Adaptive Cross-Project Bug Localization with Graph Learning

by

Venkatraman Arumugam

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

Author's Declaration

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

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

Abstract

Bug localization is the process of identifying the source code files associated with a bug report. This is important because it allows developers to focus their efforts on fixing the bugs than finding the root cause of bugs in the first place. A number of different techniques have been developed for bug localization, but recent research has shown that supervised approaches using historical data are more effective than other methods. In reality, for the supervised approaches to work, these approaches need high quality and quantity of label-rich datasets. However, preparing training data for new projects and retraining the bug localization models can be highly expensive. Additionally, most of the projects do not have rich historic bug data, as pointed out by Zimmermann et al. This necessitates cross-project bug localization, which involves using data from one project to extract the transferable features to localize bugs in a new project. In this thesis, we aim to provide a bug localization model to locate buggy source code files in a new project without retraining by leveraging the transfer learning capability of deep learning models.

Deep learning models can be trained once in a label-rich dataset and transferred to a new dataset. By leveraging deep learning, we propose AdaBL and AdaBL+GL, which can be trained once and transferred to a new project. The main idea behind AdaBL is to learn the syntactic and semantic relationship between bug reports and source code separately. The syntactic patterns are transferable features that exist between cross-projects. We pair AdaBL with a graph neural network to represent the source code as a graph to improve the semantic learning capability. We also performed a detailed survey to compile the bug localization research published since 2016 to examine the experimental settings practiced and the availability of the replication package of deep learning-based bug localization research.

Acknowledgements

I would like to thank Meiyappan Nagappan for being my advisor and mentor during my Master's. I must also acknowledge other researchers from various publications, this work would not have been possible without their contributions.

Table of Contents

List of Figures viii

List of Tables ix

1 Introduction 1

1.1 Bug Localization . . . 1
1.2 Cross-Project Bug Localization . . . 1
1.3 Source Code is Multi-Faceted . . . 2
1.4 DL-based Bug Localization Literature Survey . . . 3
1.5 Contributions . . . 3
1.6 Research Questions . . . 4

2 Related Works 5

3 Dataset 12

3.1 Motivation . . . 12
3.2 Dataset . . . 12
3.2.1 Dataset settings . . . 13
3.2.2 Data collection . . . 14
3.3 Experiments Setup . . . 14
3.3.1 Evaluation and Metrics . . . 15

4 Replication of Benchmark Baselines **17**
4.1 rVSM . . . 18
4.2 DNNLOC . . . 19
4.3 Globug . . . 20

5 Methodology **22**
5.1 Motivation . . . 23
5.2 Adaptive Cross-project Bug Localization . . . 23
5.2.1 Model Architecture and implementation . . . 24
5.2.2 Hyperparameter Specifications . . . 26
5.2.3 Model Training . . . 26
5.3 Adaptive Cross-project Bug Localization with Graph Learning . . . 27
5.3.1 Model Architecture and implementation . . . 27

6 Results **33**
6.1 R1. How do different dataset settings impact the performance of the bug localization model? . . . 33
6.1.1 Top K baselines . . . 34
6.1.2 MRR & MAP Baselines . . . 34
6.2 R2. Does the performance of ML/DL-based localization for a project vary depending on the dataset setting? . . . 37

7 Result Analysis **38**
7.1 Motivation . . . 38

8 Threats to Validity **42**
8.1 Internal Validity . . . 42
8.2 Construct Validity . . . 43
8.3 External Validity . . . 43

9 Conclusion **44**

9.1 Further Work 45

References **46**

APPENDIX **54**

Our Tools, Artifacts, Results 54

List of Figures

2.1 ML/DL Bug Localization Model Architecture 5

5.1 An process overview of the bug localization based on AdaBL(+GL) 22
5.2 AdaBL model architecture 25
5.3 Procedure to construct DFG from the source code 28
5.4 AdaBL+GL model architecture 30

List of Tables

2.1 List of Recent Researches on DL based File Level Bug Localization 7
2.2 List of Recent Researches on DL based Bug Localization 8
2.3 Replication Package availability of the BL Researches Papers 10

3.1 Information on the bug reports in each project 13
3.2 Interpretation of Cliff's delta value [11] 16

4.1 Comparison between rVSM [67] and our replication of rVSM 18
4.2 Comparison between DNNLOC [30] and our replication of DNNLOC . . . 20
4.3 Comparison between Globug [30] and our replication of Globug 21

6.1 Performance comparison (Top@k, k=1,5,10) of three state-of-the-art methods (DNNLOC, Globug, rVSM) 35
6.2 Performance comparison (MRR and MAP) with three state-of-the-art methods (DNNLOC, Globug, rVSM) 36

7.1 Comparison between Globug & AdaBL+GL predictions at Top 5 39

Chapter 1

Introduction

1.1 Bug Localization

In today's world, it is hard to find a field that isn't benefiting from the advancement of technology. Software development is no exception. With the help of automated techniques, developers are able to spend less time on tedious and time-consuming tasks, like bug localization. These techniques are built using either Information Retrieval (IR) based techniques [67, 59] or using Machine Learning (ML) or Deep Learning (DL) [24, 40, 62].

Ultimately, bug localization is incredibly valuable for developers while debugging. Through IR or ML/DL-based bug localization, developers are able to reduce the amount of time spent on locating and fixing bugs – making their jobs easier and faster!. IR-based techniques use a set of keywords related to the bug in order to identify the buggy source code. Whereas the ML and DL-based techniques use historic bug data, i.e, bug reports along with associated buggy source code files, to localize the buggy source code for the new bugs. ML and DL-based techniques can be even more effective than IR-based techniques because it takes into account the non-linear relationship between a bug report and the source code files, not just bug report keywords.

1.2 Cross-Project Bug Localization

Recent research [24, 40, 27, 53] applied deep-learning techniques to bug localization, and the results have been promising. However, there are still some challenges that need to be

addressed. Most of the research has focused on projects with high-quality and quantity datasets for modeling as a good dataset is essential for accurate training of models. It is also important to consider how well these models perform on new datasets (projects) that may not have comprehensive historic bug data and this method is called cross-project bug localization.

The software engineering research community is actively researching the subject of cross-project bug localization. The closest research [24] toward cross-project bug localization is by Huo et al. Their goal is to determine if models jointly trained with two projects (source and target) can extract transferable features from both projects. And then be used for bug localization in the target project without retraining.

To our knowledge, no one in the software engineering research has looked into pure cross-project bug localization, in which a model is trained on one project and directly applied to a new project without retraining for bug localization. We present Adaptive Cross-Project Bug Localization (AdaBL), a deep transfer model that can be trained with data from one project and transferred to many others. However, when it comes to transferring knowledge from one project to another, things can get a bit more complicated. The direct transfer of learning from high quality and quantity projects brings with it the information that is exclusive to that project [56]. And this effect is called the negative transfer effect. By learning project-specific information and programming language similarity separately, AdaBL avoids the negative transfer effect.

The recent study's [36] findings in code search show a model trained in one project can be used on another project for semantic code research without retraining. These findings were promising and encouraged us to build knowledge transferable models for cross-project bug localization. We performed the bug localization in three different project settings: within the project, partial cross-project, and cross-project.

1.3 Source Code is Multi-Faceted

Source code is multi-faceted meaning source code can be represented in more than one format in the DL models. Some of the current research leverages the multifaceted representation of programming language to bridge the semantics between natural language and the programming language. For example, source code can be represented as sequential tokens [64, 40], as a dimensional feature [30], as Abstract Syntax Tree (AST) [65] . But none of the representations can represent the structural relationship that exists between source code tokens.

To our knowledge, one has used Data Flow Graph (DFG) to represent source code in the bug localization. We employ Graph Neural Network (GNN) to learn from the DFG. DFG is also less complex than traditional hierarchical representation [21] and this property makes DFG a good input for GNN. In this thesis, for cross-project bug localization, we propose to combine AdaBL and graph learning to build a two-parallel layer model AdaBL+GL.

1.4 DL-based Bug Localization Literature Survey

We conducted a survey as part of the development of this project, which provides a complete summary of bug localization research utilizing DL approaches over the last six years. This survey aims to group DL-based bug localization research by dataset settings (within project, partial cross-project, and cross-project) and replication package availability. Our goal is to give a bird's-eye view of dataset settings followed by the state-of-the-art DL bug localization. And the current state of reproducibility and public access to the research.

1.5 Contributions

The main contributions of this project are,

- Adapting AdaCS in order to build an adaptive model for cross-project bug localization
- Employing DFG for source code representation for capturing the semantics of the source code variables which is essential in understanding the semantics similarity between bug reports and the source code
- A compilation of DL-based bug localization research with a brid's-eye-view of dataset settings followed and replication package availability

1.6 Research Questions

R1. How do different dataset settings impact the performance of the bug localization model?

Many different factors can impact the performance of a bug localization model. One important factor is the dataset setting. In particular, the size and composition of the training dataset can significantly impact how well the model performs. Another important factor is the type of bug localization algorithm used. Different algorithms may perform better or worse depending on the input features, training strategy, .etc.

We will test our proposed models and the benchmark baseline models in three dataset settings in order to answer this research topic. We observed that models trained in within project and partial cross-project settings outperform models trained in a cross-project setting.

R2. Does the performance of ML/DL-based localization for a project vary depending on the dataset setting?

We will analyze whether the dataset setting impacts the ML/DL-based model performance in a project. We trained the models under different datasets settings. We discovered that a different model and dataset setting pair combo performs better for each project. There is no one-model-fits-all answer to this question.

The thesis is divided into the following chapter. The research papers from the literature survey are briefly discussed in Chapter 2. The dataset, dataset settings, and metrics used for training and evaluating the models are described in Chapter 3. The results of our replication of the benchmark baseline models are discussed in Chapter 4. We introduce our proposed methodology in Chapter 5. In Chapter 6, we will analyze the results. Analyze and compare the findings of the better performing model with the worst performing model in Chapter 7. We list the potential threats to the validity of our results in Chapter 8. In Chapter 9, we conclude our thesis by summarising our contributions and listing the potential future works.

Chapter 2

Related Works

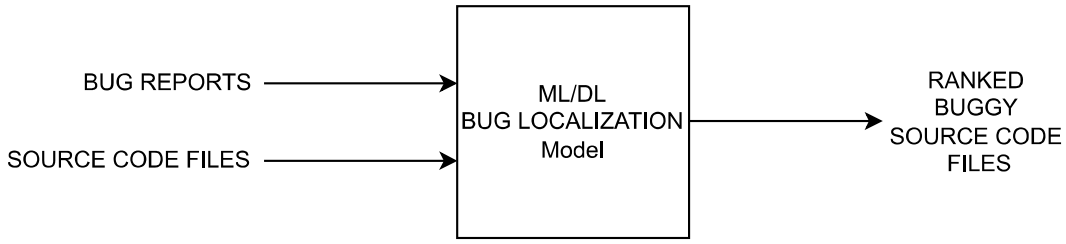

Figure 2.1: ML/DL Bug Localization Model Architecture

Several studies have been published in the software engineering literature in the past, covering different methods used for bug localization [15, 60, 72, 45]. These surveys provide a complete evaluation of the literature for bug localization, covering IR and MLDL techniques. However, none of the surveys looks into the experimental settings, particularly for DL approaches. As DL approach depends heavily on the data, and the experimental setting plays a significant role in model performance.

Moreover, most of the surveys were done till 2016, and up to our knowledge, no further study covers the recent developments in DL-based bug localization. In this survey, our motivation is to compile the experimental settings, evaluation framework, and replication availability from the recent bug localization research, particularly in MLDL-based bug localization research. Table 2.1 and 2.2 shows our complication of DL based bug localization

research. For this thesis, we consider only DL-based bug localization research that takes bug reports and source code files as primary input and outputs ranked buggy source code files, as shown in Figure 2.1.

In this survey, we extracted research published from 2016 to 2021 that used the keywords "bug," "localization," "fault," "bug localization," "fault localization," and "bug report" from IEEE, Elsevier, Springer, and DL ACM. From 435 research matching the keywords after card sorting, we identified 17 research on ML/DL-based bug localization. We then performed a snowballing analysis of the references two levels deep to identify 30 research papers on bug localization using DL techniques. For this project, we have selected only file-level DL-based bug localization research papers from 2016, which are listed in Table 2.1.

IR+DL

Lam et al.'s [29] model HyLoc combined IR and DL techniques to great effect, outperforming traditional IR-based bug localization models. The experiments conducted by the researchers showed that IR and DL complemented each other, and together, they performed better than an individual model. The results of Lam et al. [30] showed using a combination of IR and DL (IR+DL) can lead to better results than relying on a single model (IR or DL) where they have combined rVSM [75] with DNN.

Similarly, Wang et al. proposed Multi-Dimension Convolutional Neural Network (MD-CNN) for bug localization [53] that combined IR and DL. The proposal involves extracting five statistical features from the bug reports and source codes and then applying MD-CNN to the extracted statistical features. In the research [10] Cheng et al. extract statistical features from multiple sources such as bug reports, source code, file names, and stack traces and then apply the deep neural network (DNN) to the statistical features for extracting the nonlinear relationship.

Source Code as a sequential inputs

Ye et al. [68] explored word embedding in bug localization, specifically the lexical gap between source code tokens and bug reports. They found that while word embedding does help to bridge the gap, it also causes lexical mismatch on bug localization. Their findings provided valuable insights into using word embeddings for briding programming language and natural language. The word embeddings modeled using project-specific data points were found to be performed similarly to word embeddings modeled with Wiki corpus [68].

Table 2.1: List of Recent Researches on DL based File Level Bug Localization

Year	Authors	Title	Method
Partial Cross-Project			
2021	Huo et al [24]	Deep Transfer Bug Localization	DL
Within Project			
2021	Miryeganeh et al [40]	GloBug: Using global data in Fault Localization	IR
2020	Cheng et al [10]	A Similarity Integration Method based Information Retrieval and Word Embedding in Bug Localization	IR+DL
2020	Yuan et al [71]	DependLoc: A Dependency-based Framework For Bug Localization	IR+DL
2020	Kim et al [27]	Feature Combination to Alleviate Hubness Problem of Source Code Representation for Bug Localization	DL
2020	Wang et al [53]	Multi-Dimension Convolutional Neural Network for Bug Localization	IR+DL
2019	Liu et al [37]	Convolutional Neural Networks-Based Locating Relevant Buggy Code Files for Bug Reports Affected by Data Imbalance	DL
2019	Liang et al [35]	Deep Learning With Customized Abstract Syntax Tree for Bug Localization	DL
2019	Xiao et al [64]	Improving bug localization with word embedding and enhanced convolutional neural networks	DL
2019	Liu et al [38]	Mapping Bug Reports to Relevant Source Code Files Based on the Vector Space Model and Word Embedding	IR+DL
2018	Xiao et al [62]	Improving Bug Localization with Character-Level Convolutional Neural Network and Recurrent Neural Network	DL
2018	Xiao et al [63]	Machine translation-based bug localization technique for bridging lexical gap	IR+DL
2017	Lam et al [30]	Bug Localization with Combination of Deep Learning and Information Retrieval	DL
2017	Xiao et al [65]	Improving Bug Localization with an Enhanced Convolutional Neural Network	DL
2016	Ye et al [68]	From word embeddings to document similarities for improved information retrieval in software engineering	IR+DL

Table 2.2: List of Recent Researches on DL based Bug Localization

Year	Authors	Title	Method
2021	Cao et al. [7]	Automated Query Reformulation for Efficient Search Based on Query Logs From Stack Overflow	IR+DL
2021	Chen et al. [9]	GLIB: towards automated test oracle for graphically-rich applications	DL
2021	Su et al. [51]	OwlEyes-online: a fully automated platform for detecting and localizing UI display issues	DL
2020	Liu et al. [39]	Owl eyes: spotting UI display issues via visual understanding	DL
2019	Li et al. [34]	Improving bug detection via context-based code representation learning and attention-based neural networks	DL
2019	Xia et al. [61]	BugIdentifier: An Approach to Identifying Bugs via Log Mining for Accelerating Bug Reporting Stage	DL
2019	Zhang et al. [73]	FineLocator: A novel approach to method-level fine-grained bug localization by query expansion	DL
2018	Pérez et al. [47]	Fragment retrieval on models for model maintenance: Applying a multi-objective perspective to an industrial case study	IR+DL
2018	Zhong et al. [74]	Mining repair model for exception-related bug	IR+DL
2017	Böhme et al. [5]	How Developers Debug Software — The DBGBENCH Dataset	DL
2017	Chaparro et al. [8]	Detecting missing information in bug descriptions	DL
2016	Dam et al. [13]	DeepSoft: a vision for a deep model of software	DL
2016	Gu et al. [20]	Deep API learning	DL
2016	Wang et al. [55]	Automatically learning semantic features for defect prediction	DL
2016	Zhang et al. [72]	A Literature Review of Research in Bug Resolution: Tasks, Challenges and Future Directions	DL

To further bridge the gap between programming language and natural language, Xiao et al. extend their previous research [65] by introducing character-level embedding and long short-term memory (LSTM) in [62]. To further reduce the lexical gap and to preserve the difference between source code tokens and bug report tokens, Xiao et al. in [64] extended their previous work [65] by using two different vectors representations Sent2Vec and Word2Vec for representing source codes and bug reports.

Though prior studies modeled word embedding using only project-specific data points, Miryeganeh et al. proposed Globug [40], in which they use global datasets while training. Globug combined both the term frequency-inverse document frequency (TFIDF) model and Doc2Vec [32] model, which utilizes direct and indirect relevance scores to rank buggy source code files.

Source code tokens extracted from Tree represented as a sequential inputs

Xiao et al., in their work [65] to capture the structural information of the source code, used Abstract Syntax Tree (AST). They applied TF-IDuF to filer common words from bug reports. Moreover, Convolution Neural Network (CNN) is used to extract features from the skip-gram-based vector representation of source code tokens and bug reports. Liu et al. in [38] to learn surface lexical and semantic similarity between source code and bug report VSM and word2vec were used, respectively. And to dynamically modify the weights of the vector representation, authors utilize a part-of-speech (POS) tagger in the bug report and AST of source code to extract important keywords and class names, method names from the bug report, and source code, respectively. To further establish a robust semantic relationship between the bug reports and source code along with the surface lexical similarity, Liu et al. have extended their previous work [38] by applying Doc2Vec and Word2Vec for bug reports and Word2Vec for source codes.

Source code as a Tree structure

Liang et al. modeled CAST [35] with the motivation to exploit the hierarchical nature of source code. Authors utilized TBCNN [43] on AST to learn the underlying hierarchical relationship between source code tokens. CNN captured the semantic similarity between the bug report and source code. ASTs are great for capturing the hierarchical relationship between tokens in a source code file but cannot capture their interrelationship.

Source code as a Graph

Kim et al., in their recent research [27] present an interesting approach to capturing the interrelationship between source code tokens. By constructing an AST graph and using a compositional embedding model [50], they can generate a vector representation for the graph that accurately reflects the relationships between tokens. Yuan et al., in their research [71] constructed vector representation from class dependency graph using Ant colony algorithm.

Partial Cross-Project

Previously discussed DL research papers are very effective in bug localization. However, those DL research requires a lot of data in order to train. In many cases, particularly for new projects, there is not enough historic bug data [76] available to train a DL model for bug localization. Huo et al. showed in their work TRANP-CNN [24] that transferable features can be extracted through joint learning on the source and target projects, and then the trained model can be applied to the target project without retraining.

There is a lot of research on bug localization, but it has been limited to isolated problems so far. Individual techniques have been developed to utilize source code structural information, cross-project bug localization, or IR+DL for improved lexical and semantic learning. However, no research has yet addressed all these isolated scenarios in bug localization. In this project, we propose a model that utilizes source code structural information, cross-project bug localization, and IR+DL for improved lexical and semantic learning. We believe that this is the best way to achieve industry standards in bug localization.

Reproducibility and Public Access To The Research

Table 2.3: Replication Package availability of the BL Researches Papers

Year	Authors	Replication Package	Processed Data Availability	Data Processing Script availability	Feature Extraction Script Availability	Model Training Script Availability	Trained Model Availability
2021	Miryeganeh et al [40]	Globug [41]		Yes	Yes	Yes	
2020	Wang et al [53]	MD-CNN [18]				Yes	

To compare the replication package availability and public access to research, we filter the research papers based on the replication package that is available for download publicly. Our results, depicted in Table 2.3, show that the replication package is available for only

two research papers out of sixteen research papers. In which only Miryeganeh et al. [40] work is fully replicable because of the availability of data processing, feature extraction, and model training scripts. However, we cannot replicate Wang et al. [53] work due to the absence of the feature extraction script.

Table 2.3 indicate that the reproducibility of deep learning-based bug localization models is not high and there is a lack of reproducibility in the bug localization research community. In order to improve the reproducibility of bug localization research, we urge all researchers to make their replication packages available to others.

Chapter 3

Dataset

3.1 Motivation

In this project, we propose that bug localization models could be transferred to a new project without re-training the models on project-specific data points. Creating a cross-project bug localization model dataset plays a very important role as the quality of the dataset determines the DL models learning capability to extract the transferable features from one project, which can be transferred to new projects. A high-quality cross-project bug localization model dataset is essential for two reasons:

- it allows us to evaluate how well our DL models are able to learn and generalize across different projects, and
- it provides us with a training set that we can use to improve our bug localization models. In this chapter, we will describe how we collected and prepared the data for our cross-project bug localization research.

3.2 Dataset

Benchmark datasets are important tools for evaluating the effectiveness of bug localization techniques. It is important to use benchmark datasets that are representative of real-world applications. In this project, we have used the dataset by Ye et al. [67] which is large

Table 3.1: Information on the bug reports in each project

Project	Time Range	# of Bug Reports	# Avg. of Fixed Files
AspectJ	Mar/2002-Jan/2014	593	4
Birt	Jun/2005-Dec/2013	4,178	2.8
Eclipse Platform UI	Oct/2001-Jan/2014	6,495	2.8
JDT	Oct/2001-Jan/2014	6,274	2.6
SWT	Feb/2002-Jan/2014	4,151	2.1
Tomcat	Jul/2002-Jan/2014	1,056	2.4

and representative of real-world scenarios as it is collected from open-source java projects. This dataset will be useful for benchmarking as it has been used in previous DL-based bug localization research [71, 64, 63, 65] . Table 3.1 shows the information of the dataset.

- **AspectJ** is an aspect-oriented programming (AOP) extension to the Java programming language
- **Birt** provides reporting and business intelligence capabilities
- **Eclipse UI** includes building blocks of the Eclipse user interface
- **JDT** is a set of tools that support Java development in Eclipse IDE
- **SWT** is the core of Eclipse's native user interface
- **Tomcat** is an application server written in Java

3.2.1 Dataset settings

Dataset settings are important factors that can affect the performance of DL models. In this project, we will compare the performance of the proposed model with state-of-the-art bug localization in three different Dataset settings: within project setting, the partial cross-project setting, and the cross-project setting. Datapoints are sorted in chronological order based on the bug report timestamp. This setting has a few benefits:

- It allows using more recent bug reports to test the accuracy of your model, which is essential since models tend to become less accurate over time as they get "stale."

- It also ensures that the model is trained using older bug reports which helps generalization performance on new bug reports.

Within project setting: In this setting, 80% of the dataset is used for training, 20% is used for testing, and all bug reports used from training and testing are from only one project. Using this setting, we can compare the proposed model performance with state-of-the-models. In addition, using a common setting can help to ensure that results are comparable and accurate.

Partial cross-project settings: In this setting, the training dataset contains only 20% of the dataset is used from the target project and 80% of data from the label-rich projects. This setting is followed in the previous research [24]. [24] proposed joint feature extraction for which at least a few label-rich data is required.

Cross-project settings: Cross-project setting is essential to accurately measure the performance of bug localization on unseen data from new projects. The training dataset contains 100% of data from label-rich projects in this setting. And test dataset from the project not used in training. This is to replicate the real-world scenario where training from label-rich data and utilizing the model on the new project with little or no data.

3.2.2 Data collection

The dataset by Ye et al. [67] comprises data collected from publicly available open-source projects. The dataset is compiled by extracting the bug reports from the Bugzilla issue trackers and the associated source code file from the Git version control systems of the six open-source projects.The dataset was published and publicly available [1] thanks to the authors.

3.3 Experiments Setup

We investigate the effectiveness of different bug localization models that are trained and evaluated on a single cloud instance with 64 GPU RAM, Intel Xeon Processor (8 X 2.10GHz), and NVIDIA V100 GPU 32Gb.

3.3.1 Evaluation and Metrics

We evaluated the baseline models and proposed models in all three experiment settings. To compare the results of all models, we used Top-K, Mean Average Precision (MAP), and Mean Reciprocal Rank (MRR) as metrics during the evaluation.

3.3.1.1 Top-K

It is the number of bugs reports for which associated buggy source files are located in the top k source file, sorted based on the similarity. For Top-1, even if just one of the top results contains a buggy file relevant to the bug report, then we can confidently say that the bug is indeed located in that file. The Top-1 Ranked is determined by the percentage of all such found bugs.

3.3.1.2 MRR

The Reciprocal Rank (RR) is an information retrieval metric that estimates the reciprocal of the first relevant document's rank. If a relevant document was obtained at rank 1, is set to 1, if not, it is set to 0.5 if a relevant document was found at rank 2, and so on. This measure is referred to as the Mean Reciprocal Rank (MRR) when it is averaged across queries [12]. MRR is often used to measure the performance of the IR model in which just one relevant document must be located. Bug localization is an IR task where a bug report is the query and source code files are the search space MRR is predominately used in bug localization research for evaluation.

3.3.1.3 MAP

MAP is a metric to verify the performance of the IR model that retrieves many relevant documents for each query. In bug localization, as there exists more than one related source code file for each bug report, we employ MAP for evaluation. MAP is unlike MRR, which only considers one retrieved relevant document. MAP considers all the relevant documents that are retrieved to provide an accurate measure of performance.

3.3.1.4 Wilcoxon signed-rank test

The Wilcoxon signed-rank test [58] is a powerful tool that can be used to measure the statistical significance of differences between two models. We applied the Wilcoxon signed-

rank test at a 95% significance to performance metrics such as Top-k, MRR, or MAP, it can help us determine which model is best.

3.3.1.5 Cliff's delta (δ)

Table 3.2: Interpretation of Cliff's delta value [11]

Cliff's Delta ($\lvert\delta\rvert$)	**Effectiveness Level**
$0.000 \leq \lvert\delta\rvert < 0.147$	Negligible
$0.147 \leq \lvert\delta\rvert < 0.330$	Small
$0.330 \leq \lvert\delta\rvert < 0.474$	Medium
$0.474 \leq \lvert\delta\rvert \leq 1.000$	Large

The Cliff's delta (δ) [11] is then used to interpret these differences. Table 3.2 shows how different delta values are interpreted. A $|\delta|$ value of 0 indicates that there is no difference between the two models under consideration. In contrast, a $|\delta|$ value of 1 means that one model outperforms the other model across all datasets in terms of Top-K/MAP/MRR.

As mentioned before, top-k, MRR, and MAP are the three metrics we used to evaluate the performance of our models and benchmark baseline models. In the next chapter, we will discuss the results of benchmark baseline models replications.

Chapter 4

Replication of Benchmark Baselines

In this project, we focus on performing cross-project bug localization by combing state-of-the-art techniques DL and GNN. We compare the performance of our proposed model with benchmark baseline models such as rVSM [67], DNNLOC [30], and GloBug [40]. When comparing the efficacy of different models, it is important to use common benchmark baselines. In order to ensure a fair comparison, we replicated the benchmark baseline models using the hyperparameters provided by the authors of each model in their respective research articles.

- **rVSM** is an altered version of the TFIDF model that considers the source code's token length while calculating the similarity between the source code and bug report.
- **DNNLOC** utilizes DNN to extract the non-linear relationship between source code and bug reports, encoded as vectors using revised Vector Space Models.
- **GloBug** proposed training using a global dataset with a DL+IR model and then using the trained model for bug localization. According to the authors, applying the DL+IR model to the vector space will aid in capturing semantic and syntactical similarities between problem reports and source code.

Among these methods, rVSM is a state-of-the-art tIR-based method. Boht DNNLOC and Globug are state-of-the-art IR+DL method.

4.1 rVSM

rVSM first creates a combined vocabulary from all the tokens extracted from source code and bug reports. Then from the combined vocabulary, both inverse document frequency (IDF) and term frequency-inverse document frequency (TF-IDF) are combined. The relevance of a token is estimated in the IDF by taking into account the frequency of tokens in the combined vocabulary.

$$g(\#\text{ terms }) = \frac{1}{1 + e^{-N.(\#\text{ terms })}}$$

TFIDF is computed for each token of bug reports and source code through the dot product between each token frequency in that bug report or source code and the frequency count of that token in the combined vocabulary.

$$\text{Similarity } = \cos(br, sc) = \frac{V_{br} * V_{sc}}{(\|V_{br}\|) * (\|V_{sc}\|)}$$

Finally, the similarity between both source code and the bug report is calculated through the dot product between the cosine similarity $\cos(br, sc)$ of the bug report and source code vectors, and the tokens counts in a source code file g(# terms).

$$rVSM\,\text{Score}(br, sc) = g(\#\text{ terms }) \times \cos(br, sc)$$

Table 4.1: Comparison between rVSM [67] and our replication of rVSM

Dataset	**Top 1**		**Top 5**		**Top 10**		**MRR**		**MAP**	
	rVSM	Our Replication of rVSM	rVSM	Our Replication of rVSM	rVSM	Our Replication of rVSM	rVSM	Our Replication of rVSM	rVSM	Our Replication of rVSM
AspectJ	20.1	17	47.7	42.6	57	55.8	0.32	0.26	0.22	0.23
Birt	11.1	11.5	24.9	33.6	32.1	47	0.18	0.21	0.14	0.2
Eclipse	26.5	10.9	49.3	30.7	60.1	44.2	0.37	0.23	0.31	0.21
JDT	19.1	15.2	40.2	39.7	51.2	54.1	0.3	0.26	0.23	0.24
SWT	19.3	16.1	38.3	44.1	51.1	57.7	0.28	0.28	0.25	0.27
Tomcat	35.5	32.6	61.8	61.4	71.1	76.8	0.48	0.45	0.43	0.42
mean	21.93	17.22	43.7	42.02	53.77	55.93	0.32	0.28	0.26	0.26
p-Value	-	>0.05	-	>0.05	-	>0.05	-	>0.05	-	>0.05
δ	-	0.5	-	0.17	-	0	-	0.44	-	0.11
Improved% rVSM	-	+24.06	-	+3.92	-	-3.94	-	+13.33	-	+0.0

Authors trained and evaluated rVSM on the dataset by Ye et al. [67]. Table 4.1 shows the comparison between our replication of rVSM and the results from the rVSM research

paper. From the table, we can observe that our replication and the original rVSM results are similar (with p-Value > 0.05 and small effect size $\delta < 0.2$) in term of top-5, top-10, and map. We are using the publicly available implementation [15] for rVSM because the official replication package for rVSM has not been provided.

4.2 DNNLOC

Based on the feature combinators, Lam et al. [30] proposed DNNLOC based on DNN. A feature combinator is a module that takes one or more input layers and produces one or more output layers. More specifically, a feature combinator combines different types of inputs into a single feature vector which is useful when working with DNN. And in DNNLOC, using feature combinators, the authors combine six different input features. The six features extracted from source code and bug reports are the following:

1. *Text Similarity:* Tokens from both bug reports and source codes are separated using whitespace and based on the camel case. Then weights for both bug reports and source codes tokens are computed using TFIDF. Finally, vectors of bug reports and source codes are used to calculate the text similarity between source codes and bug reports.

2. *Collaborative Filtering:* Collaborative Filtering identifies whether a new bug report is similar to any existing reports. The score is computed by measuring the textual similarity between a source code file, the new bug report, and the previous bug reports associated with the source code.

3. *Bug Fixing Recency:* Scoring is based on the research [26] that claims developers are more likely to fix recently closed bugs than closed bugs in the past. So for the source code associated with recently fixed bugs, the bug fixing recency is 1, whereas the source code files related to older bug reports have a smaller bug fixing recency score.

4. *Bug Fixing Frequency:* Bug Fixing frequency is the count of all older bug reports in which a source code file is associated.

5. *Class Name Similarity:* The cosine similarity between the classes mentioned in the bug report and those in the source code file is used as the class name similarity score.

6. *Relevance Score:* Relevance score is computed using DNN based Autoencoder architecture for which bug reports and source code tokens are combined and given as sequential inputs.

Table 4.2: Comparison between DNNLOC [30] and our replication of DNNLOC

Dataset	Top 1		Top 5		Top 10		MRR		MAP	
	DNNLOC	Our Replication of DNN	DNNLOC	Our Replication of DNN	DNNLOC	Our Replication of DNN	DNNLOC	Our Replication of DNN	DNNLOC	Our Replication of DNN
AspectJ	47.8	33.2	71.2	51.6	85	56.6	0.52	0.34	0.32	0.26
Birt	25.2	18.7	42.2	33.6	50.9	47.5	0.28	0.21	0.2	0.19
Eclipse	45.8	33.4	70.5	56.7	78.2	68.5	0.51	0.43	0.41	0.4
JDT	40.3	27.8	65	41.9	74.3	56	0.45	0.28	0.34	0.25
SWT	35.2	29.5	69	59	80.3	69.4	0.45	0.41	0.37	0.38
Tomcat	53.9	32.2	72.9	61.1	80.4	76.6	0.6	0.44	0.52	0.42
mean	41.37	29.13	65.13	50.65	74.85	62.43	0.47	0.35	0.36	0.32
p-Value	-	<0.05	-	<0.05	-	<0.05	-	<0.05	-	>0.05
δ	-	0.72	-	0.78	-	0.67	-	0.78	-	0.17
Improved% DNNLOC	-	+34.72	-	+25.01	-	+18.09	-	+29.27	-	+11.76

Authors have modeled bug localization as a classification model. They use DNN to capture the non-linear relationship between the six features extracted from the source code and bug report pair. Finally, all the features are given as input to the DNN model, which outputs a score between 0 and 1, based on which we can classify whether the source code is associated with the bug report or not. Authors used the dataset by Ye et al. [67] for training and evaluation of DNNLOC.

Because the official replication package for DNNLOC is not published, we are utilizing the publicly available implementation [15] for DNNLOC. Table 4.2 shows the comparison between our replication of DNNLOC and the results from the DNNLOC research paper. The table shows that our replication and the original DNNLOC results are similar for Birt, Eclipse, and SWT projects. Additionally, both our replication and original DNNLOC perform similarly at the map (with p-value > 0.05 and small effect size $\delta < 0.2$).

4.3 Globug

Miryeganeh et al. discuss how using a global dataset can improve the learning of textual similarities in bug localization in their research Globug [40]. In which they utilize the dataset outside the current project to better learn the text similarities between source code and bug reports. The authors have proposed a bug localization model which combines TFIDF and Doc2Vec. First, the authors create global data by combining all the tokens from the source code. Then from the global data author computes direct relevancy and indirect relevancy using both TFIDF and Doc2Vec.

Direct Relevancy: The similarity between a bug report and source code file computed using the TFIDF model trained on the global source code files. For TFIDF computation,

the term frequency is calculated based on the current project, and IDF is calculated over the global dataset. Finally, direct relevancy is calculated by computing the cosine similarity between source code and bug report pair using the TFIDF model.

Indirect Relevancy: The global dataset was used to train the Doc2Vec model. The Doc2Vec model is then used to compute the cosine similarity between the current project bug report and the global dataset bug reports. According to the author, because the Doc2Vec model was trained on a global source code dataset, the bug report vector created by Doc2Vec links the bug report to the source code file in the current project.

Table 4.3: Comparison between Globug [30] and our replication of Globug

Metric	Globug	Our Replication of Globug
MRR	0.556	0.447
MAP	0.426	0.396

Finally, the direct relevancy and indirect relevancy scores are combined using a weighted average for generating source code files related to the bug report. The authors have used the Bench4BL [33] dataset for the training and testing of Globug for bug localization. Table 4.3 shows the comparison between our replication of Globug and the results from the Globug research paper. We can see from the table that our Globug replication performs almost identically to the original Globug on the Bench4BL dataset. We suspect the difference in the MAP and MRR metric in our replication is due to the project version difference. The Globug authors have not mentioned the specific version for each projects in the Bench4BL dataset.

We addressed the working of each benchmark baseline model in this chapter, as well as the results of our replication of the benchmark models. We'll go over our approach to addressing cross-project bug localization in detail in the next chapter.

Chapter 5

Methodology

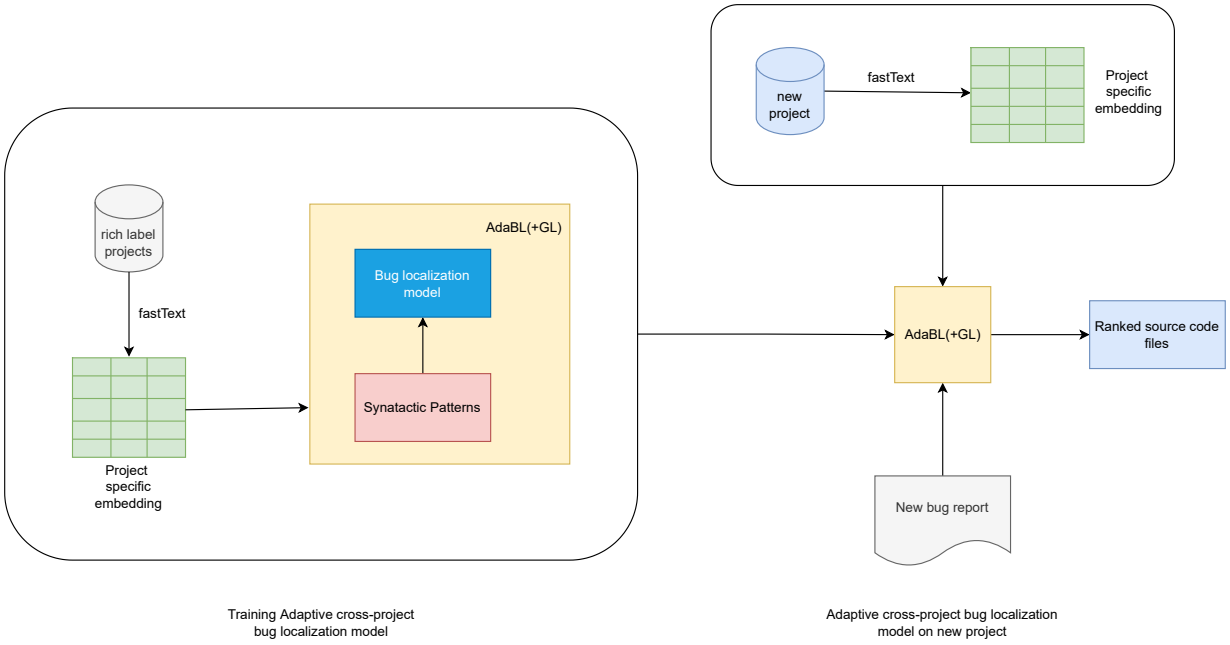

Figure 5.1: An process overview of the bug localization based on AdaBL(+GL)

In this thesis, we focused on performing cross-project bug localization through an adaptive model, which can be trained with data from label-rich projects and transferred to many other projects. We then evaluated the model against the benchmark models selected from

the literature survey. The general methodology involves representing source code in the same vector space as the bug report and then using DL methods to capture the non-linear relationships between source code and bug reports illustrated in figure 2.1.

We modeled two cross-project bug localization models. First, we built an adaptive model for cross-project bug localization (AdaBL) adapted from the research [36] in which we treated both source code and bug reports as sequential input. In the second model, we treated the source code as a graph. By utilizing the structured information of source code, we implemented a combined graph learning with the adaptive model for cross-project bug localization (AdaBL+GL). We wanted to understand the impact of graph learning on bug localization by doing this.

5.1 Motivation

There are several different approaches to bug localization, but most of them rely on using ML/DL models trained on data from a specific codebase. The idea is that these models will be able to learn common patterns between source code and bug reports, and then for new bug reports, these models can localize where bugs are in the source code. However, transferring these models from one codebase to another can be very costly and time-consuming since they often need to be re-trained using data from the new codebase.

Adaptive models offer a potential solution to this problem. These models are able to adapt themselves automatically based on differences between two codebases – meaning they don't need any retraining data or computing power once they have been deployed. AdaCS [36] is an adaptive model that achieves cross-project knowledge transfer in code search. AdaCS extracts transferable features through two learning objectives: identifying syntactic patterns shared by natural language and programming languages and detecting variations between programming languages across different codebases.

The software engineering community is still divided about whether adaptive strategies can be used to improve bug localization. In this project, we intend to adapt AdaCS to build an adaptive bug localization model.

5.2 Adaptive Cross-project Bug Localization

In this section, we present our proposed approach for bug localization, AdaBL, which can be trained once and then used to automatically adapt to new projects. AdaBL is

trained in a way that separates the abstract syntactic patterns from the project-specific lexical meanings. This ensures that the model can be applied to new projects without any retraining.

5.2.1 Model Architecture and implementation

Figure 5.2 shows the deep model architecture of AdaBL. AdaBL has four parts:

1. Training an unsupervised word embeddings model to capture project-specific words
2. Creating a similarity matching matrix with each of the tokens of the bug report and source code pair;
3. Extracting syntactic patterns from the similarity matrix using deep neural networks.
4. Predicts the score for each source code file, and then source code files are ranked based on the scores.

The details of these four parts are introduced in the following sections.

5.2.1.1 Capturing Project-Specific Words

The unsupervised word embedding training methodology was utilized in order to capture the project-specific words. In particular, we used fastText [4] to leverage the subword modeling that is capable of learning the representation of rare project-specific words. There are many word embedding algorithms out there, but we chose fastText over them because it is more reliable. fastText uses subword information to learn the word vectors, while other algorithms use classical methods like words appearance context or counting how often they appear in a corpus. This means that the words "results" and "result" from fastText will be more similar to each other than those from other algorithms as they share some parameters while training. Additionally, because fastText learns using subword information, it is better at capturing the meaning of words than other algorithms. We modeled a new word embedding model for both source projects and target projects for training and testing. The word embedding model can be trained in an unsupervised manner.

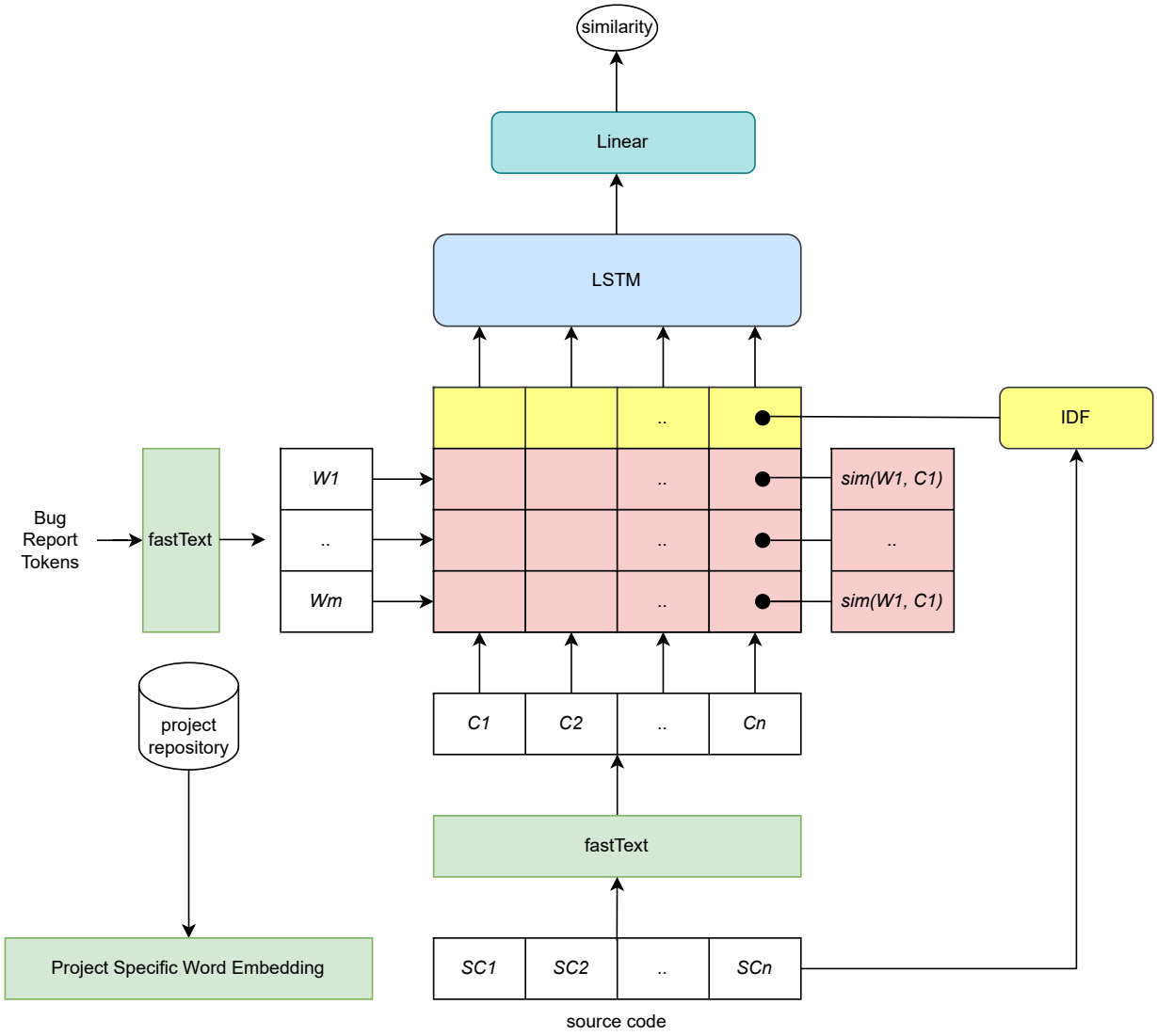

Figure 5.2: AdaBL model architecture

5.2.1.2 Similarity Matrix

The interactions between bug reports and source code tokens are captured using a lexical similarity matrix. In the matrix, each entry represents the cosine similarity between the vector representation of bug report and source code token pair. The first rows of the matrix contain the Inverse Document Frequency (IDF) value of the source code tokens to express

the specificity.

5.2.1.3 Extracting Syntactic Patterns

Cross-project syntactic patterns are captured using the LSTM model trained on label-rich bug reports and source code pairs. Additionally, negative samples were included while training to differentiate between similar and dissimilar bug reports and source code pairs. When applying the trained model to a new project, the similarity matrix can be passed as the input for predicting the similarity score.

5.2.1.4 Transferability to New Projects

The syntactic patterns of a project can be transferred from one project to another, making the application of AdaBL to a new project relatively easy. Only word embeddings need to be trained on the new project in an unsupervised manner. Then, the lexical similarity matrix can be constructed for the bug reports, and source code tokens pair with the word embedding model. Finally, the pretrained AdaBL can be utilized to score the bug report and source code pair for which the lexical similarity matrix is input. Based on the score, the source code files can then be ranked.

5.2.2 Hyperparameter Specifications

We developed AdaBL in Python using the PyTorch [44] library, an open-source deep learning framework. We build AdaBL with a two-layer LSTM with a hidden dimension of 64, with a drop rate of 5%. The ADAM optimizer with a learning rate of 0.005 and batch size of 64 was found to produce minimal training loss. We also experimented with different token sizes for bug reports and source code (128 and 256, 256 and 512, 354 and 768). Finally, we observed that the training loss is minimized when the token size is set to 354 and 768, respectively, for bug reports and source code tokens.

5.2.3 Model Training

Bug localization is the task of finding the most likely location in a program where a given bug exists. In AdaBL, we have implemented this as a classification task by comparing a bug report with the corresponding source code in the vector space and predicting whether

they are similar. If they are similar, then the predicted results will be 1; if not, then it will be 0. So the objective AdaBL is to learn similarities between bug reports and source code pairs in the vector space. The LSTM models are applied to bug reports and source code interactions matrix to learn the similarities as shown in figure 5.1. The inputs from LSTM are then fed into DNN for the final prediction which is 1 or 0.

From the figure 5.1, it is evident that AdaBL follows an adaptive bug localization workflow in order to improve bug localization. The first step is to use unsupervised learning-based methods to learn the project-specific words meaning for both the training and target projects. This helps to capture project-specific semantics. Next, we train a model to learn the general syntactic patterns using the bug report and source code pairs extracted from source projects. Finally, for bug localization for new bug reports, we use the trained model to localize on the target project and which then returns the ranked source code file as the final result.

5.3 Adaptive Cross-project Bug Localization with Graph Learning

In this section, we present AdaBL+GL in detail. AdaBL+GL can be trained once on label-rich projects and applied on new projects without retraining. AdaBL+GL as the name indicates the model architecture has two parallel layers 1. AdaBL 2. Graph Learning (GL). AdaBL+GL model follows the same training methodology, hyperparameter as AdaBL. The primary advantage of AdaBL+GL over AdaBL is its ability to encode the structural information of source code which helps to capture the inter-relationship between source code entities on top of AdaBL.

5.3.1 Model Architecture and implementation

Figure 5.3 shows the overall architecture of AdaBL+GL. AdaBL+GL solves cross-project bug localization by combining the features extracted from the data flow graph (DFG) using GNN, general syntactic patterns, and project-specific semantics through feature combinators. The use of the DNN allows for the combination of these features into a single representation. Both general syntactic patterns and project-specific semantics follow the same methodology as AdaBL.

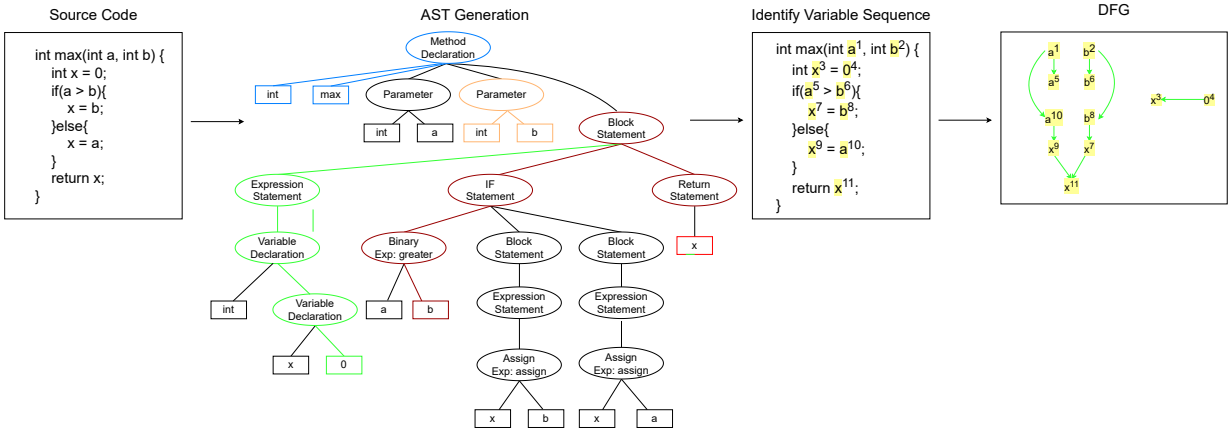

Figure 5.3: Procedure to construct DFG from the source code

5.3.1.1 DFG

As software engineering research advances, more efficient ways of representing source code are being discovered. Data Flow Graph (DFG) is one such method that has been shown to be particularly useful for deep learning. DFG is a popular technique in software engineering research [3, 21] particularly in DL based research for source code representation. Representing code as DFG provides crucial semantic information, which can help you understand the code better. The data flow graph shows how different parts of the code are related to each other; if two pieces of code share a variable, they will be connected in the data flow graph.

DFG supports understanding the semantics of code variables better. The semantic understanding of code variables is essential in bug localization because different programmers use their naming conventions instead of standard ones. Take $e = maxValue - minValue$ as an example, the variable e doesn't follow the convention and semantic understanding is hard. However using DFG, it can be understood that the value of node e can be derived as the computation involving nodes $maxValue$ and $minValue$, i.e., the value of e is derived from the $minValue$ and $maxValue$. Another example from Figure 5.3, in which there are four variables with the same name (x^3, x^7, x^9, and x^{11}) but with distinct semantics. The graph in the image depicts the relationship between these variables and enables x^{11} to focus on x^7 and x^9 rather than x^3.

We leverage the recently proposed DFG extraction methodology [21] from the given source code. Figure 5.3 depicts the DFG extraction from a source code. First, to understand the dependencies between variables in a source code, we first generated AST using

tree-sitter [2]. Then we identify variable sequence by traversing the AST and keeping track of all variable names encountered. The leaves of the tree represent variables. Variables are the nodes in a DFG graph, and edges represent the dependencies extracted from the AST for those particular variables. This means that all of the dependency information for a particular variable is contained within one edge. For example, if x is set to expr, then edges from all variables in expr to x are added to the graph. Finally, the DFG of a source code is made up of all variables as nodes and the dependency relation between the variables as edges. As shown in the figure 5.4, DFG is one of the inputs to AdaBL+GL, which supports better source code understanding in DL models.

5.3.1.2 Node Embeddings

After constructing DFG where nodes represent the variable of source codes and edges are all the dependencies for each variable we transform each node into vectors using fastText. fastText is trained in an unsupervised manner on the project's source code files as previously mentioned in the section 5.2.1.1.

5.3.1.3 Graph Neutral Network

A GNN is a great way to learn from graphs because it allows for local neighborhood information to be aggregated and passed on to the next layer. This helps ensure that features of neighbors are considered when making decisions, which can lead to better results. There have been a lot of recent developments around the potential of GNNs for learning graphs effectively.

For AdaBL+GL we have leveraged the recently proposed graph learning method [6] which has shown promising results in terms of performance on unseen graph structures. Our AdaBL+GL GNN architecture consists of three key layers: a graph convolutional layer (GCL), a graph pooling layer (GPL), and a graph readout layer (GRL). The use of these layers allows for the efficient learning and representation of graphs.GNN is used in AdaBL+GL to capture the structural inter-relationship between code tokens in DFG. In Figure 5.4 the GNN module depicts the full pipeline of graph learning in AdaBL+GL.

The first step in this pipeline is to apply a GCL to input DFG which will learn features at different scales across the entire graph. The next step is to use GPL which will reduce the number of connections while preserving important information about nodes and the edges between them. Finally, we apply GRL that will output fixed-length flatten representation which is then used in combination with AdaBL for ranking buggy source code files.

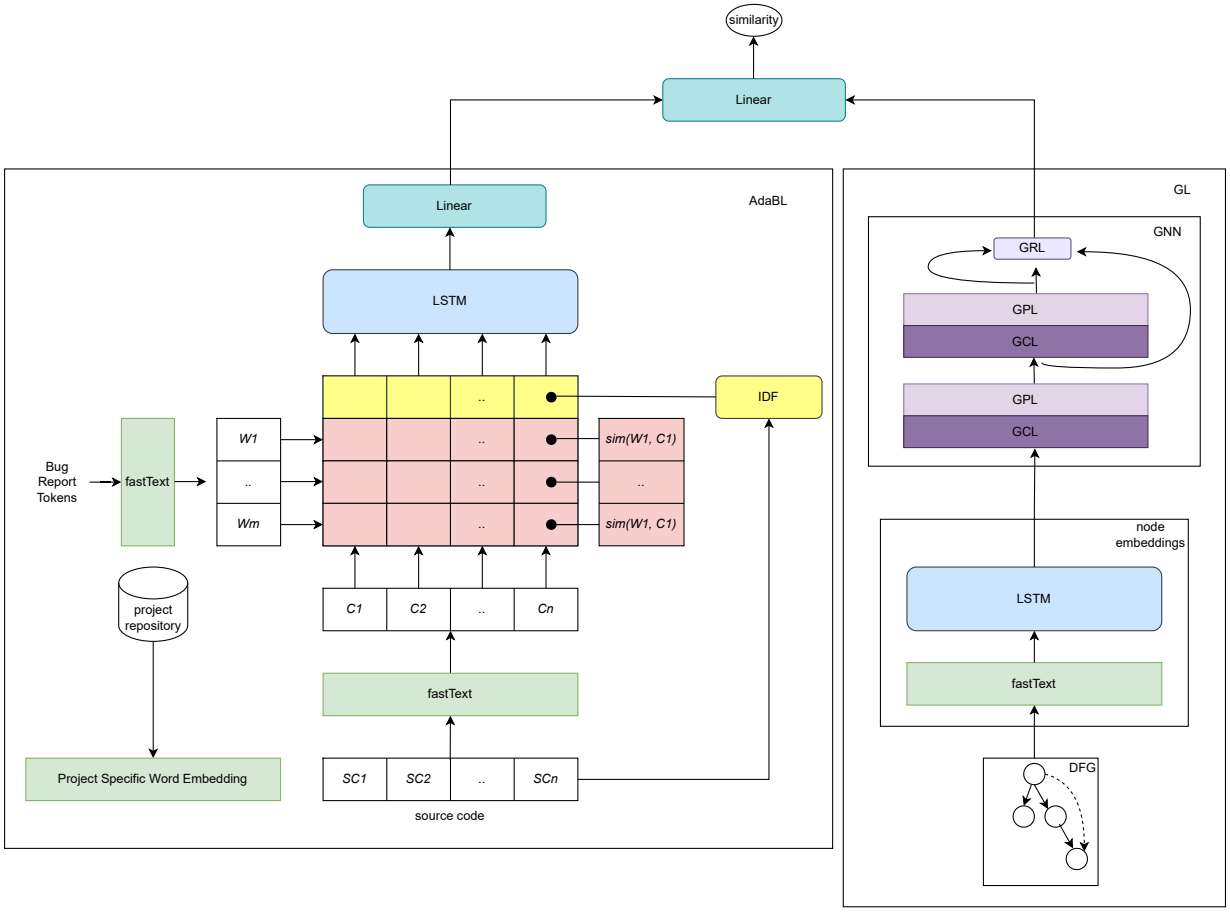

Figure 5.4: AdaBL+GL model architecture

GCL

The feature transformation for GCL follows:

$$\vec{f}^{(l)}(v) = \sigma\left(\vec{f}^{(l-1)}(v) \cdot \mathbf{W}_1^{(l)} + \sum_{\omega \in N(v)} \vec{f}^{(l-1)}(\omega) \cdot \mathbf{W}_2^{(l)}\right)$$

According to Morris et al.[42], the weight matrices $(\mathbf{W}_1)$ and $(\mathbf{W}_2)$ for each node v and its neighbor $N(v)$ respectively are different. So in the feature transformation $\vec{f}^{(l)}(v) \in \mathbb{R}^{1\times d^{(l)}}$ is the output feature of node v in the l^{th}th layer, $\sigma(\cdot)$ is the activation function

and $d^{(l)}$ denotes the dimension. The weight matrices of the l^{th}th layer are denoted by $\mathbf{W}^{(l)} \in \mathbb{R}^{d^{(l-1)} \times d^{(l)}}$.

GPL

GPL is used to reduce the size of a graph by removing $N - \lceil kN \rceil$ some of its nodes N. This can be done in several ways, but the most common approach is to use a pooling ratio $k \in (0, 1]$. This means that for every N node in the original graph, there will be $\lceil kN \rceil$ nodes after applying the pooling layer. The choice of which nodes to drop is done based on a projection score against a learnable vector $\vec{p}$. The projection scores are also utilized as gating values to allow gradients to flow into $\vec{p}$, thus retained nodes with lower scores will have less feature retention.

The adjacency matrix that represents a graph is denoted by the A. The computation of pooled graph, $(\mathrm{F}', \mathrm{A}')$, from an input graph, (F, A) by GPL, is defined as:

$$\vec{y} = \frac{\mathrm{F}\vec{p}}{\|\vec{p}\|} \quad \vec{i} = \mathrm{top}-k(\vec{y}, k) \quad \mathrm{F}' = (\mathrm{F} \odot \tanh \vec{y})_{\vec{i}} \quad \mathrm{A}' = \mathrm{A}_{\vec{i},\vec{i}}$$

The $L2$ norm is a measure of the "length" or "size" of a vector and is represented by $\| \cdot \|$. It can be computed by taking the sum of the squares of all the elements in a vector. $\mathrm{top}-k$ picking is an algorithm that selects k elements from a given input vector, based on some criterion. The $\mathrm{top}-k$ indices are determined by broadcasting elementwise multiplication $\odot$ followed by indexing $\vec{i}$. This operation allows nodes at specific indices to be selected from an input vector.

GRL

A graph readout layer is added to flatten the characteristics of all nodes in order to provide a fixed-length representation for the complete graph. We use global average pooling and global max pooling to strengthen our representation, like in traditional convolutional neural networks (CNNs). After each GPL, we do both pooling operations, and then we aggregate all of the results from each operation at each layer.

5.3.1.4 Feature Combinators

AdaBL+GL has two parallel layers, so we use feature combinators to combine the vectors outputs of each layer. We gather syntactic and semantic interactions from source code and

bug reports and features from DFG through graph learning, which we then integrate using DNN. The combined vector is then used to rank potential buggy files

5.3.1.5 Hyperparameter Specifications

We have developed a deep learning model for bug report-source code interaction called AdaBL+GL. This model was built using the PyTorch [44] and PyTorch Geometric [17] frameworks, both open-source deep learning libraries. Our model performs best when using a two-layer LSTM with a hidden dimension of 64 and a drop rate of 5%. Additionally, we used GNN (graph neural network) for graph learning on DFG (Data Flow Graph). We combined the graph representation with the features extracted from bug reports and source code using a DNN. The ADAM optimizer with a batch size of 64 and a learning rate of 0.005 proved to be most effective for minimizing training loss. Finally, we experimented with different node sizes for the DFG (512, 768, 1024) and found that 75% of all source code files had 815 nodes in size while 90% had 1420 nodes. Considering these findings and GPU limitations and training time requirements, we have fixed 1024 nodes for the DFG, which gives the best performance for our deep learning model.

5.3.1.6 Model Training

The AdaBL+GL architecture is shown in figure 5.4. AdaBL+GL approach follows the same adaptive bug localization workflow as shown in the figure 5.1. Only the LSTM-based syntactic features extraction and GNN models are trained once using the label-rich projects. The inductive nature of the GNN model makes it generalizable to unseen DFG structures from another project of the same language. AdaBL+GL has been implemented as a classification task.

This approach has several advantages over traditional approaches. First, only the word embeddings models need to train in an unsupervised manner. Second, since both LSTM and GNN models are trained jointly, they learn to work together, leading to improved accuracy. Third, this approach is modular and can easily be applied to a new project by training the word vector model in an unsupervised manner.

In this chapter, we discussed AdaBL and AdaBL+GL in detail regarding the architecture, training methodology, and input features. In the next chapter, we will discuss the results of our experiments in three different dataset settings and analyze the performance of our models when compared with the benchmark baselines

Chapter 6

Results

In this section, we will discuss the results of using AdaBL and AdaBL+GL in conjunction with various bug localization benchmark models in three different project settings: within project, partial cross-project, and cross-project. The results include Top-K, MRR, and MAP metrics scores obtained during model evaluation. We used the Wilcoxon signed-rank and Cliff's delta, in addition to the three metrics, to quantify the amount of difference between models.

6.1 R1. How do different dataset settings impact the performance of the bug localization model?

The first column in table 6.1 and 6.2 training refers to projects that we used for training the bug localization model. The word mixed in partial cross-project and cross-project setting includes all other five projects except the project mentioned in the testing dataset column. The testing column refers to the project from which the Top-K, MRR, and MAP scores are obtained by applying the training model. For the comparison of AdaBL+GL with AdaBL and the three benchmark baselines models, both tables 6.1 and 6.2 list the p-value and Cliff's Delta.

6.1.1 Top K baselines

Table 4.1 shows the comparison of Top K across all the benchmark baseline models along with AdaBL and AdaBL+GL models in all three settings. The columns Top 1, Top 5, and Top 10 are obtained from evaluating all the models (AdaBL+GL, AdaBL, DNNLOC, Globug, rVSM) on the testing, which are trained on the training dataset.

We can observe Globug model performs better in all three settings in terms of Top@k for k=1, 5, 10. For example, Globug successfully locates 43.5% bugs in Eclipse and 40.7% in SWT in top-1 in the within-project settings. And on average, Globug locates 32.4% within the project setting, 27.9% in partial cross-project, and 26.5% in cross-project in terms of top-1. Similarly, Globug is significantly better in all models for top-1, top-5, and top-10 in all three settings (with p-Value < 0.05 and large effect size with $0.4 < \delta < 0.8$) except for DNNLOC at top-1 (with p-Value > 0.05 and with small effect size $\delta \leq 0.15$).

AdaBL+GL is observed to perform better than rVSM (with p-Value < 0.05 and with large effect size $\delta > 0.7$) and AdaBL (with p-Value < 0.05 and with small to medium effect size $0.2 < \delta \leq 0.55$) in all three settings at top-10. Both DNNLOC and Globug outperform AdaBL+GL in all three settings at top-1 and top-5. In partial cross-project and cross-project, both AdaBL+GL outperforms DNNLOC and rVSM in all six projects, with comparable performance in the top-10. For top-1, top-5, and top-10, AdaBL+GL demonstrates to perform similar to DNNLOC and rVSM (p-Values > 0.05 and at large effect size in all three settings.

6.1.2 MRR & MAP Baselines

We compare the performance of AdaBL+GL and AdaBL with DNNLOC, GLobug, and rVSM, in terms of MRR and MAP. Table 4.2 shows the results. We can observe that AdaBL+GL performs slightly better than AdaBL (with medium effect size $\delta = 0.38$), DNNLOC (with small effect size $\delta = 0.16$ at MAP), and rVSM (with large effect size $\delta > 0.4$ at both MRR and MAP) on all six datasets in within project setting.

Globug consistently outperforms all models in all three settings at MRR and MAP. Globug on average, has a higher 0.45 MRR and 0.42 MAP in all three settings. The improvement of Globug over DNNLOC is more substantial than the improvement over AdaBL+GL for MRR and MAP in within the project setting.

It is evident that baseline benchmark models outperformed AdaBL+GL and AdaBL in the partial cross-project and cross-project settings from the result. Overall, Globug

Table 6.1: Performance comparison (Top@k, k=1,5,10) of three state-of-the-art methods (DNNLOC, Globug, rVSM)

Dataset		**Top 1**					**Top 5**					**Top 10**				
Within Project																
Training	**Testing**	AdaBL+GL	AdaBL	DNNLOC	Globug	rVSM	AdaBL+GL	AdaBL	DNNLOC	Globug	rVSM	AdaBL+GL	AdaBL	DNNLOC	Globug	rVSM
AspectJ	AspectJ	0.231	0.193	0.325	0.3	0.091	0.509	0.452	0.507	0.582	0.227	0.741	0.663	0.571	0.745	0.355
Birt	Birt	0.078	0.054	0.182	0.252	0.136	0.262	0.198	0.326	0.53	0.287	0.409	0.369	0.465	0.668	0.367
Eclipse	Eclipse	0.3	0.27	0.336	0.273	0.109	0.576	0.493	0.558	0.525	0.235	0.771	0.687	0.679	0.699	0.307
JDT	JDT	0.437	0.381	0.293	0.435	0.171	0.748	0.674	0.406	0.733	0.335	0.802	0.77	0.557	0.848	0.421
SWT	SWT	0.137	0.119	0.294	0.278	0.174	0.416	0.321	0.584	0.607	0.34	0.646	0.515	0.687	0.731	0.481
Tomcat	Tomcat	0.121	0.109	0.323	0.407	0.308	0.242	0.198	0.605	0.692	0.429	0.571	0.397	0.798	0.791	0.538
mean		0.217	0.188	0.292	0.324	0.165	0.459	0.389	0.498	0.612	0.309	0.657	0.567	0.626	0.747	0.412
p-Value Globug		>0.05	<0.05	>0.05	-	<0.05	>0.05	<0.05	<0.05	-	<0.05	>0.05	<0.05	<0.05	-	<0.05
δ Globug		0.472	0.722	0	-	0.778	0.556	0.778	0.556	-	1	0.333	0.722	0.611	-	1
Improved% Globug		+39.56	+53.12	+10.39	+0.0	+65.03	+28.57	+44.56	+20.54	+0.0	+65.8	+12.82	+27.4	+17.63	+0.0	+57.81
p-Value AdaBL+GL		-	<0.05	>0.05	>0.05	>0.05	-	<0.05	>0.05	>0.05	>0.05	-	<0.05	>0.05	>0.05	<0.05
δ AdaBL+GL		-	0.222	0.444	0.472	0.167	-	0.278	0.167	0.556	0.5	-	0.389	0.194	0.333	0.833
improved% AdaBL+GL		+0.0	+14.32	-29.47	-39.56	+27.23	+0.0	+16.51	-8.15	-28.57	+39.06	+0.0	+14.71	+4.83	-12.82	+45.84
Partial Cross-Project																
20 % AspectJ+ 80% Mixed	AspectJ	0.125	0.106	0.138	0.261	0.16	0.352	0.314	0.302	0.578	0.31	0.587	0.493	0.396	0.701	0.433
20 % Birt+ 80% Mixed	Birt	0.097	0.068	0.122	0.23	0.122	0.328	0.241	0.264	0.53	0.266	0.584	0.473	0.352	0.678	0.349
20 % Eclipse+ 80% Mixed	Eclipse	0.082	0.055	0.267	0.214	0.18	0.316	0.284	0.399	0.494	0.335	0.578	0.493	0.471	0.651	0.418
20 % JDT+ 80% Mixed	JDT	0.231	0.217	0.175	0.362	0.161	0.533	0.485	0.349	0.684	0.322	0.746	0.692	0.427	0.809	0.411
20 % SWT+ 80%Mixed	SWT	0.066	0.032	0.321	0.205	0.163	0.286	0.215	0.507	0.483	0.326	0.537	0.484	0.592	0.657	0.448
20 % Tomcat+ 80% Mixed	Tomcat	0.074	0.045	0.332	0.401	0.327	0.315	0.296	0.53	0.728	0.507	0.536	0.485	0.619	0.825	0.599
mean		0.112	0.087	0.226	0.279	0.186	0.355	0.306	0.392	0.583	0.344	0.595	0.52	0.476	0.72	0.443
p-Value Globug		<0.05	<0.05	>0.05	-	<0.05	<0.05	<0.05	<0.05	-	<0.05	<0.05	<0.05	<0.05	-	<0.05
δ Globug		0.833	0.889	0.333	-	0.778	0.833	0.944	0.75	-	0.889	0.778	0.833	1	-	1
Improved% Globug		+85.42	+104.92	+20.99	+0.0	+40.0	+48.61	+62.32	+39.18	+0.0	+51.56	+19.01	+32.26	+40.8	+0.0	+47.64
p-Value AdaBL+GL		-	<0.05	>0.05	>0.05	>0.05	-	<0.05	>0.05	>0.05	>0.05	-	<0.05	>0.05	>0.05	<0.05
δ AdaBL+GL		-	0.444	0.778	0.833	0.667	-	0.611	0.111	0.833	0.111	-	0.722	0.444	0.778	0.722
improved% AdaBL+GL		+0.0	+25.13	-67.46	-85.42	-49.66	+0.0	+14.83	-9.91	-48.61	+3.15	+0.0	+13.45	+22.22	-19.01	+29.29
Cross-Project																
100% Mixed	AspectJ	0.128	0.113	0.129	0.211	0.17	0.39	0.285	0.312	0.552	0.312	0.629	0.573	0.388	0.732	0.426
100% Mixed	Birt	0.047	0.033	0.115	0.198	0.115	0.239	0.198	0.245	0.478	0.253	0.491	0.392	0.332	0.641	0.336
100% Mixed	Eclipse	0.049	0.029	0.246	0.256	0.196	0.243	0.195	0.381	0.527	0.348	0.498	0.356	0.455	0.703	0.429
100% Mixed	JDT	0.159	0.145	0.166	0.365	0.152	0.422	0.389	0.33	0.691	0.306	0.656	0.574	0.413	0.823	0.397
100% Mixed	SWT	0.058	0.036	0.281	0.244	0.161	0.26	0.22	0.465	0.556	0.324	0.495	0.414	0.559	0.726	0.441
100% Mixed	Tomcat	0.056	0.031	0.335	0.318	0.326	0.249	0.232	0.543	0.601	0.532	0.519	0.465	0.642	0.742	0.614
mean		0.083	0.064	0.212	0.265	0.187	0.3	0.253	0.379	0.568	0.346	0.548	0.462	0.465	0.728	0.44
p-Value Globug		<0.05	<0.05	>0.05	-	<0.05	<0.05	<0.05	<0.05	-	<0.05	<0.05	<0.05	<0.05	-	<0.05
δ Globug		1	1	0.333	-	0.722	1	1	0.889	-	0.889	0.944	1	0.944	-	1
Improved% Globug		+104.6	+122.19	+22.22	+0.0	+34.51	+61.75	+76.74	+39.92	+0.0	+48.58	+28.21	+44.71	+44.09	+0.0	+49.32
p-Value AdaBL+GL		-	<0.05	>0.05	>0.05	>0.05	-	<0.05	>0.05	>0.05	>0.05	-	<0.05	>0.05	>0.05	>0.05
δ AdaBL+GL		-	0.5	0.833	1	0.833	-	0.556	0.444	1	0.389	-	0.556	0.5	0.944	0.778
improved% AdaBL+GL		+0.0	+25.85	-87.46	-104.6	-77.04	+0.0	+17.0	-23.27	-61.75	-14.24	+0.0	+17.03	+16.39	-28.21	+21.86

Table 6.2: Performance comparison (MRR and MAP) with three state-of-the-art methods (DNNLOC, Globug, rVSM)

Dataset		**MRR**					**MAP**				
Training	**Testing**	AdaBL+GL	AdaBL	DNNLOC	Globug	rVSM	AdaBL+GL	AdaBL	DNNLOC	Globug	rVSM
Within Project											
AspectJ	AspectJ	0.443	0.421	0.339	0.459	0.211	0.406	0.372	0.258	0.419	0.209
Birt	Birt	0.217	0.194	0.237	0.401	0.236	0.186	0.158	0.203	0.385	0.219
Eclipse	Eclipse	0.469	0.447	0.427	0.451	0.232	0.435	0.413	0.405	0.422	0.212
JDT	JDT	0.602	0.592	0.381	0.615	0.282	0.584	0.547	0.334	0.593	0.262
SWT	SWT	0.319	0.281	0.413	0.441	0.294	0.289	0.257	0.369	0.411	0.285
Tomcat	Tomcat	0.251	0.235	0.456	0.558	0.407	0.228	0.192	0.396	0.502	0.365
mean		0.383	0.362	0.376	0.488	0.277	0.355	0.323	0.328	0.455	0.259
p-Value Globug		>0.05	<0.05	<0.05	-	<0.05	<0.05	<0.05	<0.05	-	<0.05
δ Globug		0.389	0.556	0.722	-	0.944	0.444	0.611	0.889	-	1
Improved% Globug		+24.11	+29.65	+25.93	+0.0	+55.16	+24.69	+33.93	+32.44	+0.0	+54.9
p-Value AdaBL+GL		-	<0.05	>0.05	>0.05	>0.05	-	<0.05	>0.05	>0.05	>0.05
δ AdaBL+GL		-	0.167	0.056	0.389	0.5	-	0.167	0.167	0.444	0.444
improved% AdaBL+GL		+0.0	+5.64	+1.84	-24.11	+32.12	+0.0	+9.44	+7.91	-24.69	+31.27
Partial Cross-Project											
20 % AspectJ+ 80% Mixed	AspectJ	0.299	0.275	0.247	0.419	0.26	0.259	0.244	0.223	0.391	0.227
20 % Birt+ 80% Mixed	Birt	0.251	0.224	0.218	0.395	0.219	0.238	0.207	0.199	0.342	0.203
20 % Eclipse+ 80% Mixed	Eclipse	0.257	0.235	0.39	0.365	0.319	0.212	0.208	0.367	0.327	0.296
20 % JDT+ 80% Mixed	JDT	0.421	0.394	0.282	0.547	0.27	0.402	0.366	0.258	0.505	0.251
20 % SWT+ 80% Mixed	SWT	0.247	0.217	0.427	0.359	0.281	0.188	0.151	0.395	0.334	0.272
20 % Tomcat+ 80% Mixed	Tomcat	0.256	0.224	0.457	0.558	0.444	0.194	0.181	0.422	0.514	0.413
mean		0.289	0.262	0.337	0.44	0.299	0.249	0.226	0.311	0.402	0.277
p-Value Globug		<0.05	<0.05	>0.05	-	<0.05	<0.05	<0.05	>0.05	-	<0.05
δ Globug		0.778	0.889	0.444	-	0.778	0.778	0.833	0.389	-	0.778
Improved% Globug		+41.43	+50.71	+26.51	+0.0	+38.16	+47.0	+56.05	+25.53	+0.0	+36.82
p-Value AdaBL+GL		-	<0.05	>0.05	>0.05	>0.05	-	<0.05	>0.05	>0.05	>0.05
δ AdaBL+GL		-	0.5	0.194	0.778	0.278	-	0.278	0.389	0.778	0.389
improved% AdaBL+GL		+0.0	+9.8	-15.34	-41.43	-3.4	+0.0	+9.68	-22.14	-47	-10.65
Cross-Project											
100% Mixed	AspectJ	0.338	0.288	0.238	0.397	0.263	0.291	0.268	0.217	0.374	0.23
100% Mixed	Birt	0.218	0.174	0.205	0.376	0.21	0.154	0.145	0.191	0.352	0.196
100% Mixed	Eclipse	0.193	0.162	0.377	0.465	0.339	0.15	0.133	0.349	0.405	0.31
100% Mixed	JDT	0.353	0.322	0.273	0.557	0.261	0.314	0.287	0.249	0.537	0.241
100% Mixed	SWT	0.211	0.186	0.39	0.415	0.278	0.173	0.147	0.365	0.418	0.27
100% Mixed	Tomcat	0.218	0.175	0.462	0.462	0.451	0.169	0.139	0.437	0.449	0.423
mean		0.255	0.218	0.324	0.445	0.3	0.209	0.186	0.301	0.423	0.278
p-Value Globug		<0.05	<0.05	<0.05	-	<0.05	<0.05	<0.05	<0.05	-	<0.05
δ Globug		1	1	0.694	-	0.833	1	1	0.722	-	0.778
Improved% Globug		+54.29	+68.48	+31.47	+0.0	+38.93	+67.72	+77.83	+33.7	+0.0	+41.37
p-Value AdaBL+GL		-	<0.05	>0.05	>0.05	>0.05	-	<0.05	>0.05	>0.05	>0.05
δ AdaBL+GL		-	0.556	0.5	1	0.333	-	0.556	0.667	1	0.5
improved% AdaBL+GL		+0.0	+15.64	-23.83	-54.29	-16.22	+0.0	+11.65	-36.08	-67.72	-28.34

outperformed both AdaBL+GL and AdaBL and benchmark models DNNLOC and rVSM in all three settings at MRR and MAP.

6.2 R2. Does the performance of ML/DL-based localization for a project vary depending on the dataset setting?

Many different factors can influence the performance of a bug localization model for a project. The input (word2vec, hand-engineered features), the training strategy used, and the dataset setting can all impact how well a model performs. The dataset setting can also affect how well a bug localization model performs. Suppose two models are trained on different dataset compositions using different input formats and training strategies but then applied to similar projects. In that case, they may not perform equally well because each model has learned unique feathers from the training corpus. There is a lack of research on how dataset settings influence bug localization models in the software engineering research community. Both tables 4.1 and 4.2 shows the performance of the bug localization model for each project under different dataset settings.

We observed that all models performed significantly better in terms of top-1, top-5, top-10, MRR, and MAP within the project setting. Particularly Globug has a better score at top-1(with p-Value < 0.05 for both partial cross-project and cross-project and with a large effect size where $\delta = 0.5$) in all six projects in the within project setting. Even Globug has 0.48 and 0.45 at MRR and MAP, respectively, in the within project setting, which is better than MRR and MAP in partial cross-project (with p-Value < 0.05 and medium effect size) and cross-project setting (with p-Value < 0.05 and small effect size). Birt is the only project that benefited from the partial cross-project setting in AdaBL+GL, AdaBL, and Globug models in terms of top-5, top-10, MRR, and MAP. Strangely rVSM is the only model that performed better under the cross-project setting for three projects (AspectJ, Eclipse, and Tomcat) in terms of top-5, MRR, and MAP. One reason for the rVSM performance in the cross-project setting is improved textual similarity learned from the mixed dataset.

In the next chapter, we will analyze why Globug performs consistently better in parietal cross-project and cross-project settings.

Chapter 7

Result Analysis

7.1 Motivation

Deep learning models are increasingly being used in NLP, Software Engineering. However, these models are often opaque and difficult for humans to understand. This lack of explainability can lead to unintended consequences of incorrect predictions. From the previous chapter, it is evident that Globug outperformed AdaBL+GL. In order to analyze the reason behind the prediction of both Globug and AdaBL+GL, we plan to analyze the bug report and code pair based on four prediction combinations from Globug and AdaBL+GL at the Top 5 ranked buggy source code files.

- Match - where both Globug and AdaBL+GL have predicted relevant buggy source code files for the given bug report
- Mismatch - where both models have retrieved irrelevant source code files for the given bug report
- Match Mismatch - where Globug has predicted the retrieved source code files as buggy, and the AdaBL+GL has predicted irrelevant source code files.
- Mismatch Match - the predictions are vice versa to Match Mismatch

Table 7.1 shows the percentage of the relevant files predicted as buggy source code at each project in the Top 5 and also shows the percentage of four prediction combinations.

Table 7.1: Comparison between Globug & AdaBL+GL predictions at Top 5

Project Setting	**Dataset**	**Top 5**		**Globug & AdaBL+GL**			
		Globug	AdaBL+GL	Match	Mismatch	Match & Mismatch	Mismatch & Match
Within Project	AspectJ	58.33	51.17	28.7	23.15	29.63	18.52
	Birt	52.78	26.94	11.55	37.11	41.24	10.1
	Eclipse	52.56	58.51	30.23	22.56	22.33	24.88
	JDT	73.67	75.22	58.21	12.08	15.46	14.25
	SWT	59.74	42.12	22.82	30.06	36.92	10.2
	Tomcat	70.33	25.03	14.29	25.27	56.04	4.4
	Average	**61.24**	**46.5**	**27.63**	**25.04**	**33.6**	**13.73**
Partial Cross Project	AspectJ	59.09	35.62	13.64	29.55	45.45	11.36
	Birt	52.67	33.51	16.39	38.4	36.28	8.94
	Eclipse	49.71	31.84	12.79	41.16	36.92	9.13
	JDT	66.97	53.53	34.18	21.29	32.78	11.74
	SWT	47.33	29.11	9.65	44.32	37.67	8.35
	Tomcat	69.91	31.74	14.9	25.79	55.01	4.3
	Average	**57.61**	**35.89**	**16.93**	**33.42**	**40.69**	**8.97**
Cross Project	AspectJ	55.27	39.13	19.17	31.63	36.1	13.1
	Birt	47.03	23.99	7.49	46.91	39.54	6.06
	Eclipse	52.57	24.81	7.57	42	45	5.43
	JDT	68.81	43.05	26.82	23.66	41.99	7.53
	SWT	54.16	26.15	9.75	39.53	44.41	6.31
	Tomcat	56.65	25.02	10.52	38.63	46.14	4.72
	Average	**55.75**	**30.36**	**13.55**	**37.06**	**42.2**	**7.19**

From the table 7.1 we can observe that for JDT, both Globug and AdaBL+GL have predicted relevant source code files for more than 70% of bug reports, with 58% of predictions agreeing with each other under within project setting. From our manual inspection of JDT bug reports and source code pairs, we observe that most of the bug reports have the source code information in terms of source snippets, stack trace, or program-related keywords. We assume that due to the popularity of the JDT and the end-users being developers, the bug reports have more source code-related information. And the mismatch prediction in both models is due to the lack of sufficient information. Most of the bug report contains platform-specific errors and Java programming language-related bugs. For example, bug 264606 is about the "warning to be displayed to the Java variables without 'this' keyword." And we can also observe that both Globug and AdaBL+GL performance degrades for partial cross-project and cross-project settings. And we assume that this is related to a lack of project-specific features, as project-specific information is limited in both settings.

From the 7.1, we can observe that AdaBL+GL has consistently performed for Tomcat at all three settings when compared to Globug. From our manual inspection of the bug report and the source code pairs, we can conclude that Globug has performed well due to the availability of abundant natural information and less source code relation information (code snippets, stack information .etc). We suspect Globug has higher performance because of its capability to learn the semantic similarity between bug reports and the source code. And AdaBL+GL failed to capture the relevant source code files because AdaBL+GL encodes the cosine similarity of each token of bug report and the source code, and we assume by doing so, most of the semantic features are lost.

Overall the, for the match, both Globug and AdaBL+GL predict the relevant file when source-related information such as source code snippets, stack trace, .etc available in the bug reports. Particularly AdaBL+GL predicts the source code files even with subwords, and Globug fails to rank the source code files. For example, in the bug report 399408 of AspectJ, the description has the source-related keyword StackMapAdder and AdaBL+GL managed to rank the relevant source code file based on the subword "stackMap" presence. The mismatch predictions in both models often occur for the bug report with bugs that can only be solved using domain knowledge without lacking sufficient information (description or stack trace) in the bug report. For example, in the bug report 264606 of JDT, the bug is because of Java compilation. We believe the models need to capture the domain knowledge to localize the source code associated with such bugs. Both Globug and AdaBL+GL are not capable of capturing the domain knowledge.

We observe that the AdaBL+GL capability to source code files based on the subword match and the false syntactic pattern match is the reason for most mismatch predictions. For example, bug report 260751 from the AspectJ contains the stack trace information,

but AdaBL+GL failed to predict the relevant files. We believe the subword-based features, with more feature weightage, lead to incorrect predictions on further manual inspection. In the case of bug report 260751, the subword "stringIndex" from the description phrase "java.lang.StringIndexOutOfBoundsException" leads to an incorrect prediction.

From the table 7.1 we can observe that performance of the AdaBL+GL is decreasing as the domain-specific data points split decreases, i.e., AdaBL+GL performs well on the within project followed by partial cross-project and finally cross-project. But Globug, in all three settings, the prediction is consistently better, and we believe this is due to the input format, which is the entire vector. Based on these observations, we conclude that the input to the model, which is a similarity matrix, is the cause of the AdaBL+GL's poor performance.

Chapter 8

Threats to Validity

8.1 Internal Validity

Author bias in the experiments is a threat to internal validity. We've gone through our code for bugs and patched those we found. We use the dataset that Ye et al. proposed in their research [67], and the dataset was previously used in multiple bug localization research [71, 64, 63, 65] for benchmarking and evaluation purpose. The data has been collected and compiled from open-source projects by software engineering researchers, so the data is valid.

We use the Wilcoxon Signed Rank test [58] and Cliff's Delta (δ) [11] effect size in each comparison to ensure results are statistically valid. This means that we can be confident in the conclusions drawn from our study. The Wilcoxon Signed Rank Test is a non-parametric test used to compare two related samples. The Cliff's Delta effect size measures the difference between the two groups.

The effectiveness of ML/DL models is determined by several factors, including architecture and training features. The proposed AdaBL+GL model has an LSTM layer, uses fastText for embedding learning, and employs GNN to learn from DFG. Modifying the model architecture, such as experimenting with state-of-the-art transformers-based embeddings, and utilizing other graphs structures for source code representation, can help increase the model result.

8.2 Construct Validity

We leveraged existing benchmarks, toolsets, and libraries to prevent implementation biases for construct validity. We used the replication package made public by the Globug authors, and we used the implementations for DNNLOC and rVSM that almost match the results. Also, to ensure that our measures are valid, we employed top-k, MAP, and MRR benchmark metrics which are previously used in bug localization research [24, 40, 10].

8.3 External Validity

To address the external validity for evaluation, we use the dataset that Ye et al. proposed in their research [67]. The data is based on bug reports from bug tracking systems from actual projects (e.g., Tomcat-Java), so it's accurate. However, it would be beneficial to be able to apply the findings to a real-world case study to see if they are limited to open-source systems or may be applied to other situations.

Chapter 9

Conclusion

In our thesis, we have constructed a repository of DL-based bug localization research publications with 31 papers. We have classified bug localization paper based on the dataset setting (within project, partial cross-project, and cross-project) category. We have grouped the research based on DL-based source code representation for each category and have discussed each research strength and weakness. There is a growing need for reproducible replication packages in research, particularly in bug localization. So We have collected and compiled the publicly available replication package list for each paper. We have found that the replication package availability for DL-based bug localization is few. We have discussed the need for a reproducible replication package, particularly for bug localization research.

Also inspired by Adaptive Deep Code Search (AdaCS) [36] we have developed Adaptive Bug Localization (AdaBL) for cross-project bug localization. We have demonstrated that a data flow graph (DFG) can be employed to represent the source code as a graph, and GNN can be applied to DFG to learn the structural information in the source code. We also showed that GNN could be generalized to unknown graph structures due to its inductive nature, which is essential for cross-project bug localization.

Additionally, we have discussed the need for a pure cross-project bug localization technique in detail. We have evaluated our proposed models and the benchmark-based baselines in three different dataset settings for comparison purposes. Our experiments show the traditional model will not perform well in cross-project settings. Our experiment in different dataset settings helped us identify a hybrid model with a new training strategy that can consistently perform better in all three settings.

From our survey for replication packages in DL-based bug localization, we have emphasized the need for creating reproducible replication packages and the availability of those

packages to fellow researchers. We hope that more researchers will adopt this practice in their own work and help make their research available to everyone.

We have also presented new insights and research directions with the goal of making cross-project bug localization more efficient and practical. We hope that this would encourage more researchers to build DL-based bug localization by leveraging the source code's structural information and looking into alternate methods for pure cross-project bug localization.

9.1 Further Work

- As previously mentioned, we suspect features extracted using the source code and bug reports interaction matrix are not sufficient for the DL model to capture transferable features.
  - We are interested in applying state-of-the-art deep learning architecture such as transformers [52] instead of LSTM to learn the syntactic patterns and evaluate the transfer
  - We also want to test cross-project by replacing the interaction matrix with state-of-the-art models like BERT [14] and testing transfer learning capabilities.
- We utilized fastText for node features in the GNN. We wish to look into further ways to expand the node features in the future, enriching the node features for improving the AdaBL+GL model capability to capture the inter-related relationships between the source code tokens.
- We trained and test our only on the dataset by Ye et al. [67] which contains only the Java projects. We are interested in applying our models to projects from other languages. To explore the transfer learning capabilities of bug localization between cross-project and cross-language.

References

[1] https://figshare.com/articles/dataset/The_dataset_of_six_open_source_Java_projects/951967/10, Aug 2014.

[2] tree-sitter, Apr 2022.

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv:1711.00740 [cs]*, May 2018.

[4] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, Dec 2017.

[5] Marcel Böhme, Ezekiel Olamide Soremekun, Sudipta Chattopadhyay, Emamurho Juliet Ugherughe, and Andreas Zeller. How developers debug software — the dbgbench dataset. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, page 244–246, May 2017.

[6] Cătălina Cangea, Petar Veličković, Nikola Jovanović, Thomas Kipf, and Pietro Liò. Towards sparse hierarchical graph classifiers. *arXiv:1811.01287 [cs, stat]*, Nov 2018.

[7] Kaibo Cao, Chunyang Chen, Sebastian Baltes, Christoph Treude, and Xiang Chen. Automated query reformulation for efficient search based on query logs from stack overflow. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, page 1273–1285, May 2021.

[8] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, page 396–407, Paderborn Germany, Aug 2017. ACM.

[9] Ke Chen, Yufei Li, Yingfeng Chen, Changjie Fan, Zhipeng Hu, and Wei Yang. Glib: towards automated test oracle for graphically-rich applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 1093–1104, Athens Greece, Aug 2021. ACM.

[10] Shasha Cheng, Xuefeng Yan, and Arif Ali Khan. A similarity integration method based information retrieval and word embedding in bug localization. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, page 180–187, Dec 2020.

[11] Norman Cliff. *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[12] Nick Craswell. Mean reciprocal rank.

[13] Hoa Khanh Dam, Truyen Tran, John Grundy, and Aditya Ghose. Deepsoft: a vision for a deep model of software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 944–947, Seattle WA USA, Nov 2016. ACM.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805 [cs]*, May 2019.

[15] Emre Dogan. Bug localization by using bug reports. https://github.com/emredogan7/bug-localization-by-dnn-and-rvsm, Mar 2022.

[16] Emre Doğan and Hamdi Alperen Çetin. A survey on machine learning approaches for software bug localization. page 3.

[17] Matthias Fey and Jan Eric Lenssen. *Fast Graph Representation Learning with PyTorch Geometric*. May 2019.

[18] git disl. Md-cnn. https://github.com/git-disl/MD-CNN, Feb 2022.

[19] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[20] 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*, page 631–642, Seattle WA USA, Nov 2016. ACM.

[21] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. *arXiv:2009.08366 [cs]*, Sep 2021.

[22] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *2013 35th International Conference on Software Engineering (ICSE)*, page 392–401, May 2013.

[23] Xuan Huo, Ming Li, and Zhi-Hua Zhou. Learning unified features from natural and programming languages for locating buggy source code. page 7, 2016.

[24] Xuan Huo, Ferdian Thung, Ming Li, David Lo, and Shu-Ting Shi. Deep transfer bug localization. *IEEE Transactions on Software Engineering*, 47(7):1368–1380, Jul 2021.

[25] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, page 437–440. ACM Press, 2014.

[26] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *29th International Conference on Software Engineering (ICSE'07)*, page 489–498, May 2007.

[27] Youngkyoung Kim, Misoo Kim, and Eunseok Lee. Feature combination to alleviate hubness problem of source code representation for bug localization. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, page 511–512, Dec 2020.

[28] Donald Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1986.

[29] 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 *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 476–481, Nov 2015.

[30] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, page 218–229, May 2017.

[31] Leslie Lamport. *LaTeX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[32] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. *arXiv:1405.4053 [cs]*, May 2014.

[33] Jaekwon Lee, Dongsun Kim, Tegawendé F. Bissyandé, Woosung Jung, and Yves Le Traon. Bench4bl: reproducibility study on the performance of ir-based bug localization. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 61–72. ACM, Jul 2018.

[34] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, Oct 2019.

[35] Hongliang Liang, Lu Sun, Meilin Wang, and Yuxing Yang. Deep learning with customized abstract syntax tree for bug localization. *IEEE Access*, 7:116309–116320, 2019.

[36] Chunyang Ling, Zeqi Lin, Yanzhen Zou, and Bing Xie. Adaptive deep code search. In *Proceedings of the 28th International Conference on Program Comprehension*, page 48–59. ACM, Jul 2020.

[37] Guangliang Liu, Yang Lu, Ke Shi, Jingfei Chang, and Xing Wei. Convolutional neural networks-based locating relevant buggy code files for bug reports affected by data imbalance. *IEEE Access*, 7:131304–131316, 2019.

[38] Guangliang Liu, Yang Lu, Ke Shi, Jingfei Chang, and Xing Wei. Mapping bug reports to relevant source code files based on the vector space model and word embedding. *IEEE Access*, 7:78870–78881, 2019.

[39] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. Owl eyes: spotting ui display issues via visual understanding. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, page 398–409, Virtual Event Australia, Dec 2020. ACM.

[40] Nima Miryeganeh, Sepehr Hashtroudi, and Hadi Hemmati. Globug: Using global data in fault localization. *Journal of Systems and Software*, 177:110961, Jul 2021.

[41] S. Nima Miryeganeh. Globug. https://github.com/miryeganeh/GloBug, May 2021.

[42] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. *arXiv:1810.02244 [cs, stat]*, Nov 2021.

[43] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. page 7.

[44] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[45] Dhanashree P Pathak and Srinu Dharavath. A survey paper for bug localization. 3(11):4, 2012.

[46] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, and G. Antoniol. Combining probabilistic ranking and latent semantic indexing for feature identification. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, page 137–148, Jun 2006.

[47] Francisca Pérez, Raúl Lapeña, Jaime Font, and Carlos Cetina. Fragment retrieval on models for model maintenance: Applying a multi-objective perspective to an industrial case study. *Information and Software Technology*, 103:188–201, Nov 2018.

[48] Francisca Pérez, Raúl Lapeña, Jaime Font, and Carlos Cetina. Fragment retrieval on models for model maintenance: Applying a multi-objective perspective to an industrial case study. *Information and Software Technology*, 103:188–201, Nov 2018.

[49] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 345–355, Nov 2013.

[50] Hao-Jun Michael Shi, Dheevatsa Mudigere, Maxim Naumov, and Jiyan Yang. Compositional embeddings using complementary partitions for memory-efficient recommendation systems. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, page 165–175. ACM, Aug 2020.

[51] Yuhui Su, Zhe Liu, Chunyang Chen, Junjie Wang, and Qing Wang. Owleyes-online: a fully automated platform for detecting and localizing ui display issues. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 1500–1504, Athens Greece, Aug 2021. ACM.

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv:1706.03762 [cs]*, Dec 2017.

[53] Bei Wang, Ling Xu, Meng Yan, Chao Liu, and Ling Liu. Multi-dimension convolutional neural network for bug localization. *IEEE Transactions on Services Computing*, page 1–1, 2020.

[54] 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 - ICPC 2014*, page 53–63. ACM Press, 2014.

[55] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, page 297–308, Austin Texas, May 2016. ACM.

[56] Zirui Wang, Zihang Dai, Barnabás Póczos, and Jaime Carbonell. Characterizing and avoiding negative transfer. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, page 11285–11294, Jun 2019.

[57] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. Locus: locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, page 262–273. ACM, Aug 2016.

[58] Frank Wilcoxon. Individual comparisons by ranking methods. In *Breakthroughs in statistics*, pages 196–202. Springer, 1992.

[59] 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*, page 181–190, Sep 2014.

[60] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, Aug 2016.

[61] Wensheng Xia, Ying Li, Tong Jia, and Zhonghai Wu. Bugidentifier: An approach to identifying bugs via log mining for accelerating bug reporting stage. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, page 167–175, Jul 2019.

[62] 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)*, page 703–704, Dec 2018.

[63] 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, Jul 2018.

[64] 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, Jan 2019.

[65] Yan Xiao, Jacky Keung, Qing Mi, and Kwabena E. Bennin. Improving bug localization with an enhanced convolutional neural network. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, page 338–347, Dec 2017.

[66] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. *arXiv:1806.03536 [cs, stat]*, Jun 2018.

[67] 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*, page 689–699. ACM, Nov 2014.

[68] 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*, page 404–415. ACM, May 2016.

[69] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *arXiv:1806.08804 [cs, stat]*, Feb 2019.

[70] 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, Feb 2017.

[71] Wei Yuan, Binhang Qi, Hailong Sun, and Xudong Liu. Dependloc: A dependency-based framework for bug localization. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, page 61–70, Dec 2020.

[72] Tao Zhang, He Jiang, Xiapu Luo, and Alvin T.S. Chan. A literature review of research in bug resolution: Tasks, challenges and future directions. *The Computer Journal*, 59(5):741–773, May 2016.

[73] 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*, 110:121–135, Jun 2019.

[74] Hao Zhong and Hong Mei. Mining repair model for exception-related bug. *Journal of Systems and Software*, 141:16–31, Jul 2018.

[75] 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 *2012 34th International Conference on Software Engineering (ICSE)*, page 14–24, Jun 2012.

[76] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - ESEC/FSE '09*, page 91. ACM Press, 2009.

APPENDIX

Our Tools, Artifacts, Results

Links to our code, tools & and results are as follows:

- GitHub link to AdaBL & AdaBL+GL
- Globug modified scripts integrated with six benchmark dataset
- rVSM and DNNLOC integrated with six benchmark dataset