder-Decoder model with LSTM cells. The workflow is shown in Figure 3.3, which illustrates an example of translating the natural language term *audio file player* into a sequence of code tokens. The source sentences are first encoded into several context vectors from which the decoder generates target sentences. The context vectors are the bridge between the source sentences and the target sentences.

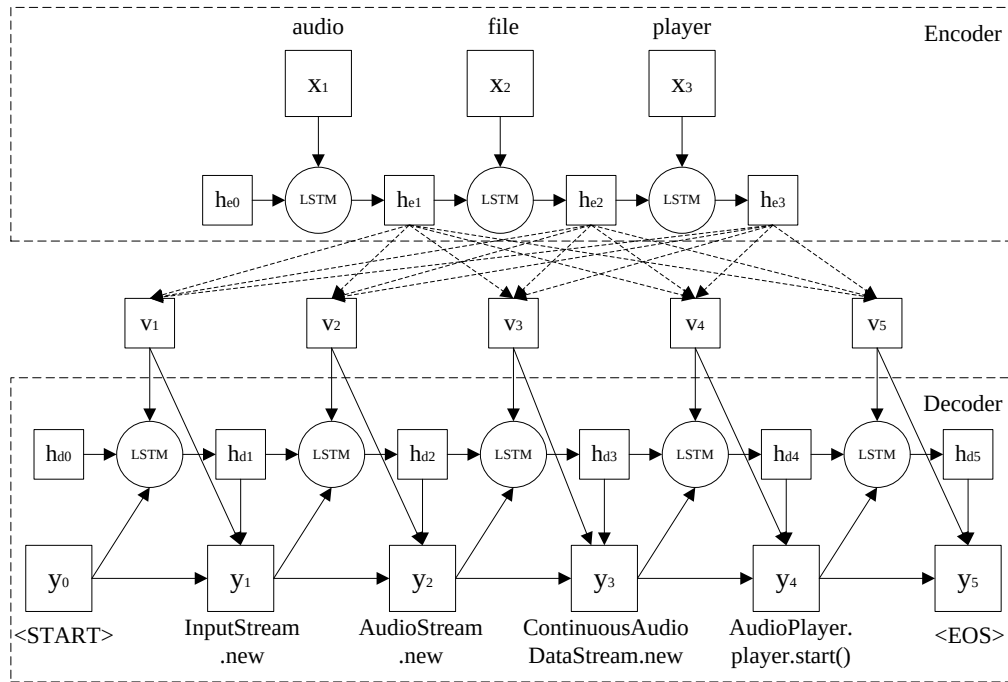


Figure 3.3: Overall workflow of attention-based RNN Encoder-Decoder with LSTM cells.

Encoder RNN The source sentences and target sentences are first embedded into 1-of-K (K is the vocabulary size)-coded word vectors [6], $X = (x_1, \dots, x_i, \dots, x_S)$ and $Y = (y_1, \dots, y_j, \dots, y_T)$ respectively, where S and T represent the lengths of the source and target sentences. The Encoder first reads the coded word vector x_1 embedded by the first word *audio* and then computes the current hidden state h_{e1} by h_{e0} and x_1 according to equation (3.7). The initial hidden state h_{e0} is set to 0. The second hidden state h_{e2} is then updated by h_{e1} and word vector x_2 of the second word. This process continues until the last hidden state h_{e3} is updated by (3.7). At each time t , the hidden state is updated by:

$$h_{et} = LSTM(h_{e(t-1)}, x_t) \quad (3.7)$$

It has been shown empirically that LSTM works well on machine translation of long sentences [104]. Because bug reports tend to include long sentences, we use RNN with LSTM cells.

In practice, each word in the source sentences has different importance to the word in the target sentences. It is inappropriate to encode the entire source sentence into only one context vector, which has also been verified experimentally in the literature [6]. Therefore, in this section, the context vector v_j at each step is expressed by the weighted sum of the hidden states of the encoder as discussed in [6].

Decoder RNN The Decoder is another RNN that is trained to generate the target sentence sequentially based on the context vectors obtained from the encoder RNN. The first word y_0 is set as $\langle START \rangle$, and the initial hidden state h_{d0} is calculated by $h_{d0} = \tanh(W_d h_{e1})$, where W_d is the weight that can be learned during training and h_{e1} is computed by equation (3.7). The Decoder then computes the hidden state h_{d1} using h_{d0} , y_0 , and the context vector v_1 by equation (3.8),

3.2. BUGTRANSLATOR

followed by prediction of the first word *InputStream.new*.

The hidden state h_{dt} at time t is computed by:

$$h_{dt} = LSTM(h_{d(t-1)}, y_{t-1}, v_t) \quad (3.8)$$

The conditional probability of y_t given the previous predicted words and context vector is defined as:

$$p(y_t | y_{t-1}, y_{t-2}, \dots, y_1, v_t) = g(h_{dt}, y_{t-1}, v_t) \quad (3.9)$$

where g is a *softmax* activation function.

This process continues until the end-of-sentence word $< EOS >$ is predicted.

The Decoder defines a probability over the target sentence Y as:

$$p(Y) = \prod_{t=1}^T p(y_t | y_{t-1}, y_{t-2}, \dots, y_1, v_t) \quad (3.10)$$

The two RNNs are then trained jointly to maximize the following conditional likelihood:

$$L(\theta) = \frac{1}{N} \sum_{n=1}^N \log(p_{\theta}(Y_n | X_n)) \quad (3.11)$$

where (X_n, Y_n) is each pair (a source sentence and a target sentence) in each batch of the training set, and θ is the set of parameters learned during training.

3.2.1.3 Translation and Generation of Scores

After creating BugTranslator, the minibatch stochastic gradient descent algorithm [64] is adopted to train the model to maximize the likelihood in equation (3.11). Given a new bug report, the trained BugTranslator can be used to score the pairs of the bug report and each source file using the probability $p_{\theta}(Y_n | X_n)$

from equations (3.10) (3.11).

3.2.2 Experiments

3.2.2.1 Preparation of Experiments

Experimental Settings: When creating the BugTranslator model, the number of hidden cells in both RNNs is set as 1000. To accelerate the training phase, we limit the vocabulary size of the source and target to 20,000.

Datasets: We use the before-fixed version of three open-source Java projects (Eclipse UI, JDT, and SWT) to evaluate the performance of BugTranslator. To make the comparison with existing techniques easier, a strategy similar to that in [134] is adopted. The oldest 1,500 bug reports of each project are used for tuning the model while the older 500 bug reports are for training the model and the remaining newest bug reports (1,656, 632, 817 bug reports respectively for Project Eclipse UI, JDT, SWT) are for testing the model. In addition to the aforementioned bug reports, we also collect corpus from Java SE 7 API Reference and project-specific documents.

Evaluation Metrics: The Accuracy@k, mean average precision (MAP) and mean reciprocal rank (MRR) are used to evaluate the performance of BugTranslator [57, 133, 134].

Competitors: Ye et al. [134] enhanced their previously proposed LR [133] with WE, and their results were comparable with the DNNLOC model [57], which outperformed other models [50, 133, 145]. Therefore, this section compares the performance of BugTranslator with WE and LR+WE.

3.2. BUGTRANSLATOR

3.2.2.2 Experimental Results and Discussions

Table 3.1 shows the MAP and MRR results of four models, as well as the values of Accuracy@5. It can be observed that the average Accuracy@5 of BugTranslator is about 56.6%. That is to say, BugTranslator can correctly locate buggy files for about 56.6% of the bug reports when recommending five source files to developers. We observe that BugTranslator achieves higher MAP and MRR values than the WE method. The WE method learns the semantic information from bug reports and the source code based on word embedding and calculates the similarity between them. However, if some word pairs never appear in the same context but are relevant to each other, word embedding has a lower probability of assigning them as close vectors, which limits the performance of WE. Moreover, because word embedding focuses on words more than sentences, it is difficult for WE to learn the relative positions between words, which can be learned by BugTranslator by benefiting from the two RNNs with memory capacity. In other words, BugTranslator learns the semantic information from not only words but also the full sentence. It can thus alleviate the effect of lexical mismatch based on the deeper understanding of semantics between bug reports and source code by benefiting from the context vectors between the two RNNs.

Table 3.1: Results of four models.

Project	Metrics	WE	LR+WE	BugTranslator	LR+BugTranslator
Eclipse UI	MAP	0.26	0.40	0.36	0.41
	MRR	0.31	0.46	0.42	0.48
	Accuracy@5	47.6	60.2	58.1	61.4
JDT	MAP	0.22	0.42	0.34	0.45
	MRR	0.27	0.51	0.41	0.52
	Accuracy@5	45.6	62.3	55.4	63.1
SWT	MAP	0.25	0.38	0.34	0.38
	MRR	0.30	0.45	0.40	0.46
	Accuracy@5	46.2	59.1	56.2	59.9

According to Table 3.1, both WE and BugTranslator perform worse than LR+WE. In addition to the semantic similarity calculated by the WE method, LR+WE also considers factors such as bug-fixing recency, frequency, and the observation that a previously fixed file may be responsible for similar bugs. These factors are concluded based on the experience of many developers, which is not learned by BugTranslator directly. We also combine BugTranslator with LR in emulation of the approach LR+WE [134]. The observed results of LR+BugTranslator are comparable to those of LR+WE, indicating the effectiveness of BugTranslator. Then we talk about two examples about the ranking results. There are two bug reports in Project SWT: “Bug 369228 Kill pre GTK 2.4 leftovers from Tree (buggy files: *Tree.java*, *Table.java*, *List.java*, *Display.java*, and *Widget.java*)” and “Bug 369227 Kill pre GTK 2.4 leftovers from List (buggy files: *List.java*)”. For Bug 369228, the buggy file *Tree.java* was ranked the first by BugTranslator because of related AST nodes (e.g., the method name). However, it did not give the other buggy files (*Table.java*, *List.java*, *Display.java*, and *Widget.java*) very high ranks (4th, 6th, 11th and 25th), whereas LR+BugTranslator gave these buggy files higher ranks (5th, 3rd, 7th and 20th). It seems that LR+BugTranslator outperformed BugTranslator. However, for Bug 369227, LR+BugTranslator located and ranked *List.java* 3rd. *Display.java* and *Table.java* were ranked 1st and 2nd because of their greater frequency and earlier recency considered in LR, which instead had a negative effect on the results. But BugTranslator located the buggy file *List.java* and ranked it first, which outperformed LR+BugTranslator.

3.2.3 Discussion

Lexical mismatch is caused by the difference in the expressions of bug reports and source code files. Existing techniques [133, 134, 57] regard both in the same language expression. These techniques measure their similarities, which does not

3.3. CHARACTER-LEVEL CONVOLUTIONAL NEURAL NETWORK AND RECURRENT NEURAL NETWORK

solve the root cause of the lexical mismatch problem. Unlike existing approaches, our proposed BugTranslator makes a distinction between bug reports and source code based on the RNN Encoder-Decoder model. The bug reports are encoded into the context vectors from which the source code files are decoded. They are placed in different language platforms, and context vectors are used to bridge them based on the semantic relevance between them. As observed from our experimental results, we argue that BugTranslator can handle and alleviate the issue of lexical mismatch in a radical way.

3.3 Character-level Convolutional Neural Network and Recurrent Neural Network

The code tokens in source files are similar to English words in natural languages while the difference is obvious. Some code tokens, especially the class or method names, are not actual words that are commonly used in natural languages. Although they are very important in bug localization, most existing studies regarded them as unknown words [134, 128, 126]. However, both bug reports and source files are composed of characters. They share same expressions in character level. Therefore, this section proposes a bug localization technique based on a language model in character-level. The proposed model first obtains the character embeddings of the preprocessed bug reports and source files. Two CNNs with multiple filters are then applied to extract features respectively from the vectors of bug reports and source files, whose outputs are fed into the subsequent RNN encoder-decoder architecture. The experiments on three open-source Java projects show the feasibility and effectiveness of the proposed model.

3.3.1 The Proposed Model

This section describes the proposed model whose overview is illustrated in Figure 3.4.

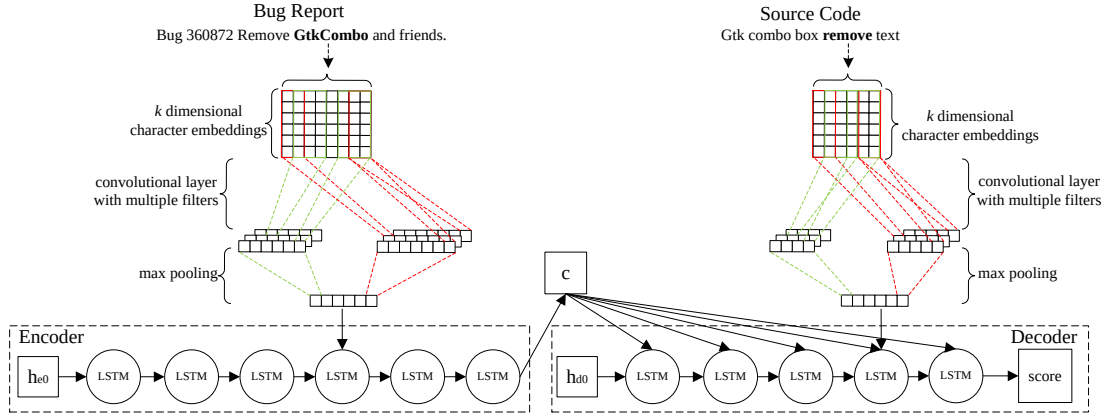


Figure 3.4: The overview of the proposed model.

3.3.1.1 Data Preprocessing

A bug report usually contains summary, meta data (status, version, severity, reporter information and so on), description, comments. Summary and description are the main source for developers to localize bugs, which are also commonly used in the existing bug localization techniques. We first combine summary and description in bug reports to be a new document. Revised term frequency-user focused inverse document frequency (TF-IDuF) is then applied to filter some common words in the new bug reports for the purpose of redundancy reduction [128]. We also extract two types of Abstract Syntax Tree nodes from each source file as [126].

3.3. CHARACTER-LEVEL CONVOLUTIONAL NEURAL NETWORK AND RECURRENT NEURAL NETWORK

3.3.1.2 Character-level CNN

After preprocessing bug reports and source code, convolutional operations are applied in each word of them respectively. Each character in a word is transformed into a k -dimensional character embeddings, which are then convolved by multiple filters with different sizes ($2 \times k$ and $3 \times k$ in Figure 3.4). The subsequent max-pooling layer is used to conclude the features, whose outputs are given to encoder and decoder respectively.

3.3.1.3 RNN Encoder-Decoder

Encoder The output features extracted from a word in bug reports by the character-level CNN are given to one long short-term memory (LSTM) cell. For example, the feature vectors of the fourth word in a bug report are fed into the fourth LSTM in the encoder as shown in Figure 3.4. The context vector c is the final state of the encoder, which is the conclusion of the features of bug reports that will be one part of input in the decoder.

Decoder Similar to the encoder, the features extracted from each word in source code by the character-level CNN are one part of the input to each LSTM cell. Besides, the context vector is concatenated with output features of each word to be fed into each LSTM.

3.3.2 The Preliminary Results

In order to validate the feasibility and effectiveness of the proposed model, we conduct several preliminary experiments on the before-fixed version of three open-source Java projects ¹ similar to [126]. 3656, 2632, 2817 bug reports respectively for Project Eclipse UI, JDT, SWT are used. Mean average precision (MAP) and

¹<https://github.com/yanxiaoj6/BugLocalization-dataset>

mean reciprocal rank (MRR) are used to evaluate the performance of the proposed model and the competitor BugTranslator [126].

Table 3.2: Results of two models.

Project	Metrics	BugTranslator	Proposed model
Eclipse UI	MAP	0.36	0.35
	MRR	0.42	0.40
JDT	MAP	0.34	0.35
	MRR	0.41	0.42
SWT	MAP	0.34	0.37
	MRR	0.40	0.42

The preliminary results are shown in Table 3.2. The MAP and MRR values of the proposed model are better than BugTranslator in Project JDT and SWT. The performance of BugTranslator is limited since it analyzes bug reports and source files in word level where many out-of-vocabulary words exist. But BugTranslator gives more emphases on the related words in buggy files with those in bug reports using an attention mechanism. It is much important when there are large amounts of source files. In Project Eclipse UI, the number of source files is 6228 that is about five times the number in Project SWT. The performance of BugTranslator is thus better than our proposed model in Project Eclipse UI.

3.4 Chapter Summary

The lexical mismatch between source code files and bug reports degrades the performance of the existing bug localization techniques [134]. This chapter presents two new translation models, BugTranslator and an enhanced version using character-level CNN, for bug localization, which differs from existing techniques to alleviate the effect of lexical mismatch in a fundamental manner based on the attention-based RNN Encoder-Decoder with LSTM cells. The experimental re-

sults show that BugTranslator performs better than the existing single model. When combined with existing models, as with other combined models, BugTranslator achieves comparable results. In the enhanced version, bug reports and source files are analyzed in character-level instead of word-level to suppress the effect of different expressions on the accuracy of bug localization. The number of unknown words is also reduced. The proposed model applies character-level CNN to extract features from bug reports and source files, whose output is fed into the subsequent RNN encoder-decoder. The preliminary results indicate the feasibility and effectiveness of the proposed model.

According to the experimental results obtained from BugTranslator and the enhanced version, the out-of-vocabulary words have more effect on the bug localization accuracy when the projects have a small number of bug reports, saying less than one thousand. Therefore, it is necessary to balance the overhead between the accuracy improvement and the consumption of time and memory since the enhanced version adding CNN on top of BugTranslator that makes the model more complex. More empirical experiments are needed to obtain more comprehensive investigations about how to balance the overhead. Significantly, we would like to explore how to add the bug-fixing history into the model itself instead of simply combined with other models as the existing work did.

Chapter 4

Semantic Information Extraction for Bug Localization

In this chapter, we first discuss preliminaries and then describe the proposed enhanced CNN for bug localization including the important bug-fixing experience (bug-fixing recency and frequency) besides the semantic information that can be extracted by conventional CNN from bug reports and source files. To extract the underlying semantic information from bug reports and source code, this chapter propose two CNN-based models for bug localization, namely DeepLocator and DeepLoc. CNN can extract local features by convolving filters and performs well in semantic parsing-related natural language processing (NLP) problems [51]. Thus, CNN is adopted to extract features from vectors of bug reports and source files converted by word-embedding techniques. Moreover, DeepLoc learns to correlate bug reports to the corresponding buggy files using an enhanced CNN, rather than textual similarity, which is used in IR-based approaches. Finally, we summarize this chapter. The proposed DeepLocator [128] and DeepLoc [127] have been published in APSEC 2017 and Information & Software Technology respectively.

4.1 Preliminaries

4.1.1 Term Frequency-Inverse Document Frequency (TF-IDF)

The Term Frequency-Inverse Document Frequency (TF-IDF) technique is widely used in text mining and information retrieval (IR). TF represents the frequency with which a term appears in a document, and IDF denotes the inverse of the number of documents in which the term appears in the entire corpus. TF-IDF can filter some common terms in documents in specified repositories. The formulas for TF-IDF are as follows [133]:

$$tf_{t,d} = f_{t,d} \quad idf_{t,d} = \log\left(\frac{N}{df_t}\right)$$

$$w_{tf-idf} = tf_{t,d} \times idf_{t,d} \tag{4.1}$$

where $f_{t,d}$ is the frequency of term t that appears in document d , N refers to the total number of documents in the document collections, and df_t reflects the number of documents in the corpus containing term t . However, this kind of general corpus is not always available in practice or for research, and the equations represent general behaviors that ignore some important individual information.

To address the aforementioned problems, Beel et al. [7] presented term frequency-user focused inverse document frequency (TF-IDuF) using personal document repositories instead of a general corpus. TF-IDuF is defined as follows [7]:

$$tf_{t,d} = f_{t,d} \quad idf_{t,d} = \log\left(\frac{N_u}{n_{t,u}}\right)$$

$$w_{tf-idf} = tf_{t,d} \times idf_{t,d} \tag{4.2}$$

where N_u refers to the total number of documents in the user collections and $n_{t,u}$

reflects the number of documents in the collections that incorporate term t .

Equations (4.1) and (4.2) are fundamental for term weights. When large differences in frequencies are present, the term weights computed by (4.1) and (4.2) do not achieve good performance in practice [22]. Therefore, it is necessary to dampen the effects of large differences in frequencies, which is further discussed below.

The logarithm variant of TF can improve the performance of TF-IDF in practice, as demonstrated in Croft’s experiments [22]. Specifically, it can dampen the effects of large differences in frequencies:

$$tf_{t,d} = \log(f_{t,d}) + 1 \quad (4.3)$$

Inspired by Croft’s work [22], we propose an rTF-IDuF to further improve the model’s performance:

$$w_{tf-idf} = (\log(f_{t,d}) + 1) \times \log\left(\frac{N_u}{n_{t,u}}\right) \quad (4.4)$$

where $f_{t,d}$ is the frequency of term t appearing in document d , N_u refers to the total number of documents in user collections, and $n_{t,u}$ reflects the number of documents in user collections that contain term t .

In reality, a complete corpus is not always available. In addition, the bug reports for different projects usually differ in their writing style and contexts. Therefore, it is convenient and reasonable to apply rTF-IDuF, which focuses on personal collections (different corpora for different projects) and is used in this thesis.

4.1.2 Word Embedding

Word embedding refers to the distributed representations of words in a vector space in which similar words are close to each other [78]. After performing word embedding, words are represented by vectors of real values instead of text, which enables text analysis using various statistical models. Mikolov et al. [76] proposed word2vec techniques for efficient learning of high-quality distributed vector representations. Furthermore, this model can capture precise syntactic and semantic word similarities.

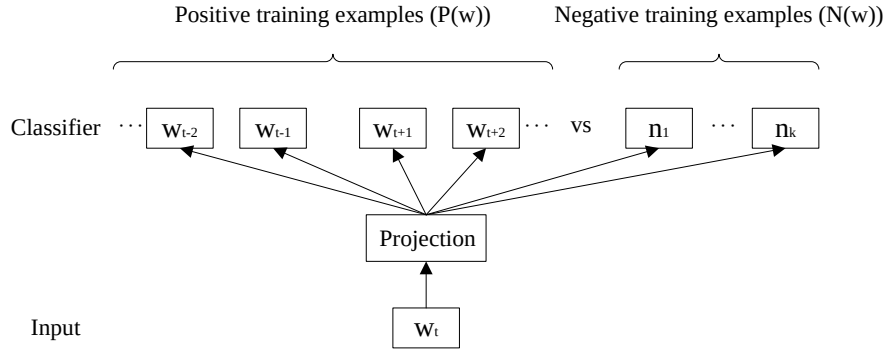


Figure 4.1: The architecture of the Skip-gram model.

Figure 4.1 illustrates the architecture of the Skip-gram model, which uses one of the word2vec techniques. The objective of the Skip-gram model is to learn word representations that maximize the classification of the surrounding words based on a current word in the same sentence. Formally, given a sequence of T training words $w_1, \dots, w_t, \dots, w_T$, each current word w_t is used as the input to predict its context words w_{t+r} within a certain range. The Skip-gram model is trained to maximize the average log probability:

$$\frac{1}{T} \sum_{t=1}^T \sum_{-c \leq r \leq c} \log p(w_{t+r} | w_t) \quad (4.5)$$

4.1. PRELIMINARIES

where c is the size of the training context centered at the current word w_t and $r \neq 0$.

Because the cost of computing the derivative of each log probability in Equation (4.5) is proportional to the number of words in large vocabulary, Mikolov et al. [78] introduced a negative sampling technique to accelerate the Skip-gram model. Using this technique, its optimization objective is defined as:

$$\frac{1}{T} \sum_{t=1}^T \left(\sum_{w_i \in P(w)} \log \sigma(w_i^\top w_t) + \sum_{w_j \in N(w)} \log \sigma(-w_j^\top w_t) \right) \quad (4.6)$$

where $\sigma(x) = 1/(1 + \exp(-x))$, $P(w)$ is the positive training examples and $N(w)$ is the randomly sampled negative training examples of w_t .

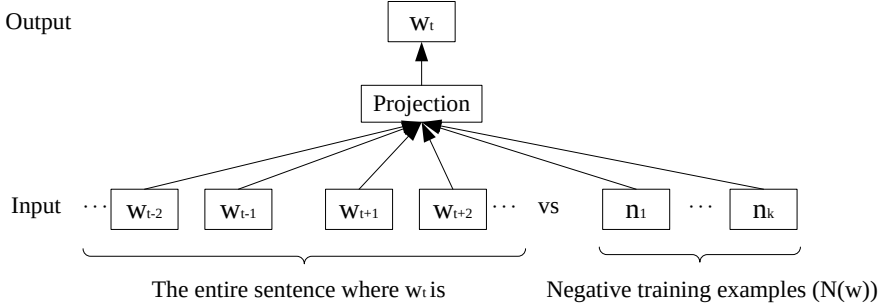


Figure 4.2: The architecture of Sen2Vec [87].

Another word2vec technique proposed by Mikolov et al. [76] is the continuous bag-of-words (CBOW) model, which is similar to the Skip-gram model. However, instead of predicting context words based on the current word, CBOW tries to maximize the classification of the current word based on the context words in the same sentence. Pagliardini et al. proposed Sent2Vec [87], which is an extension of CBOW [76], to compose sentence embedding. Unlike in CBOW, the context words running over the corpus in Sent2Vec are entire sentences instead of fixed-length context words, whose architecture is shown in Figure 4.2. w_t in formula (4.6) is

now changed into the entire sentence where w_t is, but w_t is excluded.

4.1.3 Convolutional Neural Network (CNN)

CNN, inspired by biological processes and multilayer perceptron, is the first learning algorithm based on neocognitron [27] and receptive fields [38]. CNN contains several convolutional layers that are often combined with pooling steps and then followed by a fully-connected layer similar to a standard multilayer neural network [54].

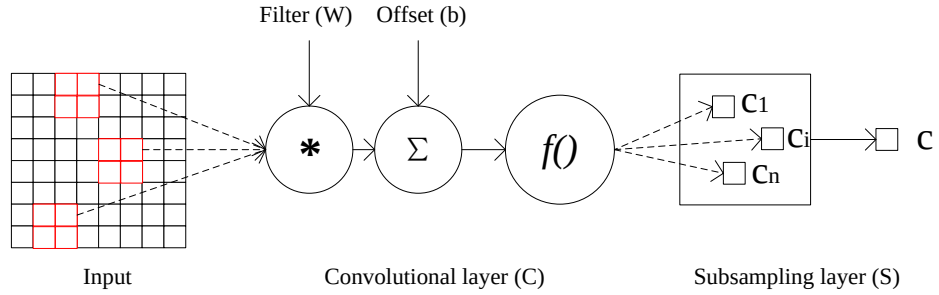


Figure 4.3: The process of convolution and subsampling of a CNN.

The process of convolution and pooling (subsampling) [111] is shown in Figure 4.3. In the process of convolution C , the input is convoluted with a trainable filter W and an added offset b . Later, a nonlinear function is used to obtain a reduced feature map c_i , followed by is the subsampling procedure S . A mean-over-time polling or max-over-time pooling operation [21] is frequently used in this layer, which takes the mean ($c = \text{mean}(c_i)$) or maximum value ($c = \text{max}(c_i)$) as the feature corresponding to this particular filter. The convolutional layer is used to detect features that are then subsampled in the pooling layer to reduce the number of parameters. The process also alleviates the over-fitting problem [111].

The basic structure of a CNN [55] is shown in Figure 4.4. C layers are used to capture features. The input of every neuron in C layers is linked with the local

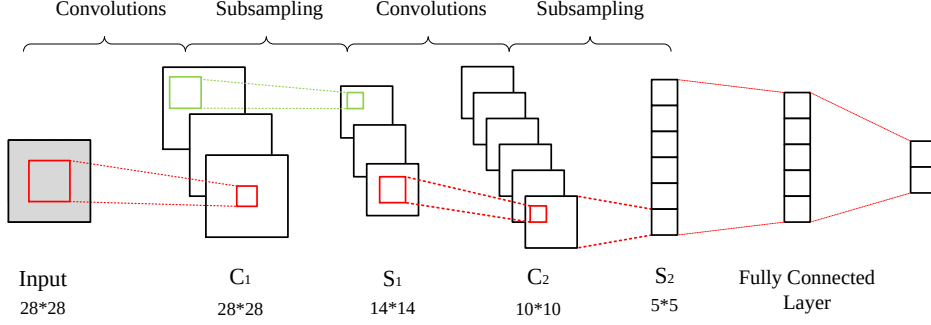


Figure 4.4: The net structure of a CNN.

receptive field of the former layer, and the local feature is extracted using convolutional operations. Once the local feature is extracted, its positional relationship with other features is known. S layers are feature mapping layers. Each computational layer has several feature maps, and every feature map is a two-dimension surface. The weight of each neuron in this surface is the same, which follows the well-known parameter sharing scheme. The Rectified Linear Units (ReLU) [54] function is used to increase the network's nonlinear properties.

4.1.4 Batch Normalization

Batch normalization was proposed by Ioffe et al. [40] to reduce internal covariate shift for deep networks. It allows the network to be trained at a much faster learning rate and reduces its sensitivity to initialization. Thus, the use of batch normalization can accelerate the training process more than tenfold while retaining the same or much greater accuracy [40]. Given a mini-batch with m values $(x_1, \dots, x_i, \dots, x_m)$, the transforming procedure is as follows:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \quad (4.7)$$

$$BN(x_i) = \gamma \hat{x}_i + \beta \quad (4.8)$$

where μ and σ are the mean ($\mu = \frac{1}{m} \sum_{i=1}^m x_i$) and variance ($\sigma = \frac{1}{m} \sum_{i=1}^m (x_i - \mu)^2$) of the mini-batch, ϵ is a constant (e.g., 1e-3) added to the variance for the numerical purpose, and γ and β are additional parameters to be learned during training.

4.2 Enhanced CNN

Although the conventional CNN has good performance in semantic parsing-related NLP problems [51], it cannot learn bug-fixing experience by itself. Therefore, this section discusses a novel technique to enhance the CNN for bug localization.

The conventional CNN is trained to minimize the following mean cross-entropy error function [32]:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^N cost_i \quad (4.9)$$

$$cost_i = - \sum_{j=1}^T t_{ij} \log(y_{ij})$$

where N is the number of samples in a batch and T is the number of classes. $cost_i$ is the cost function of sample i . t_{ij} is the true value of class j of sample i and y_{ij} is the output probability of class j of sample i .

Apart from the semantic information, the conventional CNN model contains no other important information related to bug localization, especially the fixing history of source files. Furthermore, bug-fixing recency and frequency have been verified as useful for bug localization by [56] and [133]. The change history of source files for bugs contains useful information for identification of fault-prone files [92]. This implies that source files fixed recently are more likely to be buggy than those fixed long ago (bug-fixing recency) and that source files that have been fixed many times are more likely to still contain bugs than those seldom fixed or even never fixed (bug-fixing frequency).

4.2. ENHANCED CNN

We use equations similar to those in Ye's work [133] to identify the source file's bug-fixing recency and frequency. For a sample i with source file s and new bug report r , the bug-fixing recency is defined as follows:

$$R_i = \frac{1}{r.month - s.month + 1} \quad (4.10)$$

where $r.month$ represents the month in which bug report r was generated and $s.month$ denotes the month in which source file s was last fixed before bug report r was created. $r.month$ and $s.month$ take the year into account. For example, if $r.month$ is January this year and $s.month$ is December last year, R_i is 0.5. Therefore, the bug-fixing recency R_i is the inverse of the time interval between the creation of the bug report and the last time the source file was fixed.

The bug-fixing frequency F_i is expressed by the number of times source file s was fixed before bug report r was submitted [133].

R_i and F_i may have wide value ranges that make them incomparable, which may result in detrimental effects [133]. In both the training and testing sets, R_i and F_i are scaled as follows:

$$\begin{aligned} r_i &= \begin{cases} 0 & \text{if } R_i < R_{min} \\ \frac{R_i - R_{min}}{R_{max} - R_{min}} & \text{if } R_{min} \leq R_i \leq R_{max} \\ 1 & \text{if } R_i > R_{max} \end{cases} \\ f_i &= \begin{cases} 0 & \text{if } F_i < F_{min} \\ \frac{F_i - F_{min}}{F_{max} - F_{min}} & \text{if } F_{min} \leq F_i \leq F_{max} \\ 1 & \text{if } F_i > F_{max} \end{cases} \end{aligned} \quad (4.11)$$

where R_{min} (F_{min}) and R_{max} (F_{max}) are the minimum and maximum values of R_i

(F_i) in the training set. During testing, $R_i(F_i)$ may be larger than $R_{max}(F_{max})$ or smaller than $R_{min}(F_{min})$ obtained in the training phase, so the first and last judgments, i.e., $R_i(F_i) < R_{min}(F_{min})$ and $R_i(F_i) > R_{max}(F_{max})$, are necessary for testing set.

To improve the bug localization performance, we enhance the conventional CNN by adding bug-fixing recency and frequency to the cost function in the fully connected layer as two penalty terms. The new cost function that considers bug-fixing recency (r_i) and bug-fixing frequency (f_i) is defined as follows:

$$cost_i = - \sum_{j=1}^T t_{ij} \log(y_{ij}) - \omega_1 r_i - \omega_2 f_i \quad (4.12)$$

where ω_1 and ω_2 are initialized by random values from a truncated normal distribution and tuned in the training phase similarly to other weights. r_i and f_i are the scaled values of bug-fixing recency and frequency, respectively.

4.3 DeepLocator

Localizing buggy files automatically speeds up the process of bug fixing so as to improve the efficiency and productivity of software quality teams. There are other useful semantic information available in bug reports and source code, but are mostly underutilized by existing bug localization approaches. We propose DeepLocator, a novel deep learning based model to improve the performance of bug localization by making full use of semantic information. DeepLocator is composed of an enhanced CNN (Convolutional Neural Network) proposed in this study considering bug-fixing experience, together with a new rTF-IDuF method and pre-trained word2vec technique. DeepLocator is then evaluated on over 18,500 bug reports extracted from AspectJ, Eclipse, JDT, SWT and Tomcat projects. The

4.3. DEEPLOCATOR

experimental results show that DeepLocator achieves 9.77% to 26.65% higher F-measure than the conventional CNN and 3.8% higher MAP than a state-of-the-art method HyLoc using less computation time. DeepLocator is capable of automatically connecting bug reports to the corresponding buggy files and successfully achieves better performance based on a deep understanding of semantics in bug reports and source code.

The main contributions of this section are listed as follows:

- We develop DeepLocator that consists of a new rTF-IDuF method, word2vec and a proposed enhanced CNN. The enhanced CNN makes use of the important bug-fixing experience (bug-fixing recency and frequency) besides the semantic information that can be extracted by conventional CNN from bug reports and source files.
- A set of experiments is conducted to validate the feasibility and effectiveness of DeepLocator. We make our dataset publicly available¹ and provide the tool used to construct the dataset.

4.3.1 Model Design

In this section, we describe DeepLocator, a deep learning based model that localizes the buggy files for bug reports automatically. DeepLocator adapts an enhanced CNN proposed in Section 4.2 for the task of bug localization. Figure 4.5 provides the overall workflow of DeepLocator. In the training phase, bug reports and source files are firstly pre-treated by rTF-IDuF as discussed in Section 4.1.1 and Abstract Syntax Tree (AST) detection respectively. Word embedding is then applied to transform the words into vectors. The mappings between bug reports and source files, as well as corresponding labels (buggy or not buggy), are obtained

¹<https://github.com/yanxiao6/BugLocalization-dataset>

to train the enhanced CNN. The training stage can be done offline. When we receive new bug reports, the already-trained enhanced CNN will efficiently utilize the bug reports to obtain the location of buggy files. The details of DeepLocator are discussed in the following sections.

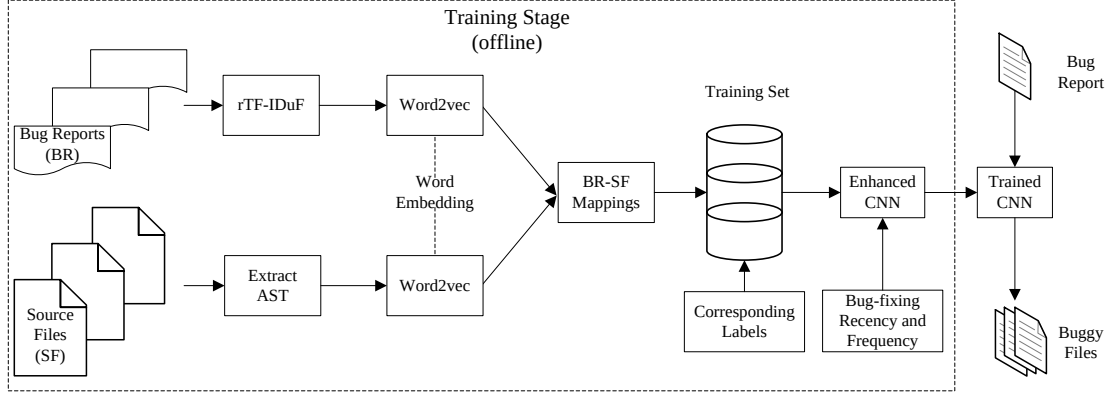


Figure 4.5: The overall workflow of DeepLocator.

4.3.1.1 Data Preprocessing

Typically, there are summaries and descriptions for each bug in the bug reports. Because most of the summaries are very short or sometimes even vacant, we first combine the summary and description of each bug into one document. Next, we apply proposed rTF-IDuF to filter some common words (such as I, the, to) to shorten the length of each sentence, which is a standard data preprocessing to eliminate redundant words [133][145][50].

The well-defined programming syntax can be represented by AST [35] that has been successfully used to extract the programming patterns of source code [86]. The two main AST nodes we extract from the source code are [116]: a) nodes of method invocations and class instance creations, e.g., method names. b) declaration nodes, e.g., method declarations.

All words are further preprocessed because developers most often combine

4.3. DEEPLOCATOR

words to create a new "word" when defining the classes or methods in the source code, which is also possible to be mentioned in bug reports. According to Camel-Case Naming Convention [9], combined words can be split into separated real words based on capital letters. For example, "WorkbenchActionBuilder" is split into "Workbench", "Action" and "Builder". Besides, we pay attention to some special capital words, such as URI, IDE and so on, which should not be separated. Lastly, we change all the capital letters into lower cases to meet the rules of word embedding.

4.3.1.2 Word Vectors Obtained by word2vec

Words themselves cannot be the input of CNN directly. Thus the pre-treated words should be embedded into vectors. We first find the maximum sentence lengths from bug reports and preprocessed AST nodes respectively. Then they are padded by <PAD> tokens to be separately same with their maximum sentence lengths. In this section, word2vec with Skip-gram model [76] is applied to transform the preprocessed bug reports and AST nodes into vectors. There may be some words absent in the set of pre-trained words of word2vec. These words are initialized randomly and fine-tuned during training [51]. Ye et al. [134] verified that word embeddings trained on Wiki corpus and project (Eclipse and Java) specific corpus had similar performance. But Wiki corpus has a larger size of vocabulary and words (96 times and 548 times the size of project specific corpus respectively). So the pre-trained word2vec used in our model is based on Wiki corpus.

4.3.1.3 Enhanced CNN Training and Prediction

Now we begin building the enhanced CNN. Firstly, we construct the training dataset. Besides the real corresponding buggy files, we only select the files that

were ever buggy files as the negative samples due to the imbalanced dataset. We consider them as potential buggy files.

As we discussed in 4.2, an enhanced CNN is adopted for the main training part, which contains one convolutional layer followed by a max pooling layer. We also add a dropout layer that is frequently used to regularize CNN. The dropout layer only enables a fraction of the neurons randomly, which prevents neurons from co-adapting and at the same time forces neurons to learn useful features individually. The following Figure 4.6 is the structure of CNN used in this section. The sentence contains pre-treated bug reports and AST nodes of source code.

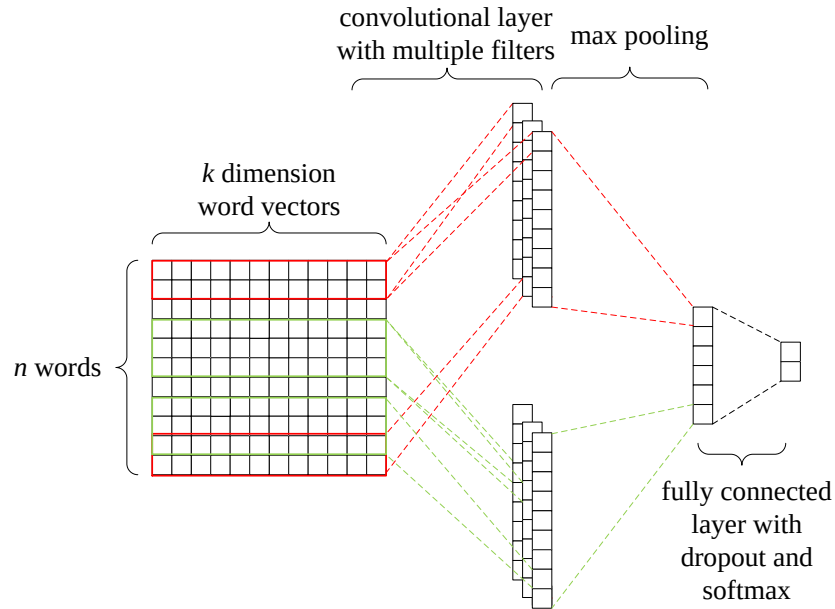


Figure 4.6: The architecture of CNN for a sentence. n is the number of words in the sentence and k is the vector dimension of each word that is tuned experimentally (details are shown in Figure 4.7).

This model is trained by gradient descent algorithm together with Adam (Adaptive Moment Estimation) optimizer, which uses an adaptive learning algorithm. Compared to Adadelta [137] that keeps an exponentially decaying average of past squared gradients, Adam can also store the exponentially decaying average of past

gradients. What’s more, Adam has shown better performance than other algorithms that automatically adjust learning rates [53].

After the trained CNN is available, the bug reports have been related to buggy files instead of matching them by textual similarity. When a new bug report comes, it is paired with each source file and then input into the trained CNN to classify buggy files. The probability of being the buggy file is used to give priority to more buggy-prone files.

4.3.2 Experiments

Several experiments are conducted to evaluate the performance of DeepLocator. We describe the datasets used, the experimental setup and performance metrics in Section 4.3.2.1. The research questions are listed in Section 4.3.2.2, followed by the results in Section 4.3.2.3.

4.3.2.1 Experimental Settings and Evaluation Metrics

As some source files may be modified or even deleted, for an old bug, we may fail to find the original mapped source files without before-fix versions. So we collect dataset (Table 4.1) with before-fix versions and relate them to each bug based on the publicly available mappings of bug reports and the corresponding commit history provided by [133]. We use tensorflow to construct the model and all experiments are run on a server with CPU Intel Xeon CPU E5-4620 2.20GHz (32 cores), 128 GB RAM.

The chronologically sorted bug reports of each project are split into 70% as the training set (older bugs) and 30% as the testing set (newer bugs). This splitting strategy is adopted by many previous studies [50, 4] and we further validate the effect of different training sizes on performance in RQ4. In order to evaluate the performance of DeepLocator, three metrics are adopted: Precision rate (P), Recall

Table 4.1: Subject projects. [133]

Project	Time Range	# of Bug Reports	# of Fixed Buggy Files	Avg. # of Buggy Files per Bug	Max. # of Words per Bug Report	Avg. # of Words per Bug Report
AspectJ	03/02-01/14	593	1,151	4.0	4,068	188.5
Eclipse UI	10/01-01/14	6,495	6,228	2.7	4,468	124.6
JDT	10/01-01/14	6,274	5,002	2.6	5,233	137.4
SWT	02/02-01/14	4,151	1,415	2.1	5,125	125.0
Tomcat	07/02-01/14	1,056	1,038	2.4	2,320	87.5

rate (R) and F-measure (F), which are widely used in classification tasks [50]. Their definitions are expressed as follows.

$$P = \frac{\text{true positive}}{\text{true positive} + \text{false positive}} \quad (4.13)$$

$$R = \frac{\text{true positive}}{\text{true positive} + \text{false negative}} \quad (4.14)$$

$$F = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \quad (4.15)$$

where true positive represents the number of predicted buggy files that are truly buggy files, while true negative denotes the number of predicted non-buggy files that are truly buggy-free files. False positive represents the number of predicted buggy files that are actually non-buggy files, while false negative denotes the number of predicted buggy-free files that are actually buggy files. A good classification model desires to obtain higher P and R. As a comprehensive measurement, F is a trade-off between P and R.

4.3.2.2 Research Questions

Specifically, we aim to answer the following questions:

RQ1: What is the effect of pre-trained word2vec on DeepLocator?

Iyyer et al. [42] reported that word vectors initialized by unsupervised neural language models had better performance than those randomly initialized in Natural Language Processing (NLP). In order to validate the effectiveness of pre-trained word2vec in our model, we compare the accuracy of word vectors obtained by pre-trained word2vec and random initialization respectively. We also analyze the influence of the dimension of word embedding on accuracy.

RQ2: What is the effect of model settings on DeepLocator? Before building DeepLocator, we need to analyze two main settings: the size of filters and the number of convolutional layers.

RQ3: What is the performance of DeepLocator? Our evaluation can be divided into two steps. First, we compare the conventional CNN with two classification models (i.e., SVM classifier [100] and Naive Bayes classifier [50]). Then, we evaluate whether the enhanced CNN improves the accuracy of DeepLocator against the conventional CNN. The only difference between the two CNNs is that the enhanced CNN considers bug-fixing recency and frequency but the conventional CNN does not.

RQ4: What is the effect of larger dataset on accuracy of DeepLocator? Some related works of bug localization, such as [56] and [133], split the datasets into 10 folders. Ye et al. [133] claimed that the increase of training datasets had little effect on the accuracy. But most deep learning models require more data to obtain the features hidden deeply in the datasets. In order to evaluate the impact of different sizes of training dataset on DeepLocator, the chronologically sorted bug reports of the large projects (Eclipse UI and JDT) are split into 10 folders. $fold_1$ consists of oldest bug reports and $fold_{10}$ contains newest bug

reports. We keep $fold_{10}$ as the unchanged testing dataset. The 9 training datasets are used incrementally. To begin, only one folder ($fold_9$) is used as training dataset and then two folders ($fold_9 \cup fold_8$), and so forth, until all 9 folders ($fold_9 \cup fold_8 \cup \dots \cup fold_2 \cup fold_1$) are used as training datasets.

4.3.2.3 Results and Analyses

RQ1: Impacts of Pre-trained Word2vec on Accuracy.

We only show the results of Project AspectJ. Note we also experiment with other projects; the results are qualitatively similar. As shown in Figure 4.7, the accuracy by using pre-trained word2vec is always higher than using random initialization, which indicates that word vectors initialized by the unsupervised neural language model (word2vec) perform better than those initialized randomly in bug localization.

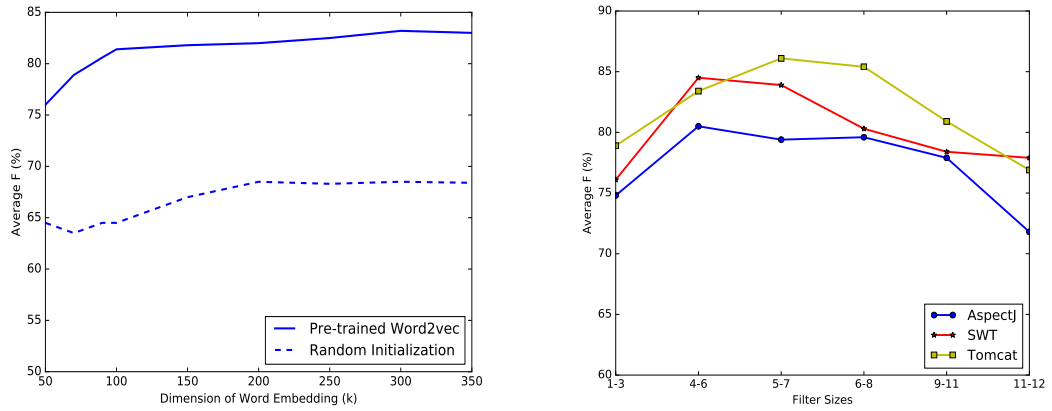


Figure 4.7: Performance of word2vec and random initialization along the dimension of word embedding (k). Figure 4.8: Performance of different filter sizes.

Moreover, according to Figure 4.7, the word embedding dimension of word2vec has little effect on the accuracy, while the increasing dimension of word vectors randomly initialized can improve the accuracy at the beginning. Compared to

word2vec, word vectors randomly initialized contain fewer characteristics. So more vectors are needed to represent and distinguish words. But if more dimensions of word vectors are used, the dimension of input will increase dramatically, which augments the training and predicting time. Therefore, the word vector dimension in our experiments is set as 100.

RQ2: Accuracy Under Different Model Settings.

We first evaluate how to choose the size of filters. According to [48], when the size of n -gram is from 4 to 8, there are more possibilities to improve the performance of expression. To make the figure distinct, the accuracy of three projects (AspectJ, SWT and Tomcat) along different filter sizes are showed in Figure 4.8. Although the accuracies of these three projects fluctuate when the filter size is between 4 and 8, they are still higher than the accuracy when the filter size is from 1 to 3 and from 9 to 12. If the filter size is very small, e.g. 1-3, the model fails to extract the semantic features because each sentence is split into one or two words that have less semantic information. On the other hand, if larger filter size is used, in other words, more words are in a group, it is harder for the model to extract semantic features from a large group. So in our experiments, the filter size is chosen from 4 to 6 whose accuracies are relatively high.

Next, we compare the average F and the average time consumption for each batch of CNN by using one convolutional layer and two convolutional layers. When using a two-layer CNN, the features between words are convolved again. So a deeper semantic information is extracted. In Table 4.2, it can be observed that the performance of the two-layer CNN is a bit better than the one-layer CNN, yet the cost time is much higher. So DeepLocator uses a one-layer CNN.

RQ3: Performance of DeepLocator.

We compare conventional CNN with other techniques. We report the highest F of the two classifiers (SVM and Naive Bayes) that are 46.71% and 53.57% re-

Table 4.2: Accuracy and time consumption on different number of layers.

Project	One-layer		Two-layer	
	F (%)	Time (s)	F (%)	Time (s)
AspectJ	79.92	8.69	81.16	692.26
Eclipse UI	84.13	8.83	85.58	689.83
JDT	82.28	8.78	83.21	702.18
SWT	83.61	7.93	84.71	682.61
Tomcat	85.04	7.69	85.73	696.67

spectively, which is lower than the average F of CNN (66.61%). Then Table 4.3 reports the average values of Precision, Recall, and F-measure for each bug paired with all source files, maximum F and minimum F of DeepLocator and CNN. We find that the P, R and F of DeepLocator are all higher than CNN. In comparison with CNN, DeepLocator achieves from 9.77% to 26.65% higher average F. Because CNN does not have the long-term memory, it cannot learn the features about bug-fixing recency and frequency, which are included in DeepLocator. Furthermore, the relative interval between F_max and F_min of CNN is larger than DeepLocator, which shows the standard deviation of CNN's accuracy is larger than DeepLocator. Based on the above results, DeepLocator is more robust and performs better.

If we analyze the results, we observe that some buggy files given low scores by HyLoc can be predicted by DeepLocator. For example, HyLoc gives one buggy file of Bug 54450 in Project Tomcat Rank 50, which is a relatively low score. However, DeepLocator predicts it as a buggy file because "resource" in bug reports is paired with "context" related words (getContext, configureContext, processContextConfig) in source files many times. Another example is Bug 399401 in Project Eclipse UI. The rank of its buggy file is 38. However, "perspective" is paired with "view" and "visible" related words (getViewLayout, isPartVisible), so DeepLocator can relate them by training on older bug reports if these pairs have ever appeared.

4.3. DEEPLOCATOR

Table 4.3: Accuracy of two models.

Project	Model	P (%)	R (%)	F (%)	F_max (%)	F_min (%)
AspectJ	DeepLocator	77.37	80.23	79.92	81.21	77.94
	CNN	57.83	53.49	55.57	60.42	52.79
Eclipse UI	DeepLocator	82.37	84.58	84.13	85.96	79.19
	CNN	55.35	61.62	59.29	67.17	49.22
JDT	DeepLocator	83.43	79.92	82.28	85.41	78.48
	CNN	58.79	61.46	60.85	65.49	54.23
SWT	DeepLocator	82.33	84.98	83.61	84.92	79.95
	CNN	73.37	75.49	73.84	76.39	69.98
Tomcat	DeepLocator	83.35	87.21	85.04	86.29	83.13
	CNN	58.01	58.62	58.39	63.59	52.81

Therefore, even tokens in source files are mismatched with words used in bug reports, DeepLocator can also relate them, which indicates that DeepLocator can learn correlations between them if they appear frequently as pairs.

RQ4: Impact of the Size of Training Dataset on Accuracy.

Figure 4.9 shows the average F of both projects (Project Eclipse UI and JDT) by using different training sets. We observe that initially adding more training data improves the F values. But when more data is involved, especially when more older bug reports are used, the accuracy decreases. This is because older bugs introduce more interference to the model, which also confirms the necessity of using bug-fixing recency as important features.

4.3.3 Discussion

4.3.3.1 Effectiveness of rTF-IDuF

In this section, we analyze the difference between TF-IDF and rTF-IDuF. Some common words may have special meanings in different collections. And bug reports for different projects are usually different in terms of writing style and

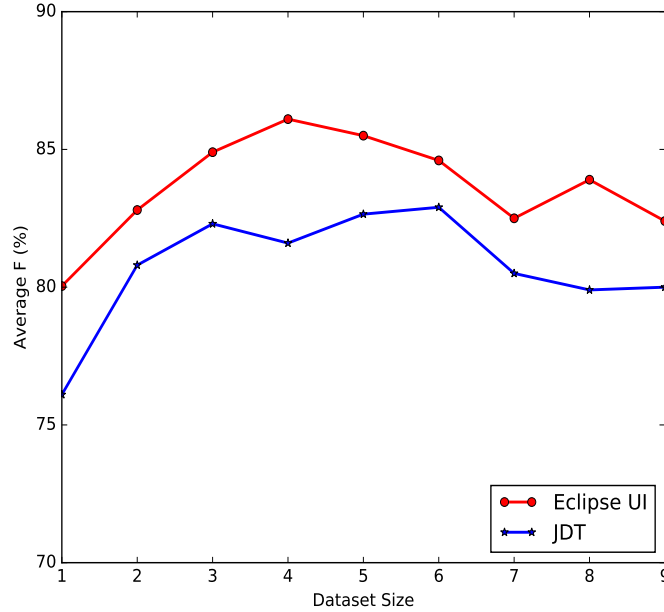


Figure 4.9: Performance of different training set sizes.

content. It is therefore not appropriate to consider them in the same perspective. For example, "aspect" exists in both Project AspectJ and Eclipse UI. There are many "aspect" in AspectJ and most of them represent the name of this project. But there are few "aspect" in Eclipse UI and they denote important information just as shown on the right of Figure 4.10. In other words, "aspect" in Bug 201616 of Project Eclipse UI should be more important than in Bug 415266 of Project AspectJ, which implies the term weight of "aspect" in Bug 201616 should be larger than in Bug 415266.

We calculate the term weights of "aspect" using TF-IDF and rTF-IDuF. TF-IDF uses the whole corpus of available bug reports, while rTF-IDuF just considers the bug reports from the same project. The term weights of "aspect" computed by TF-IDF is 7.36 and 3.09 respectively for Bug 415266 in AspectJ and Bug 201616 in Eclipse UI, while rTF-IDuF is 1.48 and 6.99 respectively. They are different, which validates that the word "aspect" has different importance in different projects.

AspectJ Bug ID: 415266 Summary: Bug 415266 LTW not working when JMX is enabled. Description: When I enable JMX remote management on a JVM along with AspectJ load-time weaving (LTW), our <code>Aspect</code> doesn't appear to get woven in. This are the JVM arguments ...	Eclipse UI Bug ID: 201616 Summary: Bug 201616 [Min.Max] Need to be able to change the orientation of a minimized stack. Description: We currently use the <code>aspect</code> ratio of the existing view stack to set the orientation to be used when showing the view as fast views. While this works in most cases there are enough ...
---	--

Figure 4.10: Bug reports that contain "aspect" in Project AspectJ and Project Eclipse UI.

When considering the whole corpus using TF-IDF, the term weight of "aspect" in Bug 201616 (3.09) is less than in Bug 415266 (7.36), which is not consistent with above observations. Actually, there are many words like "aspect" that have different meanings in different collections. Therefore, rTF-IDuF which considers the "user" collection and not the entire corpus is appropriate, practically feasible and effective.

4.3.3.2 Why does DeepLocator work?

The major challenge of bug localization is the semantic gap between bug reports and source code files. Textual similarity used in most existing techniques [133][56][145][50] is based on the term frequency rather than semantic information of words and phrases. Unlike textual similarity, DeepLocator correlates bug reports to the corresponding buggy files based on a deep understanding of semantics.

Abstract Syntax Tree (AST) is applied to represent syntax and extract programming patterns of source code, which makes the best of the information in source code. Then DeepLocator uses word embedding to map words into semantic vector space where similar words are close to each other. So embedded word vec-

tors containing the information of semantic similarities benefit DeepLocator a lot. What's more, unlike the studies in [133][134] using a linear weighted sum of features, DeepLocator uses an enhanced CNN proposed in this study to correlate bug reports to buggy files including both linear and nonlinear relationships. Besides, CNN model performs well in the semantic parsing field because of the convolving filters [51], which helps DeepLocator to extract hidden semantic information between bug reports and source files.

4.3.3.3 Threats to Validity

The experimental results demonstrate the feasibility of DeepLocator, however, we do acknowledge there are still some potential threats to validity of our approach and experiments. Firstly, the proposed approach could be affected by stemming and removal of stop words process, which will be investigated in future. Secondly, the analysis on the choice of filter size (RQ1) is related to the writing style of developers. For example, if developers in a project team prefer to use very long phrases to express bug reports, the filter size should also be longer. We analyze the results of the dataset and provide a general conclusion. This conclusion may be inadequate. Thirdly, we implement HyLoc according to the algorithm provided by [56] and obtain similar results. But the results are not perfectly the same. Therefore, we choose the best ranked results to compare with our results during the results analysis. Finally, all dataset used are obtained from Java projects. The results may not generalize to other projects written in other languages. In the future, we intend to improve the model and use this model in other projects written in different languages.

4.4 DeepLoc

To further improve the performance of bug localization, we propose DeepLoc, a novel deep learning-based model that makes full use of semantic information. DeepLoc is composed of an enhanced convolutional neural network (CNN) that considers bug-fixing recency and frequency, together with word-embedding and feature-detecting techniques. DeepLoc uses word embeddings to represent the words in bug reports and source files that retain their semantic information, and different CNNs to detect features from them. We conduct empirical experiments on five project datasets to show the effect of the enhanced CNN on DeepLoc and evaluate DeepLoc’s performance by comparing it to four state-of-the-art bug localization approaches: DeepLocator, HyLoc, LR+WE, and BugLocator. All these competitors were proposed to localize buggy files for bug reports, which were based on the textual similarity rather than semantic information of the words and phrases.

The main contributions of this section are the following:

- We transform all code tokens from source files into vectors using word-embedding techniques for bug localization.
- We develop DeepLoc, which consists of word embedding, feature detection, and a proposed enhanced CNN. The enhanced CNN uses important bug-fixing experience (bug-fixing recency and frequency) and semantic information that can be extracted by a conventional CNN from bug reports and source files.
- A set of experiments are conducted to validate DeepLoc’s feasibility and effectiveness. We make our dataset publicly available and provide the tool used to construct the dataset².

²<https://github.com/yanxiao6/BugLocalization-dataset>

In this work, we enhance our previous model, DeepLocator, presented in Section 4.3. DeepLocator first pre-treats bug reports and source files using revised TF-IDuF and Abstract Syntax Tree (AST) detection, respectively. The same word-embedding technique (word2vec) is then applied to transform the preprocessed words from bug reports and source files into vectors. Later, the features of these vectors are extracted by one CNN. However, the use of the same word-embedding technique and CNN to deal with bug reports and source files ignored the differences between them and thus limited the performance of DeepLocator.

Although the motivation for this work is somewhat derived from Section 4.3, the fundamental differences between the two studies are as follows. 1) This section uses two different word-embedding techniques (Sent2Vec and combined word2vec) to convert bug reports and source files, respectively, into vectors. 2) All code tokens are transformed in this section, instead of some AST nodes parsed from the source files in [128] discussed in last section. 3) Two CNNs are used to extract features from the vectors of bug reports and source files before they are supplied to the proposed enhanced CNN. 4) We report better results and new analyses and show the visualization of the learned vectors to demonstrate the effectiveness of the word-embedding techniques for bug localization.

The remainder of this section is structured as follows. Section 4.4.1 gives a detailed description of the proposed model. The experimental results are presented in Section 4.4.2 and the discussion is presented in Section 4.4.3. The threats to validity are discussed in Section 4.4.4.

4.4.1 Model Design

In this section, we describe DeepLoc, a deep learning-based model that automatically localizes buggy files for bug reports. DeepLoc uses two conventional CNNs to separately detect features from the vectors of bug reports and source

code files converted using two different word-embedding techniques and an enhanced CNN for bug localization. Figure 4.11 provides the overall workflow of DeepLoc. In the training phase, bug reports and source files are first transformed into vectors using two different word-embedding techniques (Sent2Vec [87] and weighted average word2vec [47]). The corresponding vectors are then fed into two conventional CNNs to detect their features. Finally, the features of the bug reports and source files are concatenated as the input to the enhanced CNN, along with corresponding labels (buggy or not buggy) and the bug-fixing histories. The training stage is conducted offline. When we receive new bug reports, the trained model will efficiently utilize the bug reports to obtain the locations of buggy files. The details of DeepLoc are discussed in the following sections.

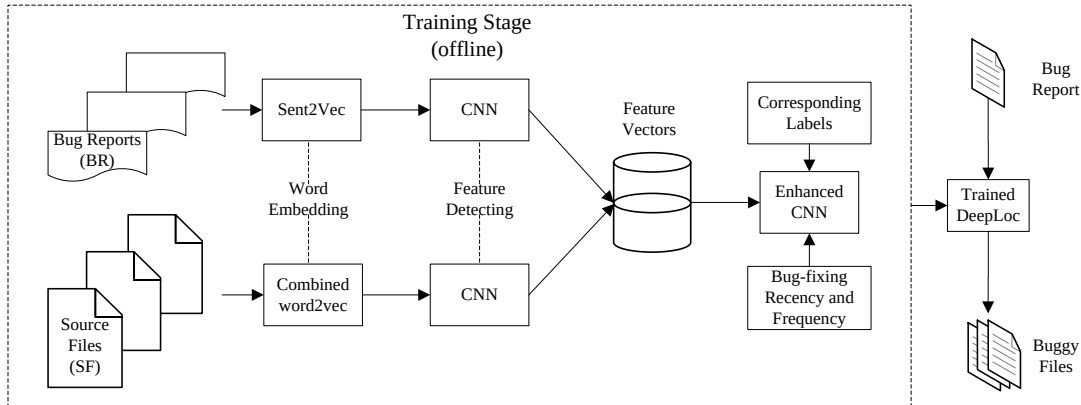


Figure 4.11: The overall workflow of DeepLoc.

4.4.1.1 Data Pre-processing

Developers often combine words to create a new “word” when defining classes or methods in source code files, which can also be mentioned in bug reports. Thus, we preprocess combined words in both bug reports and source files. According to the CamelCase Naming Convention [9], combined words can be split into separate real words based on capital letters. For example, “WorkbenchActionBuilder” is split

into “Workbench”, “Action” and “Builder”. Note that some special capital words, such as URI (Uniform Resource Identifier [73]), IDE (Integrated Development Environment [109]), etc., should not be separated. Finally, we change all capital letters to lowercase letters, to adhere to the rules of word embedding.

4.4.1.2 Word Vectors Generation

Words themselves cannot be directly input into a CNN [141]. Thus, the pre-treated words must be embedded into vectors. Two variants of word-embedding techniques are adopted in this section to transform words in bug reports and source files into vectors that retain the words’ semantics.

As shown in Table 4.1, bug reports usually contain summaries and descriptions that comprise many words. If all words are transformed into vectors as input for CNNs, the model will be difficult to train due to the redundant input. In addition, a large amount of memory is needed to store these vectors. Fortunately, bug reports are written in natural languages and consist of sentences, so we can convert each sentence in bug reports into a vector instead of transforming each word into a vector. Summaries are always written as one sentence containing condensed and abbreviated information. It is difficult to retain important information if summaries are regarded as sentences. However, descriptions usually contain many sentences and some redundant information. So, to distinguish summaries and descriptions, we convert summaries into vectors based on words using a word2vec technique with the Skip-gram model [76] and transform descriptions into vectors based on sentences using Sent2Vec [87]. Figure 4.12 illustrates an example of transforming descriptions in bug reports into vectors using Sent2Vec. Each sentence in a description is transformed into a k -dimensional vector. The dimension of word vectors of each description in a bug report is $n_S \times k$, where n_S is the number of sentences in each description and k is the dimension of word embeddings. Sent2Vec is

an extension of CBOW [76] to compose sentence embeddings. Unlike in CBOW, the context words running over the corpus in Sent2Vec are entire sentences rather than fixed-length context words, which meets the requirements of our model. For experimentation purpose, we have evaluated that the adopted strategy has a similar or even better performance with less memory and computation cost than other kinds of strategies (e.g., only word2vec).

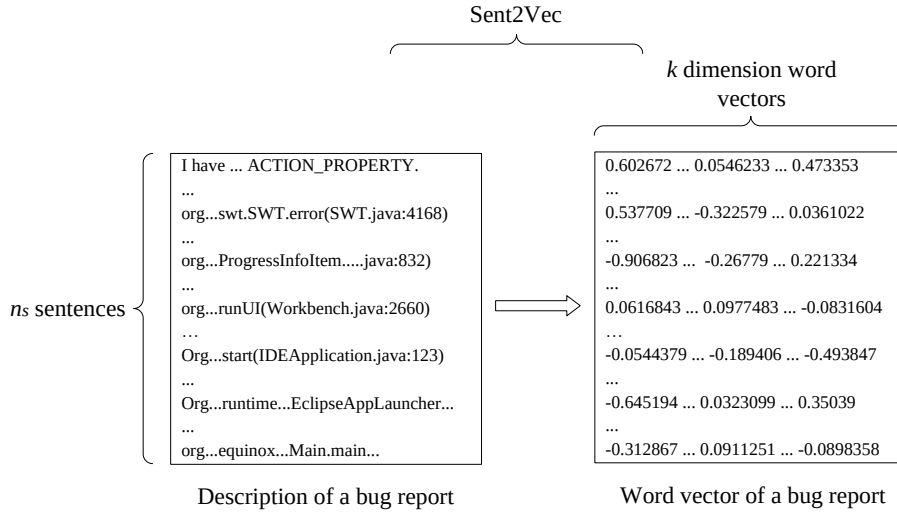


Figure 4.12: An example of transforming descriptions in bug reports into vectors using Sent2Vec.

Source files are composed of code tokens that are similar, but not identical, to natural languages [126]. Some keywords (e.g., `int`, `public`) are frequently presented in source code that may dampen the performance of Sent2Vec. In addition, it is difficult to represent source code in sentences. Thus, Sent2Vec is not good for converting source files into vectors. Importantly, the average word embeddings of words in a sentence have been proven to be efficient for obtaining sentence embeddings [47]. To dampen the effect of frequent keywords in the source code, this section adopts weighted average word embeddings based on the revised TF-IDuF discussed in Section 4.1.1 to obtain the vectors of each line in source code

as shown in the following equation:

$$V_l = \frac{1}{n_l} \sum_{i=1}^{n_l} w_i v_i = \frac{1}{n_l} wv \quad (4.16)$$

where in the l_{th} line in a source file, $V_l \in \mathbb{R}^{1 \times k}$ is the vector of the l_{th} line of code, w_i ($w \in \mathbb{R}^{1 \times n_l}$) is the rTF-IDuF term weight of the i_{th} word in the l_{th} line of code, and v_i ($v \in \mathbb{R}^{n_l \times k}$) is the word vector of the i_{th} word in the l_{th} line of code, in which n_l and k are the number of words in the l_{th} line and the dimension of the word vector, respectively. It is better to use the right vectorized form during implementation, to accelerate the computational process. The word vector (v) is obtained using the word2vec technique with the Skip-gram model [76]. Some words may be absent in the set of pretrained words in word2vec. These words are initialized randomly and fine-tuned during training [51]. Ye et al. [134] verified that word embeddings trained on a Wiki corpus and a project-specific corpus (Eclipse and Java) had similar performance. However, a Wiki corpus has a greater vocabulary and more words (96 times and 548 times the size of a project-specific corpus, respectively), which is beneficial for deep learning models [147]. Thus, the pretrained word2vec used in our model is based on a Wiki corpus.

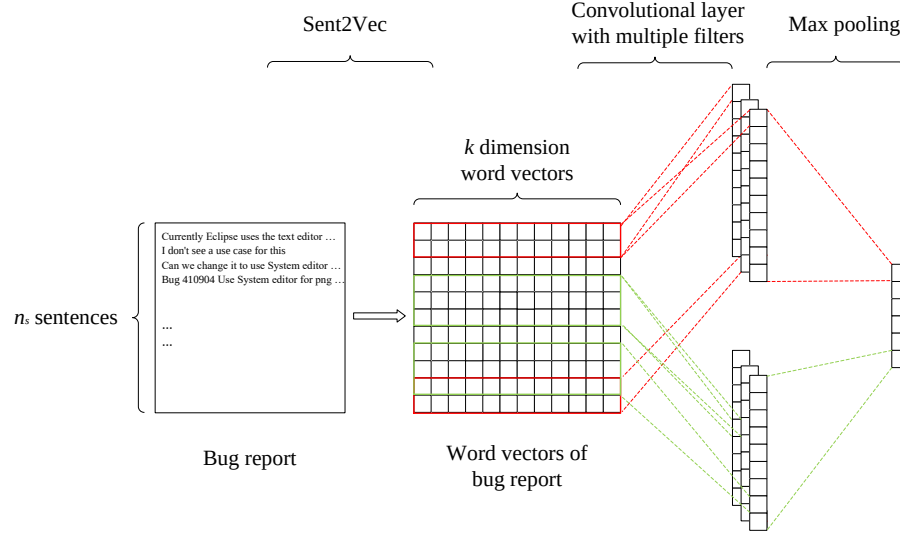
4.4.1.3 Feature Detecting

After converting bug reports and source files into a set of vectors, the CNN model is applied to extract their features. Max-over-time pooling [52] is used when constructing a CNN. We first find the maximum number (n_S) of sentences from each bug report and the maximum number (n_L) of lines from each source file. They are then padded by zeros to be separately same with their maximum lengths [61]. Figure 4.13 illustrates the process of feature detection from bug reports and source files using a CNN, in which k is the vector dimension of each

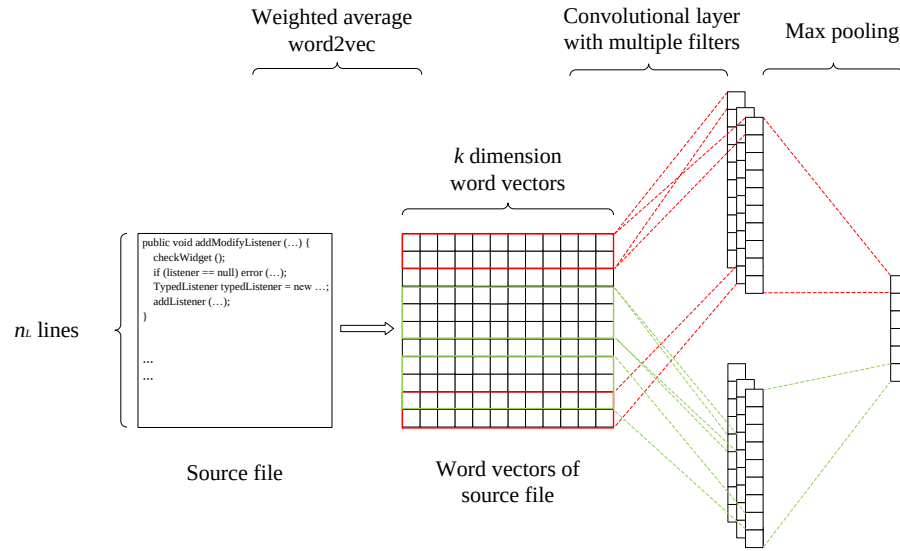
sentence/line. The two CNNs have similar architecture, containing one convolutional layer with multiple filters followed by a max pooling layer [52]. We will evaluate how to choose the size of filters, the number of filters, and the number of convolutional layers when answering RQ1 in Section 4.4.2.3. The size of filters in the max pooling layer is $(n_S - \text{the height of the filter size} + 1) \times 1$ for bug reports and $(n_L - \text{the height of the filter size} + 1) \times 1$ for source files. The size of the feature vectors extracted from a bug report is *the number of filter sizes * the number of filters for each filter size*. The size of the feature vectors extracted from a source file is also *the number of filter sizes * the number of filters for each filter size*. We use different filter size in the max pooling layer to ensure that the final feature vectors extracted from a bug report and source file have the same size so that they can be concatenated in parallel as the input to the subsequent enhanced CNN.

4.4.1.4 Enhanced CNN Training and Prediction

Now we use the enhanced CNN proposed in 4.2 to learn the relationship between bug reports and corresponding buggy files. The feature vectors extracted from bug reports and those from source files are concatenated in parallel into 2-dimensional feature vectors as the input to the enhanced CNN. The concatenation process is shown in Figure 4.14. The convolutional layer in the enhanced CNN uses filters with the size of 2×2 to learn the correlated relationships between bug reports and source files by convolving the input of the enhanced CNN that is a concatenation of the feature vectors extracted from bug reports and source files. The size of filters in the max pooling layer is $2 \times (\text{the number of filter sizes} * \text{the number of filters for each filter size})$. The enhanced CNN was used to correlate bug reports to buggy files including both linear and nonlinear relationships because of its convolutional operation and fully connected layer. Other fac-



(a) Detecting features from bug reports



(b) Detecting features from source files

Figure 4.13: The architecture of feature detection.

tors (bug-fixing recency and frequency) are included in the enhanced CNN using Equation (4.12).

All three CNNs use max-over-time pooling and are regularized by batch normalization to speed up the training process. DeepLoc is trained using the Adam

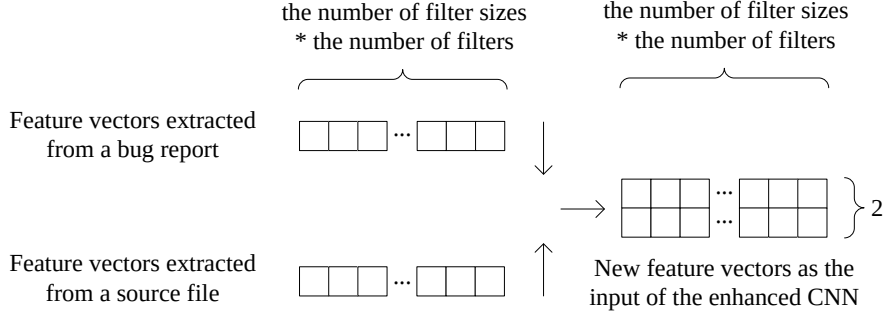


Figure 4.14: The concatenation process.

(Adaptive Moment Estimation) Optimizer [53], which uses an adaptive learning algorithm. Compared to Adadelata [137], which retains an exponentially decaying average of past squared gradients, Adam can also store the exponentially decaying average of past gradients. In addition, Adam has shown better performance in large empirical investigations than other algorithms that automatically adjust learning rates [53].

After training, the bug reports are related to buggy files rather than matched by textual similarity. When a new bug report appears, it is paired with each source file and then input into the trained DeepLoc to predict buggy files.

4.4.2 Experiments

Several experiments are conducted to evaluate DeepLoc’s performance. We describe the datasets used, the experimental setup and the performance metrics in the first part, followed by the listed research questions, and the results in the last part.

4.4.2.1 Experimental Settings and Evaluation Metrics

Because some source files may have been modified or even deleted, we may fail to find the original mapped source files for an old bug without before-fix versions.

Thus, we collect a dataset (Table 4.1) with before-fix versions of source files and relate them to each bug based on publicly available mappings of bug reports and the corresponding commit history provided by [133].

The chronologically sorted bug reports for each project are split into 60% as the training set (oldest bugs), 20% as the validation/development set, and 20% as the test set (newest bugs). Large numbers of source files exist for a given bug report, and it is infeasible to involve them all in the training set because of the necessary training time and memory. For fair comparison with other state-of-the-art techniques, we use a procedure similar to that in [56, 133]. In training, for each bug report, we rank the cosine similarity of the word vectors between the bug report and all of the source files and then select the top 300 irrelevant files as the negative samples, which implies that the actual buggy files and other 300 least similar files are together with each bug report to be set as the training set. In the validation and testing set, each bug report is paired with all source files. We use tensorflow to construct the model, and all experiments are run on a server with CPU Intel Xeon CPU E5-4620 2.20 GHz (32 cores), 128 GB RAM.

To evaluate DeepLoc’s performance, three metrics are adopted: Accuracy@ k , Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP). These metrics have been widely used in existing bug localization studies [56, 133, 134, 145]. Their definitions are expressed as follows.

- *Accuracy@ k* measures the percentage of bug reports for which at least one real buggy file is located within the top k rank.
- *MRR* is the mean of the Reciprocal Rank that accumulates the inverse of the position of the first correctly-located buggy file for each bug. For a set

of Q bug reports, the MRR in this thesis is computed as follows:

$$MRR = \frac{1}{Q} \sum_{i=1}^Q \frac{1}{first_i} \quad (4.17)$$

where $first_i$ denotes the position of the first correctly-located buggy file for the i_{th} bug report.

- *MAP* can measure performance when a query has multiple relevant documents. It is more suitable in the field of bug localization because a bug report may have more than two buggy files on average (as shown in Table 4.1). MAP is the mean of the Average Precision (AvgP) values of Q bug reports, each of which is the average of the precision values for a bug report. The formulation of MAP for bug localization is as follows:

$$MAP = \frac{1}{Q} \sum_{i=1}^Q AvgP(i) \quad (4.18)$$

$$AvgP(i) = \sum_{j=1}^M \frac{(T(j)/j) \times ind(j)}{B(i)} \quad (4.19)$$

where M is the maximum position of correct buggy files for the i_{th} bug report located using the bug localization technique, $ind(j)$ indicates whether the file located in rank j is the correct buggy file ($ind(j) = 1$) or not ($ind(j) = 0$), $B(i)$ stands for the number of buggy files for the i_{th} bug report, and $T(j)$ denotes the number of buggy files in the top j .

The higher the Accuracy@k, MRR, and MAP values, the better the performance of the bug localization technique.

4.4.2.2 Research Questions

Specifically, we aim to answer the following questions.

RQ1: What effect do the model settings have on DeepLoc? Before building DeepLoc, we need to analyze three settings using the validation set: the size of filters, the number of filters, and the number of convolutional layers.

RQ2: What effect does the enhanced CNN have on DeepLoc? We then evaluate whether the enhanced CNN improves DeepLoc’s accuracy against the conventional CNN. The only difference between the two CNNs is that the enhanced CNN considers bug-fixing recency and frequency, but the conventional CNN does not. Furthermore, we examine their time overheads for prediction.

RQ3: How good is DeepLoc’s performance compared to state-of-the-art techniques? To validate DeepLoc’s effectiveness and efficiency, four state-of-the-art approaches are used as competitors:

- *DeepLocator* proposed in our previous work [128] uses CNN to extract features from bug reports and source files that are pre-treated using rTF-IDuF and AST.
- *HyLoc* proposed by Lam et al. [56] combines deep learning with information retrieval (IR) technique.
- *LR+WE* proposed by Ye et al. [134] enhances the previously proposed learning-to-rank (LR) model [133] with word embedding (WE).
- *BugLocator* proposed by Zhou et al. [145] uses a revised Vector Space Model (rVSM) to measure the textural similarity between bug reports and source files.

Both **RQ2** and **RQ3** use a testing set for evaluation. There are 50 instances in each batch.

4.4.2.3 Results and Analyses

RQ1: Model Performance Under Different Model Settings.

We first evaluate how to choose the size of filters using the validation set. The number of filters is set as constant (100) as a control variable. Figure 4.15 reports the MAP values of each dataset with the filter size ranging from $1 \times k$ to $10 \times k$ (where k is the width of word vectors). The results show that the best filter size for the five datasets is $3 \times k$ (AspectJ), $6 \times k$ (Eclipse UI), $4 \times k$ (JDT), $3 \times k$ (SWT), and $4 \times k$ (Tomcat) respectively. According to the empirical analysis of Zhang et al. [141], using multiple filter sizes near the best single size produces the best results. So in our experiments, the filter sizes are set as $(2 \times k, 3 \times k, 4 \times k)$, $(5 \times k, 6 \times k, 7 \times k)$, $(3 \times k, 4 \times k, 5 \times k)$, $(2 \times k, 3 \times k, 4 \times k)$, and $(3 \times k, 4 \times k, 5 \times k)$ for the five datasets (AspectJ, Eclipse UI, JDT, SWT, and Tomcat). We

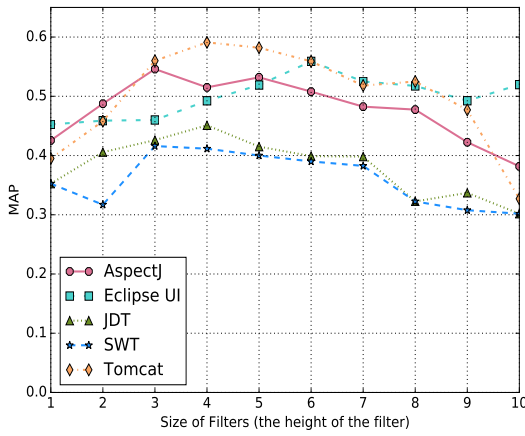
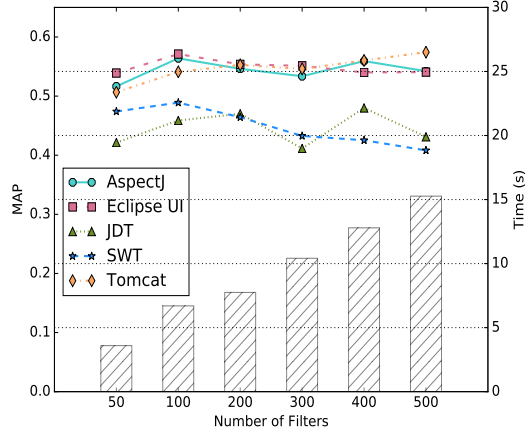


Figure 4.15: Performance of different filter sizes. Figure 4.16: Performance of number of filters.

fine-tune the number of filters for each dataset given the formerly determined filter size. We consider values $\{50, 100, 200, 300, 400, 500\}$. Figure 4.16 represents the MAP values (line chart) and the number of filters for the five datasets. We also provide the average training time per batch (bar chart) for the five datasets to

help us choose the appropriate number of filters balanced between the results and the time consumption. The results show that the greater the number of filters, the higher the computation time. However, the MAP values do not always increase because more filters may lead to overfitting. After weighing the trade-offs, we set the number of filters as 100 for the whole dataset.

Next, we compare the MAP values and the average time consumption for each batch of the CNN, using one convolutional layer and two convolutional layers. When using a two-layer CNN, the features between words are convolved again, which is able to extract deeper semantic information. Table 4.4 demonstrates that the performance of the two-layer CNN is slightly better than that of the one-layer CNN, but the time consumption is much higher. Thus, DeepLoc uses a one-layer CNN.

Table 4.4: MAP and time consumption for different numbers of layers.

Project	One-layer		Two-layer	
	MAP	Time (s)	MAP	Time (s)
AspectJ	0.55	6.36	0.58	212.18
Eclipse UI	0.56	6.92	0.57	217.15
JDT	0.48	6.38	0.49	212.86
SWT	0.49	6.65	0.48	213.56
Tomcat	0.57	6.74	0.59	213.91

RQ2: The enhanced CNN’s effects on DeepLoc.

In this section, we use the testing set to compare DeepLoc’s performance using enhanced CNN and conventional CNN. We find that both the MAP and MRR of the enhanced CNN are higher than those of the conventional CNN, with a 10.87% to 13.4% higher MAP (as shown in Figure 4.17). Because CNN does not have long-term memory, it cannot learn the features of bug-fixing recency and frequency, which are included in the enhanced CNN.

We report the time overhead of DeepLoc using the enhanced CNN and the

4.4. DEEPLOC

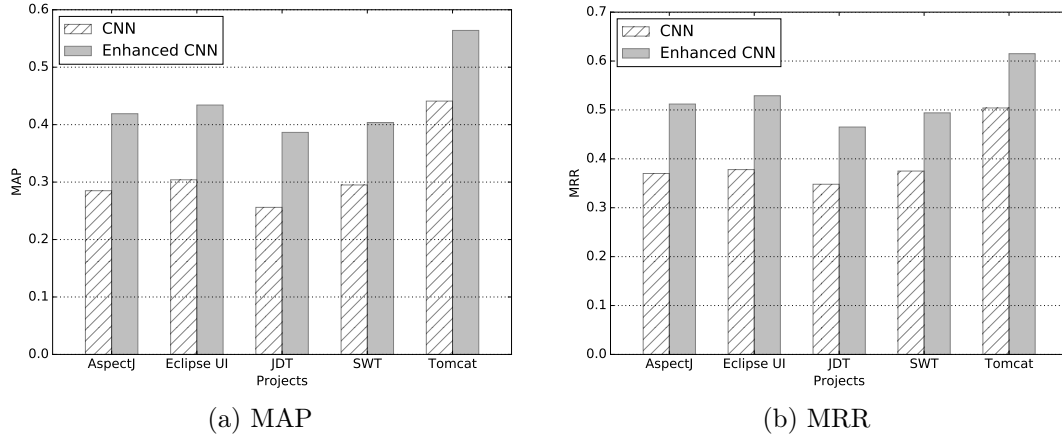


Figure 4.17: MAP and MRR of conventional CNN and enhanced CNN for five projects.

conventional CNN. Because more weights in DeepLoc must be adjusted in the training phase than in the CNN, DeepLoc’s average time consumption per batch in training is slightly higher (about 0.03 seconds) than the CNN’s. Table 4.5 presents the average prediction time for one bug report by DeepLoc using the enhanced CNN and the conventional CNN, and the existing deep learning-related model (HyLoc) [56]. The first two rows demonstrate that DeepLoc’s average prediction times using the enhanced CNN and the conventional CNN are very similar.

Table 4.5: Prediction time (seconds) per bug report for three models.

Prediction Time	AspectJ	Eclipse UI	JDT	SWT	Tomcat
DeepLoc	127.9	93.5	152.0	51.0	36.5
CNN	123.3	95.3	152.3	53.2	36.3
HyLoc	144.0	126.0	198.0	108.0	60.0

RQ3: DeepLoc’s Performance.

Figure 4.18 presents the Accuracy@k results for the four models (DeepLoc, HyLoc, LR+WE, and BugLocator) on Project SWT, with k ranging from 1 to 20. The results show that BugLocator has the worst performance. The difference

between HyLoc and LR+WE is not clear. DeepLoc shows evident improvements, especially from Top 1 to 11, which is more significant for bug localization because developers prefer to search only a few source files to find buggy files. For example, DeepLoc achieves Accuracy@k of 39.0%, 65.7%, and 77.1% for $k = 1, 5, 10$, respectively. That is to say, it can correctly locate buggy files for 39.0% of bug reports when recommending only one source file to developers, and it can fix 77.1% of bug reports when ten source files are recommended. In comparison, HyLoc achieves Accuracy@k 33.4%, 71.1%, LR+WE achieves 34.0%, 71.0%, and BugLocator achieves 22.3%, 51.7% for $k = 1, 10$, respectively. Using a Mann-Whitney-U-Test [71], the Accuracy@k results of HyLoc and LR+WE are not significantly different, but DeepLoc significantly ($p < 0.05$) outperforms both of them.

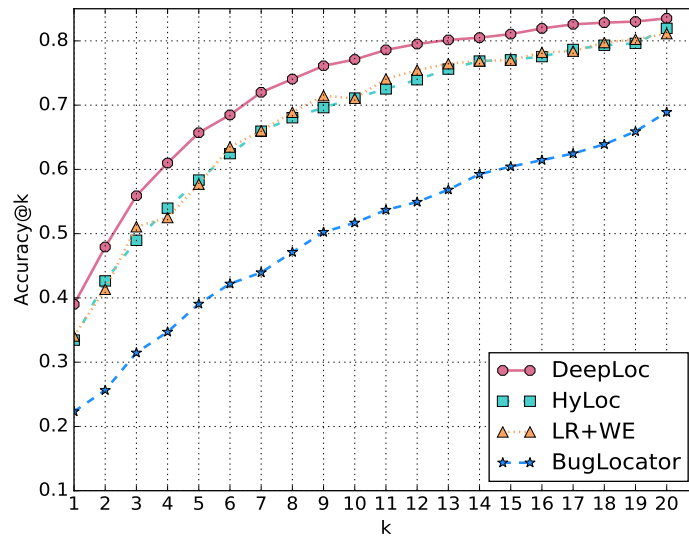


Figure 4.18: Accuracy@k of the four models on Project SWT.

Table 4.6 lists the Accuracy@1,5,10, MAP, and MRR values of the five models using the test set for five projects. Again, DeepLoc outperforms the other four models for all five projects.

Compared to DeepLocator, DeepLoc achieves 2% to 5% higher Accuracy@1

4.4. DEEPLOC

Table 4.6: Performance comparison.

Project	Model	Accuracy @1	Accuracy @5	Accuracy @10	MAP	MRR
AspectJ	DeepLoc	0.45	0.71	0.80	0.42	0.51
	DeepLocator	0.40	0.66	0.78	0.34	0.49
	HyLoc	0.38	0.65	0.75	0.32	0.48
	LR+WE	0.29	0.58	0.74	0.30	0.45
	BugLocator	0.22	0.46	0.58	0.28	0.36
Eclipse UI	DeepLoc	0.45	0.70	0.79	0.43	0.53
	DeepLocator	0.43	0.66	0.74	0.42	0.51
	HyLoc	0.40	0.64	0.73	0.41	0.51
	LR+WE	0.39	0.60	0.71	0.40	0.46
	BugLocator	0.29	0.50	0.60	0.33	0.38
JDT	DeepLoc	0.43	0.65	0.77	0.44	0.53
	DeepLocator	0.40	0.64	0.73	0.39	0.47
	HyLoc	0.33	0.59	0.69	0.34	0.45
	LR+WE	0.41	0.65	0.75	0.42	0.52
	BugLocator	0.19	0.40	0.51	0.29	0.37
SWT	DeepLoc	0.39	0.66	0.77	0.40	0.49
	DeepLocator	0.36	0.60	0.75	0.39	0.48
	HyLoc	0.33	0.58	0.71	0.37	0.45
	LR+WE	0.34	0.57	0.71	0.38	0.45
	BugLocator	0.22	0.39	0.52	0.27	0.31
Tomcat	DeepLoc	0.54	0.72	0.81	0.56	0.62
	DeepLocator	0.52	0.72	0.80	0.54	0.60
	HyLoc	0.52	0.71	0.78	0.52	0.60
	LR+WE	0.49	0.70	0.76	0.50	0.56
	BugLocator	0.36	0.62	0.71	0.43	0.48

and 1% to 5% higher Accuracy@10. DeepLoc distinguishes bug reports and source files using two different word-embedding techniques and CNNs. All information in bug reports and source files is included and deeper networks are used to further improve the bug localization performance.

In addition, DeepLoc achieves 16% to 24% greater Accuracy@1 than BugLo-

cator. Unlike BugLocator, whose performance for unseen bug reports is unclear due to the only parameter α being tuned on the same dataset for both training and evaluation [145], all of the weights for DeepLoc are learned on the training set, tuned on the validation set, and then used to predict the testing set, whose performance is scalable. At the same time, LR+WE achieves 2% to 16% lower Accuracy@1 than DeepLoc. LR+WE may be limited by the linear weighted sum of features, but DeepLoc does not suffer this problem because of the convolutional networks.

Finally, DeepLoc achieves 2% to 10% higher Accuracy@1 than HyLoc. Because Lam et al. [56] validated HyLoc’s comparable or superior performance against existing approaches [84, 133, 134, 145], we use HyLoc as the main competitor and analyze the results. When analyzing the ranking results, we observe that some buggy files given low scores by HyLoc can be located more precisely by DeepLoc. For example, HyLoc gives one buggy file for Bug 54450 in Project Tomcat Rank 50, which is a relatively low score. However, DeepLoc predicts it as a buggy file by ranking it in fourth place because "resource" in bug reports is paired with words related to "context" (getContext, configureContext, processContextConfig) in source files many times. Another example is Bug 399401 in Project Eclipse UI. The rank for its buggy file given by HyLoc is 38. However, "perspective" is paired with words related to "view" and "visible" (getViewLayout, isPartVisible), so DeepLoc can relate them if these pairs have ever appeared in older bug reports and corresponding buggy files used in its training. Therefore, even when tokens in source files are mismatched with the words used in bug reports, DeepLoc can relate them, which indicates that DeepLoc can learn correlations if they frequently appear as pairs. In particular, DeepLoc’s prediction time, shown in Table 4.5, for one bug report is much lower than HyLoc’s, which indicates that DeepLoc is more suitable in practice.

4.4.3 Discussion

The major challenge for bug localization is the semantic gap between bug reports and source code files. Textual similarity, which is used in most existing techniques [133, 56, 145, 50], is based on the term frequency rather than semantic

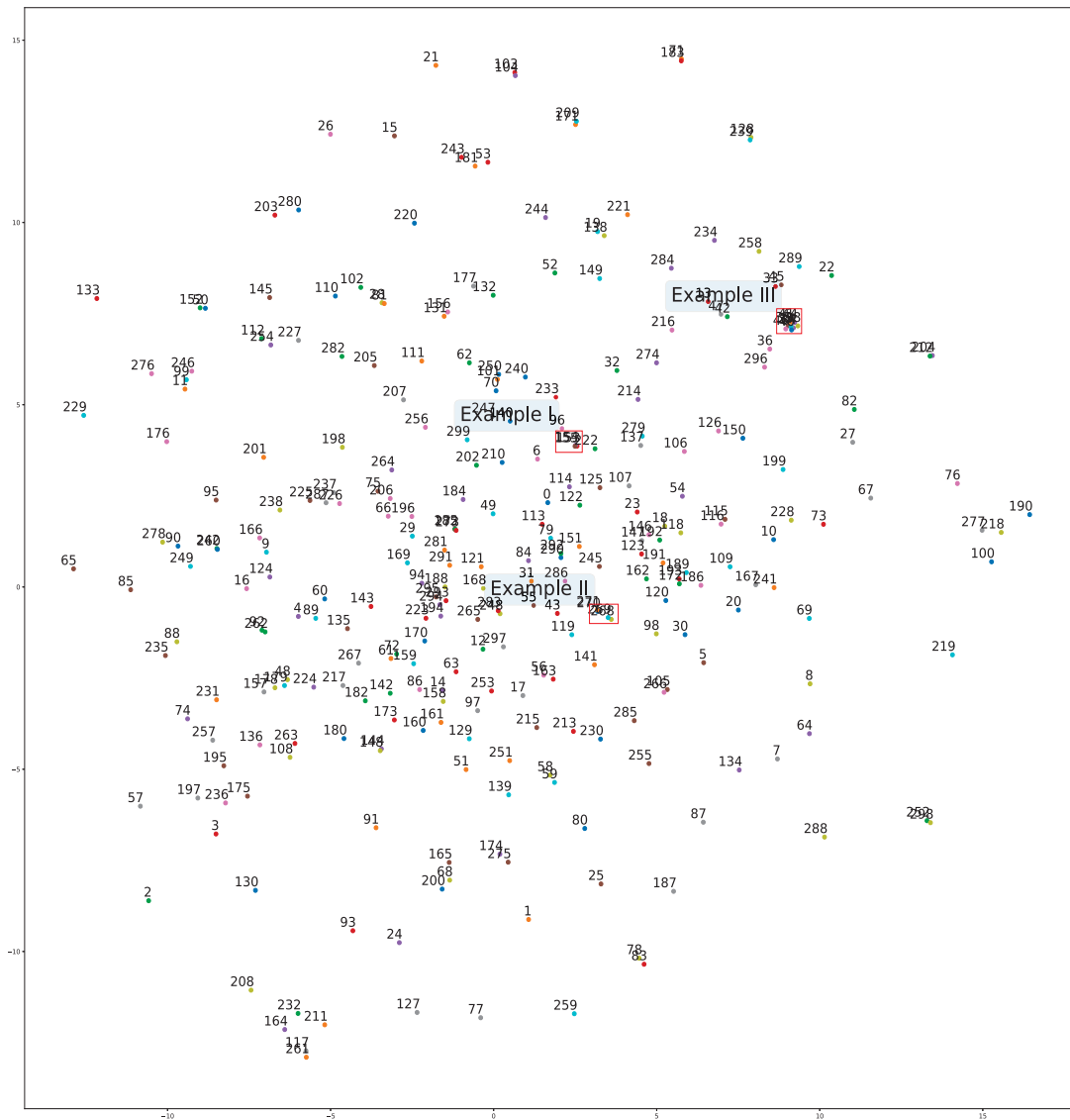


Figure 4.19: Visualization of the learned embeddings of 500 bug reports from Project SWT using t-SNE.

information of the words and phrases. Unlike textual similarity, DeepLoc correlates bug reports to the corresponding buggy files based on a deep understanding of semantics.

DeepLoc uses word-embedding techniques to map bug reports and source files into semantic vector spaces where similar words are close to each other. According to their characteristics, DeepLoc uses two different word embedding-based techniques for bug reports and source files. Figure 4.19 visualizes the learned embeddings of 500 bug reports from Project SWT using t-SNE [110], where a number represents a bug report. Each bug report is transformed into a word vector as discussed in Section 4.4.1.2. These vectors are then projected into 2-D dimension by t-SNE to make the vectors visible as shown in Figure 4.19. The closer two bug reports are in the vector space, the greater the similarity between them. In Example I, the three bug reports are "Unused and breaks compilation on GTK 3.x. Bug 391413 Remove GdkGCValues_sizeof", "Remove GdkVisual_sizeof - not used at all and breaks compilation on GTK 3.x. Bug 391408 Remove GdkVisual_sizeof", and "Method is not used anywhere and breaks compilation on GTK 3.x. Bug 391404 Remove GdkImage_sizeof". They are all related to GTK and the "sizeof" operation. Most importantly, these three bug reports point to a same buggy file, namely OS.java, but the first bug report still contains other buggy files. The contents of the bug reports in Example II, "Bug 369228 Kill pre GTK 2.4 leftovers from Tree", "Bug 369227 Kill pre GTK 2.4 leftovers from List", "Bug 369226 Cleanup pre GTK 2.4 leftovers in Table", and "Bug 369225 Cleanup pre GTK 2.4 leftovers", also share similar operations. In addition, the first three bug reports share the same buggy files. The bug reports in Example III also share the same or similar words like "widget", "unused 2.10 version guard" and so on. The aggregation of some points in Figure 4.19 indicates that these bug reports may share similar meanings or even point to the same buggy files. Therefore, the

word-embedding technique is useful for converting text into vectors that retain semantic information, and can help to localize buggy files. However, it is still insufficient, which is why it requires consequent models to detect features and learn correlations between bug reports and source files.

The embedded vectors that contain information about semantic similarities greatly benefit DeepLoc. The CNN model performs well in the semantic parsing field because of the convolving filters [51], which helps DeepLoc to extract hidden semantic information from bug reports and source files. DeepLoc applies two CNNs to extract features from bug reports and source files and then uses an enhanced CNN to correlate bug reports and buggy files including both linear and nonlinear relationships, which makes it different from the studies in [133, 134], which use a linear weighted sum of features. Bug-fixing history is also included in DeepLoc, to make it more powerful.

4.4.4 Threats to Validity

The experimental results demonstrate DeepLoc’s feasibility, however, we acknowledge some potential threats to the validity of our approach and experiments. Following the suggestions of Wohlin et al. [123], we discuss threats to internal validity, construct validity, external validity, and statistical conclusion validity.

4.4.4.1 Internal Validity

The proposed approach converts each sentence in a bug report into a vector, which could be affected by stemming and removal of stop words process. This potential will be investigated in a future study. Both bug reports and source files are transformed into vectors based on word-embedding techniques. These techniques make texts from bug reports and source files into adequate input for CNNs, retaining their semantic information and saving memory space. However,

the performance of the proposed approach relies to some extent on the ability of word-embedding techniques. It would be best to test these techniques before adding to the proposed model. Improving these techniques will also help to enhance our model. We leave this for future studies. Finally, the analysis on the choice of filter size (RQ1) is related to the writing style of the developers. For example, if the developers in a project team prefer to use very long phrases to express bug reports, the filter size should also be long. We analyze the results of the dataset and provide a general conclusion, which may not be adequate.

4.4.4.2 Construct Validity

As shown in Table 4.1, the dataset for bug localization is imbalanced. Because the focus of this section is not a data imbalance problem, we selected a simple sampling strategy. A better technique for solving a data imbalance is important for bug localization, which will be investigated in detail in a future study. On the other hand, we split the dataset into a training set (60%), a validation set (20%), and a testing set (20%). This approach is adopted in most machine learning-related papers [18, 89, 81] but not in bug localization-related papers [50, 145, 133]. The effects of various splitting strategies on deep learning-related bug localization techniques is still unknown and is worthy of investigation in further studies.

4.4.4.3 External Validity

Because the codes for HyLoc, LR+WE, and BugLocator are not published, we implemented them according to the algorithms provided by [56, 134, 145] and obtained similar results. However, the results are not identical. Fortunately, Ye et al. [133] and Lam et al. [56] provided their results for the same datasets. We choose the best results for each project comparing our results to theirs. Finally, the datasets used were obtained from Java projects. The results may not be

generalizable to other projects written in other programming languages. In the future, we intend to improve the model for use in other projects written in different programming languages.

4.4.4.4 Statistical Conclusion Validity

In this study, we used a Mann-Whitney-U-Test for the Accuracy@ k analyses. We provided Accuracy@ k results of the four models (DeepLoc, HyLoc, LR+WE, BugLocator) with k ranging from 1 to 20 and applied a Mann-Whitney-U-Test for comparison. However, the sample size is close to the minimum required number of observations in a standard statistical test. We thus admit the experimental results might be subject to a threat to validity.

4.5 Chapter Summary

This chapter first discusses DeepLocator, a deep learning based model that consisted of an enhanced CNN, together with a new rTF-IDuF method and the pre-trained word2vec technique, is proposed to improve the performance of bug localization. The experimental results show that DeepLocator performs better than existing classification models. DeepLocator bridges the semantic gap using Abstract Syntax Tree to parse the syntax of source code, word embedding to obtain semantic similarities and the enhanced CNN to learn the correlations between bug reports and source code.

Secondly, DeepLoc, a deep learning-based model that consists of an enhanced CNN, together with word-embedding techniques and two CNNs for detecting features from bug reports and source files, is proposed to address these challenges and improve the performance of bug localization. Unlike the existing approaches, word-embedding techniques are adapted to convert each sentence of bug reports

and each line of source code into vectors, retaining their semantics in vector space. These vectors are then fed into the CNNs to extract their hidden semantics and features. An enhanced CNN is used to detect the correlation between the feature vectors extracted from bug reports and source code considering the bug-fixing experience (bug-fixing recency and frequency). We provide empirical suggestions on how to construct the model and evaluate DeepLoc’s performance against four state-of-the-art bug localization techniques (DeepLocator, HyLoc, LR+WE, and BugLocator) using more than 18,500 bug reports extracted from five projects. The experimental results show that DeepLoc performed better than conventional CNN (10.87% to 13.4% MAP improvements), which indicates the importance of bug-fixing experience for bug localization. Compared to the other four approaches, DeepLoc achieved higher Accuracy@k, MAP, and MRR, using less computation time. DeepLoc bridges the semantic gap by adapting word embedding to retain the semantics of bug reports and source code, CNNs to parse their syntax, and the enhanced CNN to learn the correlations between bug reports and source code considering bug-fixing experience.

DeepLocator and DeepLoc both use word embedding techniques to retain the semantics of bug reports and source files, CNN to learn the semantic parsing and an enhanced CNN to learn the correlation between bug reports and source files considering bug-fixing experience. The experimental results comparing their performance demonstrate the importance of processing bug reports and source files respectively. On the other hand, the higher Accuracy@k, MAP, and MRR compared with the state-of-the-art bug localization techniques that did not retain semantics and extract semantic parsing information has revealed the importance of semantic information.

Chapter 5

Structural Information Extraction for Bug Localization

Correctly localizing buggy files for bug reports together with their semantic and structural information is a crucial task, which would essentially improve the accuracy of bug localization techniques. To empirically evaluate and demonstrate the effects of both semantic and structural information in bug reports and source files on improving the accuracy of bug localization, this chapter discusses CNN_Forest [129], which has been published in EASE 2018, involving convolutional neural network and ensemble of random forests that have excellent performance in the tasks of semantic parsing and structural information extraction.

5.1 Preliminaries

5.1.1 Random Forest

Random Forest (RF) is an ensemble learning method based on multiple decision trees [11]. As shown in Figure 5.1, RF first extends the original training

set $\{D_1, D_2, \dots, D_n\}$ into k sub-training sets $\{Sub_1, Sub_2, \dots, Sub_k\}$ using random sampling with replacement. The dimension of the extended training set is the same as that of the original training set. Then a decision tree is trained on each sub-training set and a total of k decision trees $\{\text{Decision Tree 1, Decision Tree 2, } \dots, \text{Decision Tree } k\}$ is obtained. The final output is determined by combining the results of the k decision trees, e.g., averaging or voting.

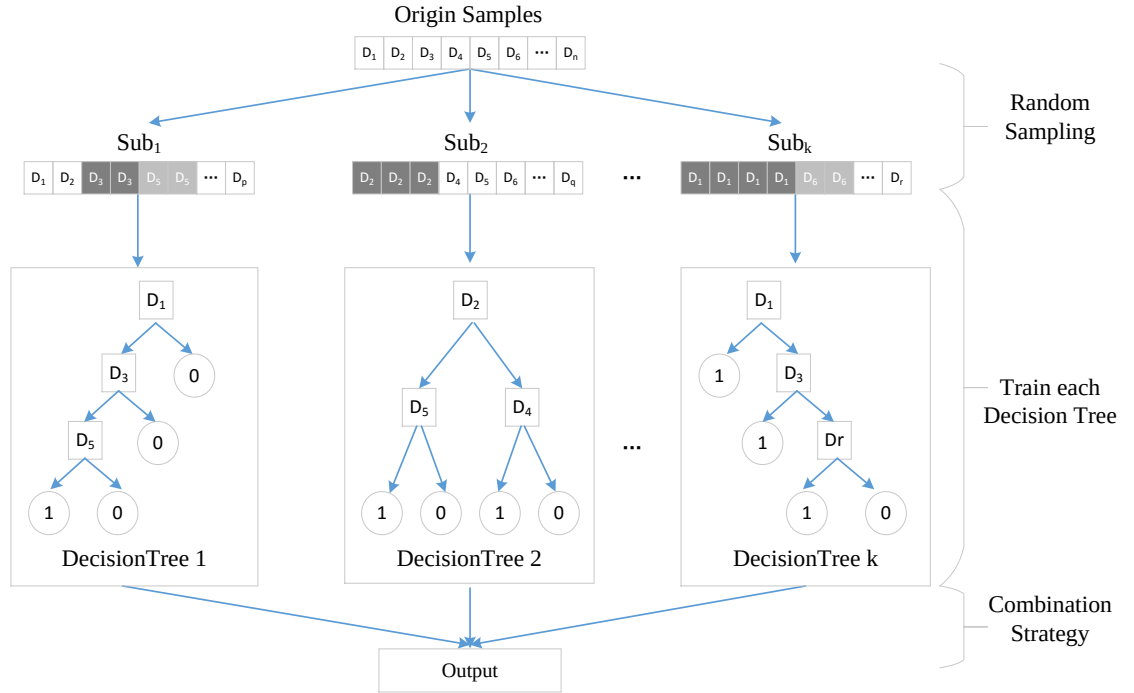


Figure 5.1: The illustration of random forest.

RF introduces randomness into decision trees, so as to obtain the integrated learners with more powerful ability of generalization. The property of randomness is shown in two aspects. Firstly, based on simple random sampling with replacement, the original training set is divided into multiple sub-training sets, which usually contain different duplicate samples. Secondly, during the process of feature selection in decision trees, a subset of all the current features is randomly selected. Then the optimal feature is selected from the sub-feature set as

the root node. By adding these two random properties, the generalization of RF (predictive ability on new data) is significantly improved. However, RF usually converges slowly at a very early stage where there are a few numbers of decision trees. Fortunately, as the number of decision tree increases, RF can converge to lower generalization error (a measure of how accurately an algorithm is able to predict outcome values for previously unseen data). Moreover, RF is simple with low cost and easy to implement, and it has achieved good performance in many practical tasks.

5.2 CNN_Forest

This section discusses CNN_Forest based on CNN and random forest for bug localization. The proposed model first converts the bug reports and source files into word vectors by the word embedding technique to preserve the semantics of them. Due to the differences between natural languages and programming languages, we then employ different techniques to extract features from bug reports and source files. CNN with multiple filters is applied to parse the semantics and extract features from bug reports by adopting the procedure leveraged by the natural language processing techniques [30, 51]. Since source files are composed of code tokens, an ensemble of random forests involving various decision trees is used to extract the structural information from them. After the feature extraction, cascade forest is adopted to extract further features, and the correlated relationships between bug reports and source files are learned benefiting from the alternate cascade structure that is similar to the layer-structure in deep learning. In bug reports, the structural information means the contextual information within each sentence extracted by CNN and that between sentences extracted by the subsequent cascade forest. In source files, the structural information means the contextual informa-

tion within each statement extracted by the ensemble of random forest and the interactions between statements extracted by the subsequent cascade forest.

The main contributions of this section are listed as follows:

- We leverage two different techniques to extract features respectively from bug reports and source files according to the differences between natural languages and programming languages.
- Both semantic and structural information of bug reports and source files are extracted. The ensemble of random forests is applied to detect the structural information from the source code. The alternate cascade forest works as the layer-structure in deep learning to learn the correlated relationships between bug reports and source files.
- A set of experiments is conducted to validate the feasibility and effectiveness of CNN_Forest. The experimental results confirm the significance of applying different techniques to extract features from bug reports and source code.

The remainder of this section is structured as follows. Section 5.2.1 provides a detailed description of the proposed model. Section 5.2.2 presents the experimental preparations and results, followed by the discussion in Section 5.2.3.

5.2.1 Model Design

In this section, we will describe the proposed model, CNN_Forest, a convolutional neural network (CNN) and random forest-based approach for bug localization. Figure 5.2 presents the overall workflow of the proposed approach. As shown in Figure 5.2, the words in bug reports and source files are first transformed into vectors by the word embedding technique to preserve the semantic information.

5.2. CNN_FOREST

Due to the different characteristics of bug reports and source files, their vectors are then processed respectively by CNN with multiple filters and the ensemble of random forests with multi-grained scanning to extract the features and reduce dimensions. After that, these feature vectors are fed into the cascade forest to learn the correlated relationships between bug reports and source files, and to further extract features to obtain the class vectors, whose average values are the final class vectors (output).

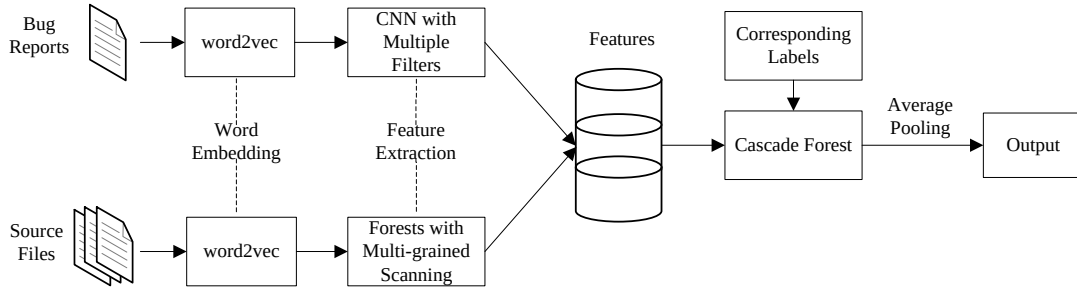


Figure 5.2: The overall workflow of CNN_Forest.

5.2.1.1 Data Preprocessing

Bug reports usually consist of summaries and descriptions but one of them may be vacant. We thus first combine them into one document as the new bug reports. Each text in bug reports and source files is then filtered for punctuation (e.g., transform `gtk_widget_get_window` into `gtk widget get window`) using the Natural Language Toolkit (NLTK) [103] package¹. In addition, some texts are combinations of multiple words, which are also needed to be split into the single word based on the CamelCase Naming Convention [9]. For example, “Workbench-WindowConfigurer” can be split into “Workbench”, “Window” and “Configurer”. Finally, all texts are cast to lower case.

¹<http://www.nltk.org/>

5.2.1.2 Word Embedding

After pre-processing, all the words in bug reports and source files are converted into word vectors by a word embedding technique (word2vec [76]). As reported in the literature [134], the word vectors obtained by word2vec from project-specific and Wikipedia corpus have similar performance for bug localization. Therefore, we use the Skip-gram model pre-trained on the Wikipedia corpus to obtain the word vectors. For words not present in the corpus, we randomly initialize them [51].

5.2.1.3 Feature Extraction

Given bug reports $B = \{b_1, \dots, b_i, \dots\}$ and source code files $C = \{c_1, \dots, c_j, \dots\}$, each bug report $b_i \in \mathbb{R}^{n_{bs} \times n_{bw} \times k}$ and each source file $c_j \in \mathbb{R}^{n_{cl} \times n_{cw} \times k}$ are 3-dimensional vectors after word embedding, where k is the dimension of the word vectors, n_{bs} denotes the maximum number of sentences in bug reports, and n_{bw} denotes the maximum number of words in each sentence, while n_{cl} denotes the maximum number of lines in source files, and n_{cw} denotes the maximum number of words in each line. Next, we will describe how to respectively extract features based on these 3-dimensional vectors from bug reports and source files and simultaneously reduce the dimensions.

The reason of using different strategies to extract features from bug reports and source files are discussed in the following. For example, *path = getNewPath();* *Filef = File.open(path);*, the *path* is used in the next line. The interactions between each statement are very common and can be linked by the same word. This kind of relationship can be represented well by tree-based architecture using the nodes and links between them in the decision tree.

But there are less same words to link such interactions between each sentence in bug reports. It is common in bug reports or natural languages that using some

pronouns to replace same words. Thus the CNN model with semantic parsing properties is a good choice for bug reports.

CNN with Multiple Filters for Bug Reports

Bug reports are written in natural languages and transforming them into word vectors will still contain the semantic information of bug reports. However, this kind of 3-dimensional vectors for each bug report will cause high memory overhead if directly fed into the following cascade forest. In addition, CNN performs excellently in the semantic parsing field of natural language processing because of the convolving filters that can extract local features. Therefore, we adopt the CNN with multiple filters to extract the features of bug reports and reduce the dimensions.

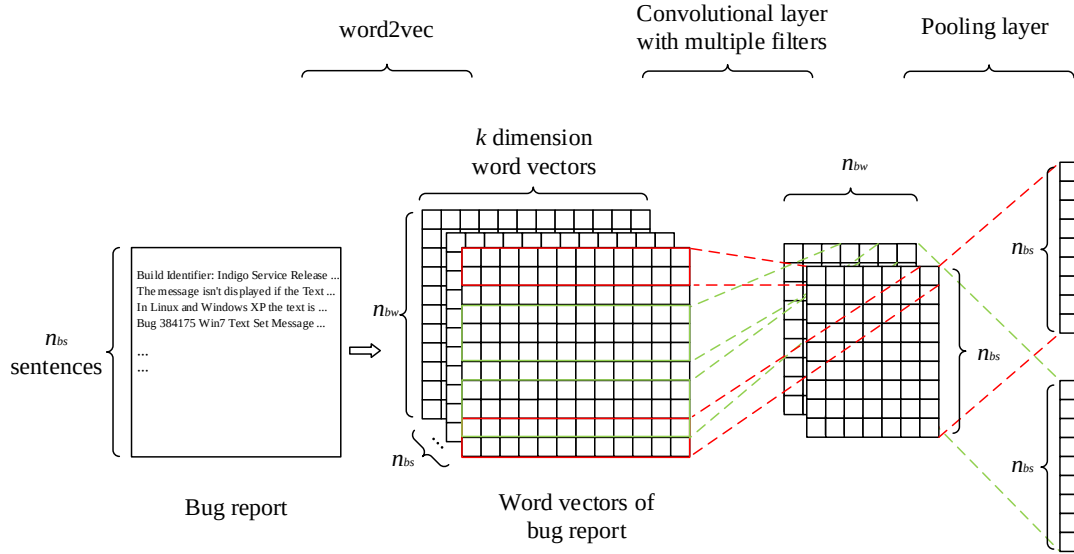


Figure 5.3: The feature extraction of bug reports using CNN.

Figure 5.3 illustrates how to extract features from bug reports using CNN with two kinds of filter sizes, which are the sliding windows with the size of $2 \times k$ and $3 \times k$. The convolutional layer employs a filter $w \in \mathbb{R}^{d \times k \times n_{bs}}$ ($d = 2, 3$ in Figure 5.3) to convolve the vectors of each bug report. The Rectified Linear Units (ReLU) [54]

function is then used to extract more nonlinear features from the vectors. After the convolutional layer, the dimension of each bug report is reduced from $n_{bs} \times n_{bw} \times k$ to $n_{bs} \times n_{bw}$, followed by the pooling layer to further reduce the dimension of the features into $n_{bs} \times 1$. In Figure 5.3, the CNN has two kinds of filter sizes in each of which there is only one filter. To preserve the integrity of the features in bug reports, we implement the CNN with not only more kinds of filter sizes but also more kinds of filters for each filter size. For example, if we adopt two kinds of filter sizes and each filter size contains 100 kinds of filters, then the dimension of the features is $200n_{bs} \times 1$ after the pooling layer. The size and number of filters can be fine-tuned during training.

Ensemble of Random Forests with Multi-grained Scanning for Source Files

Source files are composed of code tokens that are similar to the texts in natural languages, but the programming languages contain more structural information than natural languages. We will adopt the ensemble of random forests involving multiple decision trees to extract the structural features from the source files.

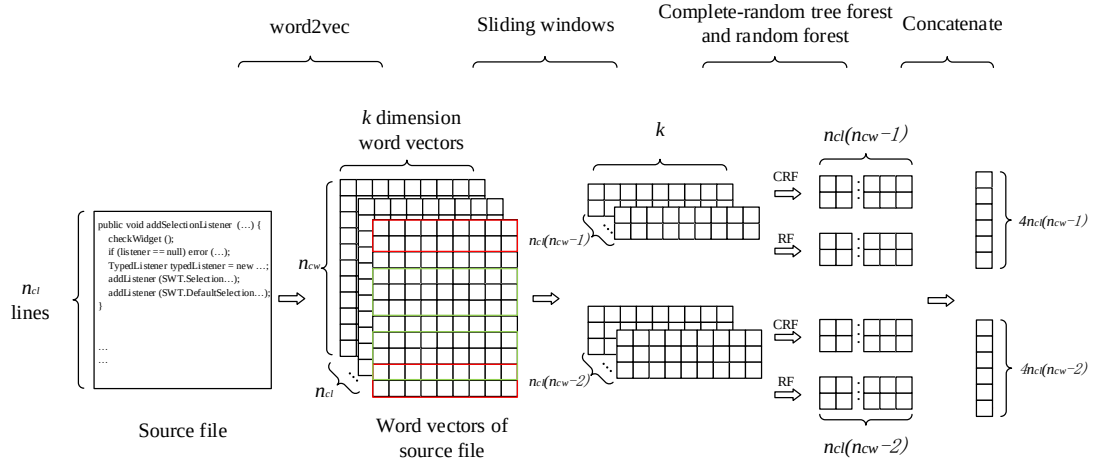


Figure 5.4: The feature extraction of source files using ensemble of random forests.

Figure 5.4 illustrates how to extract features from source files using the en-

semble of random forests with multi-grained scanning. Firstly, different sliding windows with the size of $d \times k$ ($d = 2, 3$ in Figure 5.4) are employed to scan the word vectors of the source files. The new instance vectors with the dimension of $n_{cl}(n_{cw} - d + 1) \times k$ are then generated from the original features (the dimension is $n_{cl} \times n_{cw} \times k$) by the multi-grained scanning process. These new instances obtained from the same sliding window are further fed to train the completely-random tree forest (CRF) [68] and random forest (RF) [11]. Two types of forests are used to encourage diversity that is crucial for ensemble models [146]. For each kind of sliding window, the new feature vectors generated from each forest have the dimension of $2 \times n_{cl}(n_{cw} - d + 1)$, suppose there are 2 classes, and then these feature vectors are concatenated as the transformed features. As shown in Figure 5.4, the final $4n_{cl}(n_{cw} - d + 1) \times 1$ dimensional feature vectors are generated from the original $n_{cl} \times n_{cw} \times k$ dimensional input for each kind of sliding window with the size of $d \times k$.

5.2.1.4 Cascade Forest

The *deep* in deep learning is beneficial from the layer-by-layer processing of the input to extract deep features from them. Similar to this, the cascade structure in cascade forest performs as the layer structure in the deep learning model. In order to extract deeper features from the bug reports and source files, and learn the correlated relationships between them, cascade forest is employed to process their feature vectors obtained respectively from the CNN with multiple filters and the ensemble of random forests with multi-grained scanning. Similar to the method in the literature [147], two completely-random tree forests (CRF) and two random forests (RF) are leveraged to constitute one ensemble in the cascade forest, where there are several such ensembles. This kind of ensemble structure of ensembles is able to extract deeper features from the bug reports and source files.

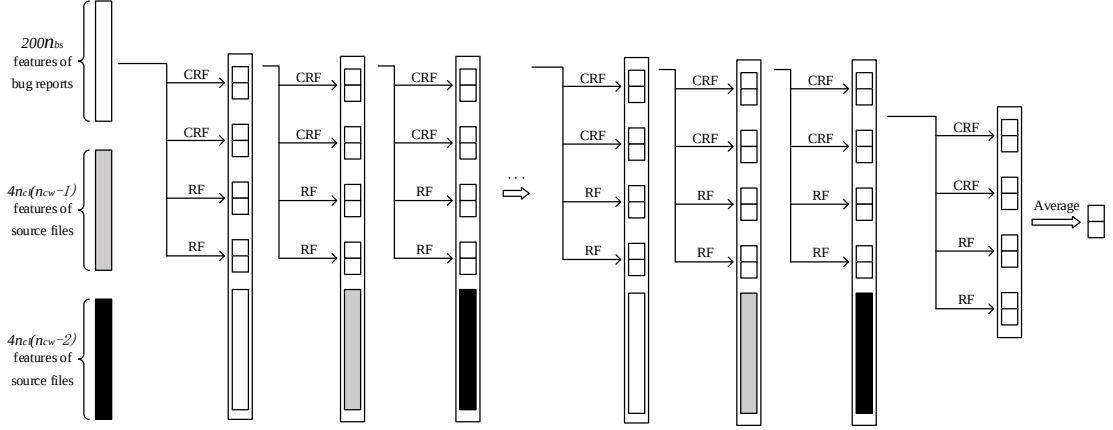


Figure 5.5: Cascade forest to further extract features and learn the correlated relationships between bug reports and source files.

Figure 5.5 depicts how the cascade forest deals with the feature vectors ($200n_{bs} \times 1$) extracted from the bug reports and those ($4n_{cl}(n_{cw} - d + 1) \times 1$) from source files. The $200n_{bs} \times 1$ dimensional feature vectors of bug reports are first fed into the first level of cascade forest with two CRFs and two RFs to generate the first four class vectors. These class vectors are then concatenated with the original $200n_{bs} \times 1$ dimensional feature vectors to be fed into the second level. Likewise, the second class vectors are concatenated with the original $4n_{cl}(n_{cw} - 1) \times 1$ dimensional feature vectors extracted from the source files using the first type of sliding window to be the input of the third level. Similarly, the third class vectors are concatenated with the $4n_{cl}(n_{cw} - 2) \times 1$ dimensional vectors extracted from the second type of sliding window. These cascade procedures will continue until there are no significant improvements during training. This alternate cascade structure is used to learn the correlated relationships between bug reports and source files. The final class vectors are the average values of the last four class vectors.

5.2.2 Experiments

Several experiments are conducted to evaluate the performance of our model CNN_Forest. We will first describe the experimental preparations in Section 5.2.2.1. The experimental results and the corresponding analyses will be presented in Section 5.2.2.2.

5.2.2.1 Experimental Preparations

This section will present the experimental preparations including datasets, experimental settings, evaluation metrics, and competitors.

Datasets

In this section, we use five open-source Java projects (AspectJ², Eclipse UI³, JDT⁴, SWT⁵ and Tomcat⁶) to evaluate the performance of CNN_Forest. The before-fixed version of the source code [133] in each project is obtained to perform the experiments. We make the before-fixed version [133] of the source code in each project publicly available⁷ and provide the tool used to construct the dataset. To make the comparison with existing techniques easier, a splitting strategy similar to that in [134] is adopted. The oldest bug reports of each project are used for tuning the model (validation set) while the older bug reports are for training the model (training set) and the newest bug reports are for testing the model (testing set). Table 5.1 provides more details about the number of bug reports in each set.

Experimental settings

The hyper-parameters of CNN_Forest include the dimension of word vectors (k), the size and the number of filters used in CNN, the size of sliding windows in

²<http://eclipse.org/aspectj/>

³<http://projects.eclipse.org/projects/eclipse.platform.ui>

⁴<http://www.eclipse.org/jdt/>

⁵<http://www.eclipse.org/swt/>

⁶<http://tomcat.apache.org>

⁷<https://github.com/yanxiao6/BugLocalization-dataset>

Table 5.1: Subject projects.

Project	Time Range	# of Bug Reports for Evaluation	# of Bug Reports for Tuning	# of Bug Reports for Training	# of Bug Reports for Testing
AspectJ	03/2002-01/2014	593	100	400	93
Eclipse UI	10/2001-01/2014	3,656	1,500	500	1,656
JDT	10/2001-01/2014	2,632	1,500	500	632
SWT	02/2002-01/2014	2,817	1,500	500	817
Tomcat	07/2002-01/2014	1,056	400	500	156

the ensemble of random forests for scanning, and the number of trees in each forest. According to the work [128], the dimension of word vectors $k = 100$ can achieve comparable performance and consume relatively less time. The filter size ranging from 4 to 6 and the number of filters equal to 100 performs well in bug localization, which implies that the size of sliding windows in CNN can be chosen from $4 \times k$, $5 \times k$, and $6 \times k$. The size of sliding windows in the ensemble of random forests to scan the source files is set to 2 and 3 as suggested in the literature [39] considering the characteristics of the statements in source code. In order to evaluate the generalized effects of the forest ensemble and cascade forest on bug localization, the default number (500) of the trees in each forest as recommended by [147] is used in this section.

Evaluation metrics

The mean average precision (MAP) and mean reciprocal rank (MRR) are used to evaluate the performance of CNN_Forest, which are frequently used in existing studies of bug localization [133, 134, 57]. The higher the values of MRR and MAP, the better the performance of the bug localization technique.

Competitors

In order to exhibit the importance of applying different strategies to extract features from bug reports and source files, we will first compare CNN_Forest with the following two models on the validation set:

- *CNN_CNN* uses the same strategy (CNN) to extract features from bug reports and source files.
- *Forest_Forest* applies the same ensemble of random forests to extract features of bug reports and source files.

All CNNs and random forests use same parameters, as well as the subsequent cascade forest.

CNN_Forest is then compared to the following four state-of-the-art techniques on the testing set:

- *NP-CNN* proposed by Huo et al. [39] applied CNN to learn unified features from natural and programming languages.
- *LR+WE* [134] enhanced their previously-proposed learn-ing-to-rank model (LR) by adding new features obtained by word embedding (WE) techniques.
- *DNNLOC* proposed by Lam et al. [57] combined deep learning techniques with the information retrieval techniques to localize the buggy files for bug reports.
- *BugLocator* [145] measured the textural similarity between the texts in bug reports and source files using the revised Vector Space Model (rVSM).

Since the original implementations of the four models are not released, we tried our best to implement our own versions of these four models. We strictly followed the steps given by [39, 134, 57, 145], but all the detailed implementations might

not be all the same. We used the same dataset to test their performance and got similar results. Thus, we are confident to argue that the comparisons are fair.

5.2.2.2 Results and Analyses

In this section, we report the results of CNN_Forest and competitors on the intrinsic and extrinsic evaluation, and analyze the results.

Intrinsic Evaluation

The intrinsic evaluation is conducted on the validation set to show the improvements achieved by CNN_Forest compared to CNN_CNN and Forest_Forest. The MAP and MRR values of these models are presented in Table 5.2 and the best MAP and MRR values for each project has been highlighted in bold. It can be observed from the results that CNN_Forest is able to achieve the best MAP and MRR except that Forest_Forest obtains 0.005 higher MAP than CNN_Forest on the Project Eclipse UI, which is not a big loss. The highest improvement on MAP achieved by CNN_Forest compared to the best result of the other two models is 0.023, that is about 5% relative improvement.

Thus, we can conclude that it is important to adopt different strategies to extract features from bug reports and source files. The different results also validate that the characteristics of the texts and code tokens in bug reports and source files are different even though they are both written by humans. Bug reports are written in natural languages and CNN performs well in semantic parsing in the field of natural language processing because of the convolving filters. Therefore, CNN_Forest performs better than Forest_Forest. On the other hand, source files are written in programming languages that contain rich and explicit structural information. The source code could be parsed into abstract syntax tree and also reverted. Moreover, tree-structure is good at extracting the structural information from source code [65, 81, 83, 121]. CNN_Forest thus has better performance than

Table 5.2: Performance comparison for intrinsic evaluation.

Project	Model	MAP	MRR
AspectJ	CNN_Forest	0.458	0.563
	CNN_CNN	0.449	0.560
	Forest_Forest	0.430	0.549
Eclipse UI	CNN_Forest	0.460	0.590
	CNN_CNN	0.441	0.561
	Forest_Forest	0.465	0.588
JDT	CNN_Forest	0.448	0.517
	CNN_CNN	0.448	0.502
	Forest_Forest	0.435	0.492
SWT	CNN_Forest	0.462	0.530
	CNN_CNN	0.415	0.507
	Forest_Forest	0.439	0.512
Tomcat	CNN_Forest	0.627	0.681
	CNN_CNN	0.623	0.669
	Forest_Forest	0.604	0.647

CNN_CNN.

Extrinsic Evaluation

The CNN_Forest is then compared to other four current state-of-the-art techniques (NP-CNN [39], LR+WE [134], DNNLOC [57], and BugLocator [145]) in the field of bug localization on the testing set. Their performances on five open-source projects are shown in Table 5.3. The best MAP and MRR results are also highlighted in bold. As we can observe from the table, CNN_Forest always performs better than the four competitors except for the MRR in Project JDT, but the loss is negligible. The improvements on MAP of CNN_Forest compared to the best results of the four competitors range from 0.003 to 0.034. Moreover, the improvements are more obvious when evaluating on the small dataset, i.e. Project AspectJ and Tomcat.

NP-CNN applied CNN to extract features from both bug reports and source

Table 5.3: Performance comparison with the state-of-the-art techniques.

Project	Model	MAP	MRR
AspectJ	CNN_Forest	0.436	0.519
	NP-CNN	0.402	0.487
	LR+WE	0.302	0.454
	DNNLOC	0.323	0.475
	BugLocator	0.278	0.364
Eclipse UI	CNN_Forest	0.432	0.534
	NP-CNN	0.429	0.529
	LR+WE	0.398	0.461
	DNNLOC	0.413	0.514
	BugLocator	0.332	0.383
JDT	CNN_Forest	0.423	0.514
	NP-CNN	0.405	0.463
	LR+WE	0.417	0.516
	DNNLOC	0.342	0.452
	BugLocator	0.290	0.367
SWT	CNN_Forest	0.394	0.482
	NP-CNN	0.371	0.466
	LR+WE	0.381	0.446
	DNNLOC	0.369	0.445
	BugLocator	0.269	0.312
Tomcat	CNN_Forest	0.550	0.614
	NP-CNN	0.529	0.585
	LR+WE	0.503	0.556
	DNNLOC	0.523	0.604
	BugLocator	0.425	0.481

files. The performance is limited because the characteristics of the texts in bug reports and the code tokens in source files are different just as we observed in the intrinsic evaluation. The structural information of the source files extracted by NP-CNN is subject to each code line, however, the programming syntactic information inside the program blocks is not extracted. Our proposed model CNN_Forest

considers both the structural information and programming syntactic information benefiting from the ensemble of random forests with multi-grained scanning. Moreover, the texts and code tokens are processed by word embedding techniques before fed into the feature extraction part. The semantic information is therefore preserved, which is lost in NP-CNN without word embedding techniques. That's why CNN_Forest has better performance than NP-CNN. On the other hand, compared to CNN_Forest and NP-CNN, LR+WE and DNNLOC tried to add the semantic information into their models but did not take the program structure related information into consideration. Thus, their performances are worse than CNN_Forest. BugLocator was based on textural similarity and did not involve the semantic and structural features of bug reports and source files. Therefore, BugLocator's performance is much worse than the other techniques.

5.2.3 Discussion

5.2.3.1 Why does CNN_Forest Work?

CNN_Forest tries to extract both semantic information and programming structural information from source files, which is different from the ones in bug reports written in natural languages. This kind of information is seldom considered by existing techniques.

The word embedding technique is first applied to convert the texts in bug reports and source files into word vectors. Similar words are close to each other in the vector space, which preserves the semantic information. Different strategies are then used to extract the features from bug reports and source files. Bug reports are written in natural languages where CNN has performed excellently because of convolving filters. On the other hand, source files are composed of code tokens that involve more stringent structural information. They can be parsed into

abstract syntax tree and also be reverted from the abstract syntax tree. Thus, we leverage the ensemble of random forests involving many trees with multiple sliding windows to extract the programming structural information. Therefore, CNN_Forest not only preserves the semantics but also leverages the structural information of lexical terms in bug reports and the programming code tokens in source files. Finally, unlike the linear combination of the extracted features used in the literature [133, 134, 39], the cascade forest is used to further extract features from both bug reports and source files, and the correlated relationships between them are learned by the alternate cascade structure that is similar to the layer-structure in deep learning.

5.2.3.2 Threats to Validity

The experimental results demonstrate the feasibility and effectiveness of CNN_Forest, however, we do acknowledge that there are still some potential threats to validity of our study. Following the recommendations of Wohlin et al. [123], we discuss the threats to internal validity, construct validity and external validity of this study.

Internal Validity

The performance of our proposed model may be somewhat dependent on the performance of the word embedding techniques. We have checked the property of the adopted word embedding techniques before adopting them in our model. It is important to do so before implementing our model. Secondly, the hyper-parameters configuration set could introduce some bias in the experimental results. However, we set the parameters according to the suggestions of the existing studies, which enabled our choices to be reasonable. For instance, the size and number of the filters used in CNN and the size of the sliding windows used in the ensemble of random forests were set as the suggestions given in the literature [128, 39]. And

in order to examine the intrinsic property of the forest structures, we used the default number of trees in each forest as recommended by Zhou et al. [147]. More fine-tunes might be needed for our model. We leave this for future studies.

Construct Validity

When conducting experiments, we split the dataset in each project into validation set (oldest bug reports), training set (older bug reports) and testing set (newest bug reports) similar to the strategy used in the literature [134]. However, the splitting percentage is different from that used in most machine learning techniques [18, 81, 89]. It would be interesting to examine the effects of different splitting percentages on our proposed model, which is worth to be explored in further studies. In this section, we used two measures (MAP and MRR) that are widely used in most bug localization studies [56, 134, 145]. In future work, we will consider other evaluation metrics, such as Accuracy@k [133] and Area Under the receiver operator characteristic Curve (AUC) [62]. Besides, other validations (e.g., statistical significance and effect size of the results) will be considered in future when there are more project dataset used to evaluate the proposed model.

External Validity

We evaluated our model on five dataset from Java projects as many bug localization studies [56, 133, 134] did and tried to report the general results. However, the selection of only five projects may have potentially limited representativeness. On the other hand, the performance of our model on other projects written in other programming languages is still unknown. We intend to examine our model on more projects, especially the ones written in, for example, C++, in a future study.

5.3 Chapter Summary

This chapter proposes CNN_Forest, a new CNN and random forest-based model for bug localization. After pre-processing the texts in bug reports and source files, CNN_Forest first employs the word embedding technique to convert the words into vectors with the preservation of the semantic information in them. The CNN with multiple filters and the ensemble of random forests with multi-grained scanning are then used to extract both semantic and structural features from the word vectors of bug reports and source files. Finally, the cascade forest is used to further conclude features from the feature vectors extracted by CNN and the ensemble of random forests, and learn the correlated relationships between bug reports and source files benefiting from the alternate cascade structure. Our evaluation on five open-source projects showed that the separately learned features do improve the performance of the model. Moreover, CNN_Forest achieved higher MAP (at most 0.034) and MRR (at most 0.032) than the best results of the four current state-of-the-art techniques (NP-CNN, LR+WE, DNNLOC, and BugLocator).

Chapter 6

Conclusions and Future Work

This thesis has been devoted to the study of bug localization on the lexical mismatch analysis together with extracting both semantic and structural features from bug reports and source code files. This chapter summarizes this thesis in concluding the study on the three problems, and discusses some promising future directions including the way of enhancing the studied techniques in this thesis as well as future researches on bug localization.

6.1 Conclusions

With the increase of the number of software systems, especially those of the open source software, automatically localizing bugs given bug reports submitted by developers, testers or users becomes much more important in recent years. According to the analysis in this thesis, there are three main problems in the existing bug localization studies, i.e., lexical mismatch problem, semantic information omission and structural information omission. Meanwhile, we propose specific techniques to solve these problems.

The detailed summaries are as follows. Firstly, BugTranslator and the en-

hanced version distinguishes bug reports and source code into different symbolic classes and then extract deep semantic similarity and relevance between bug reports and the corresponding buggy files to bridge the lexical gap at its source, thereby further improving the performance of bug localization. Even though it hasn't beat all the existing techniques especially some combined models, there is a lot of room to improve this model since it is a new research direction for bug localization. Secondly, DeepLocator and DeepLoc, deep learning-based models that consist of an enhanced CNN, together with word-embedding techniques and CNNs for detecting features from bug reports and source files, are proposed to include the semantic information hidden in them and improve the performance of bug localization. Thirdly, CNN_Forest, a new CNN and random forest-based model, is proposed for bug localization to extract both the semantic and structural information in bug reports and source files.

Table 6.1 shows the performance comparison between proposed models and the state-of-the-art techniques. Compared with HyLoc that does not include semantic and structural information, all other models that consider either of them achieve improvements. DeepLocator and NP-CNN consider semantic and structural information respectively. Either of them cannot always beat the other one, but both of them have lower accuracy than CNN_Forest that includes both the semantic and structural information. Thus the word embedding techniques that retain semantics and tree-based architectures that extract structural information are important for bug localization. Different from DeepLocator, DeepLoc uses two different word embedding techniques and CNNs to process bug reports and source files. Even though DeepLoc does not use tree-based architecture for source files, it can also achieve comparable accuracies with CNN_Forest that uses a same word technique for bug reports and source files. Thus the different word embedding techniques are also important.

6.2. FUTURE WORK

Table 6.1: Performance Comparison between proposed models and the State-of-the-art Techniques.

Project	Model	MAP	MRR
AspectJ	DeepLoc	0.42	0.51
	CNN_Forest	0.436	0.519
	DeepLocator	0.34	0.49
	NP-CNN	0.402	0.487
	HyLoc	0.32	0.48
Eclipse UI	DeepLoc	0.43	0.53
	CNN_Forest	0.432	0.534
	DeepLocator	0.42	0.51
	NP-CNN	0.429	0.529
	HyLoc	0.41	0.51
JDT	DeepLoc	0.44	0.53
	CNN_Forest	0.423	0.514
	DeepLocator	0.39	0.47
	NP-CNN	0.405	0.463
	HyLoc	0.34	0.45
SWT	DeepLoc	0.40	0.49
	CNN_Forest	0.394	0.482
	DeepLocator	0.39	0.48
	NP-CNN	0.371	0.466
	HyLoc	0.37	0.45
Tomcat	DeepLoc	0.56	0.62
	CNN_Forest	0.550	0.614
	DeepLocator	0.54	0.60
	NP-CNN	0.529	0.585
	HyLoc	0.52	0.60

6.2 Future Work

There are several directions for future research on improving the accuracy of bug localization. We first present a summary of the likely future research areas regarding our approaches. Furthermore, we discuss other research directions towards

bug localization.

6.2.1 Future Work towards Proposed Techniques

To further bridge the lexical gap and improve the accuracy of bug localization, we will enhance and automate BugTranslator by adding the missing factors, including bug-fixing recency, frequency, and the observation that a previously fixed file may be responsible for similar bugs. Furthermore, RNN with LSTM cells has been noted to be sensitive to word order. Localizing buggy classes or methods instead of buggy files may produce more benefit to our model. We will also investigate this in a future study.

The existing learning-based models, including our models, seldom consider the orders of buggy files during training, which may limit their performance when using metrics such as MAP and MRR to test their ability. In the future, we intend to improve DeepLoc’s sensitivity to buggy file orders. We will also investigate DeepLoc’s performance using other TF-IDF weight schemes [133, 72]. Few papers have studied the effect of imbalanced data on bug localization and relevant techniques, which we leave for future work.

For the proposed CNN_Forest, we would like to fine-tune it to further improve its performance, such as more different kinds of random forests (e.g., extra-trees) to enhance diversity. Moreover, few existing studies focused on the projects written in other programming languages, e.g., C/C++. We intend to extend our model to these projects for bug localization and use more measures (Accuracy@k, Area Under the receiver operator characteristic Curve, and statistical significance) to evaluate their performances. In addition, it would be interesting to examine the performance of our model in other applications of software engineering, such as defect prediction.

6.2.2 Future Research towards Bug Localization

As discussed in Chapter 2 of literature review, several kinds of related information (code structure, part-of-speech tags, similar bug reports, code version history, stack traces) besides bug reports have been used to improve the accuracy of bug localization. However, there are such many choices that might confuse researchers. Whether all of them are necessary and should be included when localizing buggy files, or the subset of them are enough, or even considering too much information might confuse the proposed models? A comprehensive empirical study regarding the additional information should be put on the agenda.

According to our preliminary analysis on the results, we find that for our model, it is better to provide clear descriptions instead of only submitting summaries and it is important to ensure the described observed behavior is correct. But the comprehensive guidelines of writing good bug reports still need more empirical experiments and analyses. We will spend more time in future.

Currently, the most influential factor that hinders the improvements of bug localization techniques is the quality of bug report. There are no strict writing rules for bug reports. The summary and description in bug reports are usually used in bug localization techniques. However, many bug reports are lack of description. The summary with a few words challenges the automated bug localization techniques and even developers. The software requirements document that is professional and rigorous written in natural languages can be taken into consideration as a subsidiary [5].

There might be two sources of further improvements for bug localization. Firstly, some bug reports include stack traces in the description. But not all the files or methods in the traces contain bugs. Or even there are no buggy files in it. Few approaches try to distinguish them. Secondly, we find that there exist two or more buggy files that always appear simultaneously. Once one of the buggy

files is located, the probability of other corresponding files can also be increased, which will help improve the accuracy of bug localization.

In addition, there are several studies to localize bugs in method-level [136, 140]. However, their accuracies are too low to broadcast to industry. There are still many challenges remaining to be solved in the method-level bug localization. Firstly, most of the existing dataset is collected for the file-level bug localization. Thus, the dataset with low quality and noise for method-level bug localization challenges its development. Secondly, there is still room for improvements of the bug localization accuracy in file-level, which also hinder the improvement of method-level bug localization. Thirdly, other important information besides call dependency of methods (cyclomatic complexity [118], lines of code, version history of method) needs more effort to explore for improving the accuracy of the method-level bug localization.

Bibliography

- [1] Rui Abreu, Peter Zoetewij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.
- [2] Rui Abreu, Peter Zoetewij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.
- [3] Nasir Ali, Aminata Sabane, Yann-Gael Gueheneuc, and Giuliano Antoniol. Improving bug location using binary class relationships. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 174–183. IEEE, 2012.
- [4] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100, 2016.
- [5] Amira A Alshazly, Ahmed M Elfatratry, and Mohamed S Abougabal. Detecting defects in software requirements specification. *Alexandria Engineering Journal*, 53(3):513–527, 2014.
- [6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural ma-

- chine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.
- [7] Joeran Beel, Stefan Langer, and Bela Gipp. TF-IDuF: a novel term-weighting scheme for user modeling based on users’ personal document collections. In *Proceedings of the iConference 2017*, Wuhan, China, March 22-25 2017.
- [8] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.
- [9] Dave Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. To camelcase or under_score. In *Program Comprehension, 2009. ICPC’09. IEEE 17th International Conference on*, pages 158–167. IEEE, 2009.
- [10] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3(Jan):993–1022, 2003.
- [11] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.
- [12] Peter F Brown, Vincent J Della Pietra, Stephen A Della Pietra, and Robert L Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics*, 19(2):263–311, 1993.
- [13] Bernd Bruegge and Allen H Dutoit. *Object-oriented software engineering using UML, patterns and Java-(Required)*. Prentice Hall, 2004.
- [14] Li Cai, Guangyou Zhou, Kang Liu, and Jun Zhao. Learning the latent topics for question retrieval in community qa. In *Proceedings of 5th international joint conference on Natural Language Processing*, pages 273–281, 2011.

BIBLIOGRAPHY

- [15] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. Using bug descriptions to reformulate queries during text-retrieval-based bug localization. *Empirical Software Engineering*, pages 1–61, 2019.
- [16] Jianpeng Cheng, Zhongyuan Wang, Ji-Rong Wen, Jun Yan, and Zheng Chen. Contextual text understanding in distributional semantic space. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 133–142. ACM, 2015.
- [17] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, 2014.
- [18] Morakot Choetkiertikul, Hoa Khanh Dam, Truyen Tran, Trang Thi Minh Pham, Aditya Ghose, and Tim Menzies. A deep learning model for estimating story points. *IEEE Transactions on Software Engineering*, 2018.
- [19] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.
- [20] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.
- [21] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

- [22] W Bruce Croft, Donald Metzler, and Trevor Strohman. *Search engines: Information retrieval in practice*, volume 283. Addison-Wesley Reading, 2010.
- [23] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight bug localization with ample. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 99–104. ACM, 2005.
- [24] Tung Dao, Lingming Zhang, and Na Meng. How does execution information help with information-retrieval based bug localization? In *Proceedings of the 25th International Conference on Program Comprehension*, pages 241–250. IEEE Press, 2017.
- [25] Steven Davies, Marc Roper, and Murray Wood. Using bug report similarity to enhance bug localisation. In *2012 19th Working Conference on Reverse Engineering*, pages 125–134. IEEE, 2012.
- [26] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*, pages 23–37. Springer, 1995.
- [27] Kunihiro Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and Cooperation in Neural Nets*, pages 267–285. Springer, 1982.
- [28] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. On the use of relevance feedback in ir-based concept location. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 351–360. IEEE, 2009.
- [29] Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber. Learning precise timing with lstm recurrent networks. *Journal of machine learning research*, 3(Aug):115–143, 2002.

- [30] Edward Grefenstette, Phil Blunsom, Nando de Freitas, and Karl Moritz Hermann. A deep architecture for semantic parsing. In *Proceedings of the ACL 2014 Workshop on Semantic Parsing*, pages 22–27, 2014.
- [31] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018.
- [32] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM, 2016.
- [33] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jianguang Sun. Dlfuzz: differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 739–743. ACM, 2018.
- [34] Zellig S Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.
- [35] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012.
- [36] Thong Van-Duc Hoang, Richard J Oentaryo, Tien-Duy Bui Le, and David Lo. Network-clustered multi-modal bug localization. *IEEE Transactions on Software Engineering*, 2018.
- [37] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning deep structured semantic models for web search using click-

- through data. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 2333–2338. ACM, 2013.
- [38] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex. *The Journal of Physiology*, 160(1):106–154, 1962.
- [39] Xuan Huo, Ming Li, and Zhi-Hua Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1606–1612, 2016.
- [40] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.
- [41] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 2073–2083, 2016.
- [42] Mohit Iyyer, Peter Enns, Jordan Boyd-Graber, and Philip Resnik. Political ideology detection using recursive neural networks. In *Proceedings of the Association for Computational Linguistics*, pages 1113–1122, 2014.
- [43] Jiwoon Jeon, W Bruce Croft, and Joon Ho Lee. Finding similar questions in large question and answer archives. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 84–90. ACM, 2005.
- [44] Zongcheng Ji, Fei Xu, Bin Wang, and Ben He. Question-answer topic model for question retrieval in community question answering. In *Proceedings of*

BIBLIOGRAPHY

- the 21st ACM international conference on Information and knowledge management*, pages 2471–2474. ACM, 2012.
- [45] Rie Johnson and Tong Zhang. Effective use of word order for text categorization with convolutional neural networks. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 103–112, 2015.
- [46] James A Jones and Mary Jean 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.
- [47] Tom Kenter, Alexey Borisov, and Maarten de Rijke. Siamese cbow: Optimizing word embeddings for sentence representations. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 941–951, 2016.
- [48] Vlado Kešelj, Fuchun Peng, Nick Cercone, and Calvin Thomas. N-gram-based author profiles for authorship attribution. In *Proceedings of the Conference Pacific Association for Computational Linguistics, PACLING*, volume 3, pages 255–264, 2003.
- [49] Saket Khatiwada, Miroslav Tushev, and Anas Mahmoud. Just enough semantics: An information theoretic approach for ir-based software bug localization. *Information and Software Technology*, 93:45–57, 2018.
- [50] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE Transactions on Software Engineering*, 39(11):1597–1610, 2013.

- [51] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, 2014.
- [52] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. Character-aware neural language models. In *AAAI*, pages 2741–2749, 2016.
- [53] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [54] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.
- [55] Bogdan Kwolek. Face detection using convolutional neural networks and gabor filters. *Artificial Neural Networks: Biological Inspirations–ICANN 2005*, pages 551–556, 2005.
- [56] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 476–481. IEEE, 2015.
- [57] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Bug localization with combination of deep learning and information retrieval. In *Proceedings of the 25th International Conference on Program Comprehension*, pages 218–229. IEEE Press, 2017.
- [58] Thomas D LaToza and Brad A Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, page 8. ACM, 2010.

- [59] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.
- [60] Tien-Duy B Le, Richard J Oentaryo, and David 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.
- [61] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.
- [62] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.
- [63] Chris Lewis and Rong Ou. Bug prediction at google. *URL: <http://google-engtools.blogspot.in/2011/12/bug-prediction-at-google.html>*, 2011.
- [64] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670. ACM, 2014.
- [65] Ziyi Lin, Hao Zhong, Yuting Chen, and Jianjun Zhao. Lockpeeker: detecting latent locks in java apis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 368–378. ACM, 2016.
- [66] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of*

- the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 599–609, 2016.
- [67] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32(10):831–848, 2006.
- [68] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 413–422. IEEE, 2008.
- [69] Stacy K Lukins, Nicholas A Kraft, and Letha H Etzkorn. Bug localization using latent dirichlet allocation. *Information and Software Technology*, 52(9):972–990, 2010.
- [70] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 120–131. ACM, 2018.
- [71] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.
- [72] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. Scoring, term weighting and the vector space model. *Introduction to Information Retrieval*, 100:2–4, 2008.
- [73] Michael Mealling and Ray Denenberg. Report from the joint w3c/ietf uri planning interest group: Uniform resource identifiers (uris), urls, and uni-

BIBLIOGRAPHY

- form resource names (urns): Clarifications and recommendations. Technical report, 2002.
- [74] Larry Medsker and Lakhmi C Jain. *Recurrent neural networks: design and applications*. CRC press, 1999.
- [75] Yajie Miao and Florian Metze. On speaker adaptation of long short-term memory recurrent neural networks. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.
- [76] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
- [77] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3, 2010.
- [78] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, pages 3111–3119, 2013.
- [79] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. On the use of stack traces to improve text retrieval-based bug localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 151–160. IEEE, 2014.
- [80] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, pages 181–190. ACM, 2008.

- [81] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, pages 1287–1293, 2016.
- [82] Arvind Neelakantan, Jeevan Shankar, Alexandre Passos, and Andrew McCallum. Efficient non-parametric estimation of multiple embeddings per word in vector space. *arXiv preprint arXiv:1504.06654*, 2015.
- [83] Anh Tuan Nguyen and Tien N Nguyen. Graph-based statistical language model for code. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 858–868. IEEE, 2015.
- [84] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 263–272. IEEE, 2011.
- [85] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, David Lo, and Chengnian Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 70–79. ACM, 2012.
- [86] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar M Al-Kofahi, and Tien N Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 383–392. ACM, 2009.

- [87] Matteo Pagliardini, Prakhar Gupta, and Martin Jaggi. Unsupervised learning of sentence embeddings using compositional n-gram features. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 528–540, 2018.
- [88] John F Pane, Brad A Myers, et al. Studying the language and structure in non-programmers’ solutions to programming problems. *International Journal of Human-Computer Studies*, 54(2):237–264, 2001.
- [89] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*, pages 547–553. Springer, 2015.
- [90] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6), 2007.
- [91] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 357–367. IEEE, 2016.
- [92] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.
- [93] Shanto Rahman and Kazi Sakib. An appropriate method ranking approach

- for localizing bugs using minimized search space. In *ENASE*, pages 303–309, 2016.
- [94] Michael Rath, David Lo, and Patrick Mäder. Analyzing requirements and traceability information to improve bug localization. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 442–453. IEEE, 2018.
- [95] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 345–355. IEEE, 2013.
- [96] Giuseppe Scanniello and Andrian Marcus. Clustering support for static concept location in source code. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 1–10. Ieee, 2011.
- [97] Giuseppe Scanniello, Andrian Marcus, and Daniele Pascale. Link analysis algorithms for static concept location: an empirical assessment. *Empirical Software Engineering*, 20(6):1666–1720, 2015.
- [98] Yelong Shen, Xiaodong He, Jianfeng Gao, Li Deng, and Grégoire 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.
- [99] Zhendong Shi, Jacky Keung, Kwabena Ebo Bennin, and Xingjun Zhang. Comparing learning to rank techniques in hybrid bug localization. *Applied Soft Computing*, 62:636–648, 2018.

BIBLIOGRAPHY

- [100] Shivkumar Shivaji, Jr E James Whitehead, Ram Akella, and Sunghun Kim. Reducing features to improve bug prediction. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 600–604. IEEE Computer Society, 2009.
- [101] Bunyamin Sisman, Shayan A Akbar, and Avinash C Kak. Exploiting spatial code proximity and order for improved source code retrieval for bug localization. *Journal of Software: Evolution and Process*, 29(1):e1805, 2017.
- [102] Bunyamin Sisman and Avinash C Kak. Incorporating version histories in information retrieval based bug localization. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 50–59. IEEE, 2012.
- [103] Bird Steven, Ewan Klein, and Edward Loper. Natural language processing with python. *OReilly Media Inc*, 2009.
- [104] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.
- [105] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.
- [106] Aoi Takahashi, Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. A preliminary study on using code smells to improve bug localization. In *Proceedings of the 26th Conference on Program Comprehension*, pages 324–327. ACM, 2018.

- [107] Stephen W Thomas, Meiyappan Nagappan, Dorothea Blostein, and Ahmed E Hassan. The impact of classifier configuration and classifier combination on bug localization. *IEEE Transactions on Software Engineering*, 39(10):1427–1443, 2013.
- [108] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314. ACM, 2018.
- [109] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. Emf-incquery: An integrated development environment for live model queries. *Science of Computer Programming*, 98:80–99, 2015.
- [110] Laurens Van Der Maaten. Accelerating t-sne using tree-based algorithms. *Journal of machine learning research*, 15(1):3221–3245, 2014.
- [111] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*, pages 1096–1103. ACM, 2008.
- [112] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–I. IEEE, 2001.
- [113] Qianqian Wang, Chris Parnin, and Alessandro Orso. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 In-*

BIBLIOGRAPHY

- ternational Symposium on Software Testing and Analysis*, pages 1–11. ACM, 2015.
- [114] Shaowei Wang and David Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 53–63. ACM, 2014.
- [115] Shaowei Wang and David Lo. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process*, 28(10):921–942, 2016.
- [116] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, pages 297–308. ACM, 2016.
- [117] Yaojing Wang, Yuan Yao, Hanghang Tong, Xuan Huo, Min Li, Feng Xu, and Jian Lu. Bug localization via supervised topic modeling. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 607–616. IEEE, 2018.
- [118] Arthur Henry Watson, Dolores R Wallace, and Thomas J McCabe. *Structured testing: A testing methodology using the cyclomatic complexity metric*, volume 500. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1996.
- [119] Geoffrey I Webb. Multiboosting: A technique for combining boosting and wagging. *Machine learning*, 40(2):159–196, 2000.
- [120] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. Locus: Locating bugs from software changes. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 262–273. IEEE, 2016.

- [121] Michael W Whalen, Suzette Person, Neha Rungta, Matt Staats, and Daniela Grijincu. A flexible and non-intrusive approach for computing complex structural coverage metrics. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 506–516. IEEE Press, 2015.
- [122] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 334–345. IEEE, 2015.
- [123] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.
- [124] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 181–190. IEEE, 2014.
- [125] Yan Xiao and Jacky Keung. Improving bug localization with character-level convolutional neural network and recurrent neural network. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 703–704. IEEE, 2018.
- [126] Yan Xiao, Jacky Keung, Kwabena E Bennin, and Qing Mi. Machine translation-based bug localization technique for bridging lexical gap. *Information and Software Technology*, 99:58–61, 2018.
- [127] Yan Xiao, Jacky Keung, Kwabena E Bennin, and Qing Mi. Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology*, 105:17–29, 2019.

BIBLIOGRAPHY

- [128] Yan Xiao, Jacky Keung, Qing Mi, and Kwabena E Bennin. Improving bug localization with an enhanced convolutional neural network. In *Asia-Pacific Software Engineering Conference (APSEC), 2017 24th*, pages 338–347. IEEE, 2017.
- [129] Yan Xiao, Jacky Keung, Qing Mi, and Kwabena E Bennin. Bug localization with semantic and structural features using convolutional neural network and cascade forest. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, pages 101–111. ACM, 2018.
- [130] Xihao Xie, Wen Zhang, Ye Yang, and Qing Wang. Dretom: Developer recommendation based on topic models for bug resolution. In *Proceedings of the 8th international conference on predictive models in software engineering*, pages 19–28. ACM, 2012.
- [131] Jifeng Xuan and Martin Monperrus. Learning to combine multiple ranking metrics for fault localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 191–200. IEEE, 2014.
- [132] Xiaobing Xue, Jiwoon Jeon, and W Bruce Croft. Retrieval models for question and answer archives. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 475–482. ACM, 2008.
- [133] Xin Ye, Razvan Bunescu, and Chang 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*, pages 689–699. ACM, 2014.

- [134] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, pages 404–415. ACM, 2016.
- [135] Klaus Changsun Youm, June Ahn, Jeongho Kim, and Eunseok Lee. Bug localization based on code change histories and bug reports. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*, pages 190–197. IEEE, 2015.
- [136] Klaus Changsun Youm, June Ahn, and Eunseok Lee. Improved bug localization based on code change histories and bug reports. *Information and Software Technology*, 82:177–192, 2017.
- [137] Matthew D Zeiler. Adadelata: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.
- [138] Kai Zhang, Wei Wu, Haocheng Wu, Zhoujun Li, and Ming Zhou. Question retrieval with high quality answers in community question answering. In *Proceedings of the 23rd ACM international conference on conference on information and knowledge management*, pages 371–380. ACM, 2014.
- [139] Sai Zhang and Congle Zhang. Software bug localization with markov logic. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 424–427. ACM, 2014.
- [140] Wen Zhang, Ziqiang Li, Qing Wang, and Juan Li. Finelocator: a novel approach to method-level fine-grained bug localization by query expansion. *Information and Software Technology*, 2019.
- [141] Ye Zhang and Byron Wallace. A sensitivity analysis of (and practitioners’

BIBLIOGRAPHY

- guide to) convolutional neural networks for sentence classification. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 253–263, 2017.
- [142] Yun Zhang, David Lo, Xin Xia, Tien-Duy B Le, Giuseppe Scanniello, and Jianling Sun. Inferring links between concerns and methods with multi-abstraction vector space model. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 110–121. IEEE, 2016.
- [143] Yun Zhang, David Lo, Xin Xia, Giuseppe Scanniello, Tien-Duy B Le, and Jianling Sun. Fusing multi-abstraction vector space models for concern localization. *Empirical Software Engineering*, 23(4):2279–2322, 2018.
- [144] Guangyou Zhou, Li Cai, Jun Zhao, and Kang Liu. Phrase-based translation model for question retrieval in community question answer archives. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 653–662. Association for Computational Linguistics, 2011.
- [145] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, pages 14–24. IEEE Press, 2012.
- [146] Zhi-Hua Zhou. *Ensemble methods: foundations and algorithms*. CRC press, 2012.
- [147] Zhi-Hua Zhou and Ji Feng. Deep forest: towards an alternative to deep neural networks. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 3553–3559. AAAI Press, 2017.