

Optimizing Automated Bug Localization for Practical Use

by

Partha Chakraborty

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Partha Chakraborty 2024

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

- External Examiner: Dr. Arie van Deursen
Professor
Department of Software Technology
Delft University of Technology
- Supervisor(s): Dr. Meiyappan Nagappan
Associate Professor
David R. Cheriton School of Computer Science
University of Waterloo
- Internal Member: Dr. Jimmy Lin
Professor
David R. Cheriton School of Computer Science
University of Waterloo
- Dr. Shane McIntosh
Associate Professor
David R. Cheriton School of Computer Science
University of Waterloo
- Internal-External Member: Dr. Weiyi Shang
Associate Professor
Department of Electrical and Computer Engineering
University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the 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.

Statement of Contribution

Earlier versions of the work in this thesis have been published as follows:

- Bug Localization Tools in Practice: User Perspectives and Expectations. (Chapter 4) **Partha Chakraborty**, Meiyappan Nagappan. Under review in a Software Engineering Conference.
- Aligning Programming Language and Natural Language: Exploring Design Choices in Multi-Modal Transformer-Based Embedding for Bug Localization. (Chapter 5) **Partha Chakraborty**, Venkatraman Arumugam, and Meiyappan Nagappan. In Proceedings of the Third ACM/IEEE International Workshop on NL-based Software Engineering (NLBSE '24). Association for Computing Machinery, New York, NY, USA, 1–8.
- RLocator: Reinforcement Learning for Bug Localization. (Chapter 6) **Partha Chakraborty**, Mahmoud Alfadel, and Meiyappan Nagappan. In IEEE Transactions on Software Engineering, volume 50, no 8, pages 2163-2177, August 2024.
- BLAZE: Cross-Language and Cross-Project Bug Localization via Dynamic Chunking and Hard Example Learning. (Chapter 7) **Partha Chakraborty**, Mahmoud Alfadel, and Meiyappan Nagappan. Under review in IEEE Transactions on Software Engineering (TSE), 13 pages.

The following publication, while not directly related to the material in this thesis, was produced concurrently with the research conducted for this thesis.

- Revisiting the Performance of Deep Learning-Based Vulnerability Detection on Realistic Datasets. **Partha Chakraborty**, Krishna Kanth Arumugam, Mahmoud Alfadel, Meiyappan Nagappan, and Shane McIntosh. In IEEE Transactions on Software Engineering, volume 50, no 8, pages 2163-2177, August 2024.

Abstract

A considerable share of resources and developers’ efforts is focused on addressing software bugs. Identifying the root causes of these bugs within the codebase is crucial for their resolution. Automated tools for bug localization aim to assist in this process. However, their effectiveness is often limited, leading to low adoption rates. This low adoption rate indicates the disparity between research goals and developers’ expectations, emphasizing the need for improvements in bug localization tools.

This thesis explores and addresses the challenges faced by developers and tool-builders in implementing practical bug localization tools. Our research focuses on understanding developers’ expectations and enhancing the tools’ overall effectiveness.

Initially, we conduct a mixed-method empirical study to understand developers’ expectations. The study reveals that while developers are willing to use bug localization tools, they have concerns related to accuracy and potential leakage of intellectual property. We found that only 27.5% of developers are familiar with these tools. The study indicates that developers need more reliable performance, better integration, flexibility, transparency, and contextual understanding to increase adoption and effectiveness.

We also examine performance issues in bug localization tools, particularly with their base—the embedding model. We found that key factors such as pre-training strategies, data familiarity, and input sequence length in embedding techniques significantly affect performance. Our findings show that using project-specific data and pre-training methods like ELECTRA can improve model performance by 25.9%.

Additionally, we explore the use of reinforcement learning (RL) in bug localization and propose an RL agent called RLocator. RLocator learns from developer feedback, making it suitable for low-data environments. We also propose BLAZE, an efficient bug localization technique for cross-project and cross-language settings. By using dynamic chunking, a technique that dynamically adjusts the size of the input data to the model, and hard example learning, BLAZE achieves up to a 144% improvement in Mean Average Precision (MAP) compared to previous tools.

In conclusion, our findings highlight the shortcomings in the adaptability and efficiency of current tools. We advocate for highly adaptable cross-language, cross-project bug localizers to enhance adoption rates among developers. By leveraging our observations, curated datasets, and proposed methods, tool builders can create more user-friendly bug localization tools for software developers, inspiring a new wave of innovation in this field.

Acknowledgements

As I reflect on this dissertation, I am reminded of the many people who have played an important role in my life and academic journey. This work would not have been possible without the constant support, advice, encouragement, and sacrifices of my family, friends, mentors, colleagues, and collaborators. To all of you, I owe my deepest gratitude.

First and foremost, I would like to extend my sincerest thanks to my supervisor, Dr. Meiyappan Nagappan. Your unwavering belief in my abilities, coupled with your constant guidance and support, has shaped not just this dissertation but also me as a researcher. I have learned invaluable lessons under your supervision, and I am deeply grateful for all the time and energy you invested in helping me succeed on this journey.

I also want to extend my heartfelt thanks to my committee members: Dr. Arie van Deursen, who kindly accepted the invitation to join the committee, and Dr. Jimmy Lin, Dr. Shane McIntosh, and Dr. Weiyi Shang, who took the time to review this thesis and provide invaluable feedback. Your support has been instrumental in shaping this work.

I have been incredibly fortunate to have Professor Mahmoud Alfadel as both my mentor and collaborator throughout my Ph.D. at the University of Waterloo. Your insightful guidance, quick responses, and thoughtful feedback, always with my best interests at heart, have been a constant source of support. Thank you for making this journey a little smoother and for always being there when I needed direction.

A special thank you goes out to my SWAG Lab family at the University of Waterloo. Over the past few years, I have shared so many enriching conversations and experiences with you. Some of the names that stand out are Akinbowale Akin-Taylor, Gengyi Sun, Mahtab Nejati, Nimmi Rashinika Weeraddana, Krishna Kanth Arumugam, Venkatraman Arumugam, and Hossein Keshavarz. I am also immensely grateful to my office roommates—Farshad Kazemi, Noble Saji Mathews, and Aniruddhan Murali—our time together at conferences and in the office are memories I will always cherish. If I have missed anyone's name, please know it is not due to a lack of appreciation. This journey has introduced me to so many amazing people, and each of you has contributed to making this experience both fulfilling and enjoyable.

Finally, I owe my deepest gratitude to my family—my parents, my sister, my brother-in-law, and my nephew. Your unwavering support, love, and encouragement have been my anchor throughout this journey. Without you, this accomplishment would not have been possible. From the bottom of my heart, thank you for being by my side, for your patience, and for always believing in me.

This dissertation is a reflection of the collective efforts and support of all these wonderful people, and I am forever grateful for each and every one of you.

Dedication

To my beloved mother, whose sacrifices and resilience have made it possible for me to reach this point.

Table of Contents

Examining Committee Membership	ii
Author’s Declaration	iii
Statement of Contribution	iv
Abstract	v
Acknowledgements	vi
Dedication	vii
List of Figures	xiv
List of Tables	xvi
List of Abbreviations	xviii
1 Introduction	1
1.1 Problem Statement	2
1.2 Thesis Overview	2
1.2.1 Understanding Developers’ Expectation from Bug Localization Tools	3
1.2.2 Exploring Design Choices for Building Bug Localization Tools	4
1.2.3 Addressing Challenges in the Use of Bug Localization Tools	5

1.3	Thesis Contribution	6
1.4	Thesis Organization	7
2	Background and Definitions	8
2.1	Bug Localization Terms	8
2.1.1	Bug report	8
2.1.2	Pull Request	9
2.1.3	Bug Localization	9
2.1.4	Types of Bug Localization	10
2.1.5	Document Vectorizer	10
2.2	Overview of Bug Localization Process	11
2.3	Chapter Summary	13
3	Related Work	14
3.1	Data and Embedding	14
3.1.1	Dataset	14
3.1.2	Representation Learning	16
3.1.3	Embedding Granularity	18
3.2	Learning and Models	20
3.2.1	Types of models	20
3.3	Experiments and Applications	21
3.3.1	Experimental Setting	21
3.3.2	Evaluation Metrics	23
3.3.3	Bug Localization System in Industry	24
3.3.4	Complete tool	24
3.3.5	Chapter Summary	25

4	Bug Localization Tools in Practice: Developer Perspectives and Expectations	26
4.1	Introduction	26
4.2	Methodology	28
4.2.1	Interview guide	29
4.2.2	Open Ended Interview	30
4.2.3	Qualitative Data Analysis	31
4.2.4	Developers' Survey	31
4.3	Demographics	34
4.3.1	Interviewee Demographics	34
4.3.2	Survey Demographics	34
4.4	Results	34
4.4.1	RQ1: What are developers' current approaches and perspectives on bug localization tools?	35
4.4.2	RQ2: What are developers' expectations from automated bug localization tools?	40
4.4.3	RQ3: Who are the primary users of bug localization tools, and what are the tools' focus areas?	41
4.4.4	RQ4: What features do developers value in automated bug localization tools?	42
4.5	Discussion	44
4.5.1	Timeframe of results	44
4.5.2	Expected results in TopK	45
4.5.3	Granularity of suggestion	45
4.5.4	Willingness to adopt	45
4.6	Implication	46
4.6.1	What Needs Our Focus Now?	46
4.6.2	What Needs to Change in the Future?	47
4.7	Threats to Validity	48
4.8	Conclusion	49
4.9	Chapter Summary	49

5	Exploring Design Choices in Multi-Modal Transformer-Based Embedding for Bug Localization	50
5.1	Introduction	50
5.2	Dataset	53
5.3	Methodology	54
5.4	Results	57
5.4.1	RQ1. Do we need data familiarity to apply the embeddings?	59
5.4.2	RQ2. Do pre-training methodologies impact embedding models' performance?	60
5.4.3	RQ3. Do transformers with longer maximum input sequence lengths perform well compared to the models with shorter maximum input sequences?	63
5.5	Threats to Validity	64
5.6	Conclusion	65
5.7	Chapter Summary	65
6	RLocator: Reinforcement Learning for Bug Localization	67
6.1	Introduction	67
6.2	Motivation	69
6.3	Background	69
6.3.1	Bug Localization System	70
6.3.2	Reinforcement Learning	70
6.4	RLocator: Reinforcement Learning for Bug Localization	72
6.4.1	Pre-process	73
6.4.2	Formulation of RLocator	76
6.4.3	Developers' Workflow	80
6.5	Dataset and Evaluation Measures	81
6.5.1	Dataset	81
6.5.2	Evaluation Measures	82

6.6	RLocator Performance	85
6.6.1	Retrieval performance	86
6.6.2	Entropy Ablation Analysis: Impact of Entropy on RLocator performance	88
6.7	Threats to Validity	90
6.8	Conclusion	91
6.9	Chapter Summary	92
7	Bug Localization in Cross-Language and Cross-Project Settings	93
7.1	Introduction	93
7.2	Methodology	96
	Phase A Fine-tuning of Language Model	96
	Phase B Offline Indexing	101
	Phase C Retrieval Pipeline	102
7.3	Experimental Design	104
7.3.1	Research Questions	104
7.3.2	Dataset Curation of BEETLEBOX	104
7.3.3	Evaluation Benchmarks and Baselines	105
7.3.4	Configuration Setup	107
7.4	Results	107
7.5	Ablation Study	111
7.5.1	Impact of hybrid retrieval technique	111
7.5.2	Impact of fine-tuning	112
7.5.3	Impact of dynamic chunking	114
7.6	Threats to Validity	115
7.7	Conclusion	116
7.8	Chapter Summary	116

8 Conclusion and Future Work	117
8.1 Contributions and Findings	118
8.2 Prospects for Future Research	119
8.2.1 Customization and Personalization of Bug Localization Tools	119
8.2.2 Enhancing Bug Localization Tools Through Project-Specific Insights	120
8.2.3 Expanding the Application of Reinforcement Learning in Bug Local- ization	120
8.2.4 Explainable Reinforcement Learning for Bug Localization	120
8.2.5 Investigating Bug Propagation Across Projects	120
8.2.6 Investigating Multimodal Approaches for Bug Localization	121
References	122
APPENDICES	163
A Themes emerged from qualitative study	164
A.1 Themes identified from developers interview	164

List of Figures

1.1	An overview of the scope of this thesis.	3
2.1	An example bug report.	9
2.2	Overview of bug localization tools' working process.	12
4.1	Research methodology.	29
4.2	Time required to localize a bug.	35
4.3	Respondents' familiarity with LLM-based tools in relation to age and experience.	36
4.4	Perceived harms of bug localization tools by experience.	37
4.5	Concerns of adopting bug localization tools by familiarity with code assistance tools.	38
4.6	Current practices vs willingness to adopt.	39
4.7	Additional features in bug localization tools.	43
5.1	Steps of training a transformer.	51
5.2	Multi-modal embedding training pipeline.	55
6.1	Bug Localization as Markov Decision Process.	72
6.2	Effect of M in the reward-episode graph.	79
6.3	Developer interaction flow.	80
6.4	Feature importance of classifier model.	88

7.1	Kernel Density Estimate plot showing the distribution of similarities between buggy and non-buggy source code files and bug reports.	95
7.2	Example of mining hard examples.	97
7.3	Fine-tuning of language model (Phase A).	99
7.4	Offline indexing phase of BLAZE (Phase B).	101
7.5	Retrieval phase of BLAZE (Phase C).	102
7.6	Precision recall curves of the retrievers.	111
7.7	Scatter plots showing the chunks of the buggy and non-buggy files associated with the Apache Dubbo-1216 bug report. ● denotes bug report, ● denotes chunks from buggy files, and ● denotes chunks from non-buggy files.	113

List of Tables

3.1	Summary of the dataset.	15
3.2	Trustworthiness of the dataset.	15
4.1	Gender, educational level, and experience of the six interviewees.	31
4.2	Survey Questions.	33
5.1	Performance of the IR-based systems.	58
5.2	Average performance with varying data familiarity.	58
5.3	Average performance in different pre-training techniques.	59
5.4	Average performance with varying model architecture.	61
5.5	Average performance of the embeddings in six projects sorted by sequence length.	62
6.1	Description and rationale of the selected features.	75
6.2	Dataset statistics.	82
6.3	RLocator performance.	84
6.4	RLocator performance with and without Entropy for A2C.	89
7.1	Statistics on the used datasets, showing the distribution of bugs across training and testing, categorized by language for each dataset.	106
7.2	Comparative performance of BLAZE and cross-project bug localization tools.	108
7.3	Comparative performance of BLAZE and embedding-based bug localization tools.	109

7.4	Performance comparison between Lexical, Deep, and Hybrid Retriever. . .	112
7.5	Performance of CodeSage vs. BLAZE.	114
7.6	Static vs. Sliding window vs. Dynamic chunking.	115

List of Abbreviations

- AST** Abstract Syntax Tree 16
- BL** Bug Localization 1–6, 8, 10–14, 16, 18, 20–25
- DL** Deep Learning 4, 14, 16
- IP** Intellectual Property 4–6, 116
- IR** Information Retrieval 10, 14, 16, 24
- LLM** Large Language Model 7, 93, 118
- ML** Machine Learning 1, 14, 21
- MR** Merge Request 9
- PR** Pull Request 9, 12
- RL** Reinforcement Learning 5, 6, 21, 118

Chapter 1

Introduction

The modern world is increasingly dependent on software for driving automation and enhancing reusability across various sectors. In 2023, the U.S. software market was valued at approximately \$607 billion¹, underscoring the pivotal role that software plays in both enterprise and consumer applications. However, this growing reliance introduces significant challenges, particularly in ensuring the quality and reliability of software systems. A report from 2020 estimated that software bugs alone cost the U.S. economy approximately \$194.7 billion [160]. These costs extend far beyond just fixing individual bugs, encompassing failed projects, the upkeep of outdated systems, and operational disruptions. Ensuring high software quality is thus not only a technical challenge but a critical economic imperative.

Software bugs are deviations from intended behavior, leading to unwanted outcomes [159]. Fixing these bugs requires substantial developer effort, particularly in finding the specific source code file that needs correction. Prior research shows that debugging can take up to one-third of a developer's time [168, 306]. Often, the volume of bug reports surpasses the available resources for addressing them, causing delays in resolution. These delays can negatively impact customer satisfaction due to slower response times [77, 383]. The process of locating the source of a bug is especially challenging and error-prone in large-scale systems, which may consist of millions of lines of code [304, 209]. Any delay in identifying the root cause of a bug can slow down the development process, escalate the cost of fixes, and even result in extended periods of system downtime [212].

Modern approaches, such as automated bug localization (BL) tools and machine learning (ML) models, have been developed to help developers pinpoint the exact code files

¹<https://www.precedenceresearch.com/software-market>

that require fixing based on bug reports [323]. These tools aim to reduce the manual effort involved in the bug localization process, offering developers valuable time-saving assistance [84]. They use a range of techniques, including the analysis of code changes [54], test results [336], bug reports [359], and execution logs [45], to suggest the most probable locations of bugs. However, despite these advancements, accurately localizing bugs remains a complex and persistent challenge [173]. The high computational demands and complexity of these tools, along with their often limited ability to generalize across different systems [187], can constrain their effectiveness and make widespread adoption difficult in practice.

1.1 Problem Statement

While automated BL tools have the potential to enhance developer productivity, their practical application is hindered by several challenges. Overlooking these challenges can limit the tools' effectiveness:

Thesis Statement

Practical challenges in bug localization tools, such as low accuracy, hinder their widespread adoption by developers. Addressing these challenges through targeted solutions can not only improve the precision and accuracy of these tools but also enhance their appeal and usability, driving greater adoption.

This thesis addresses the gap between the current capabilities of BL tools and the expectations of developers. We aim to empirically identify these expectations and incorporate them into the tools, enhancing their accuracy and relevance in real-world development environments.

1.2 Thesis Overview

We now provide the scope of this thesis. Figure 1.1 highlights the overview of this thesis. First, we start by providing the necessary background information:

Chapter 2: *Background and Definitions*

In this section, we provide the reader with background information regarding the components of bug reports and an overview of the BL technique.

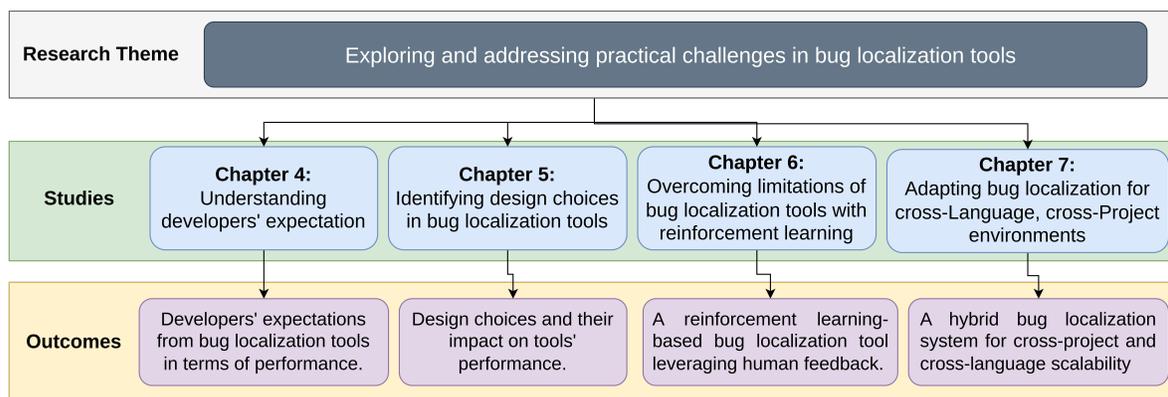


Figure 1.1: An overview of the scope of this thesis.

Chapter 3: *Related Works*

In order to situate this thesis with respect to prior research, we present a survey of research on software **BL** tools.

Next, we shift our focus to the main body of the thesis, concentrating on understanding developers' expectations from **BL** tools and tackling the associated challenges (blue boxes). For each objective, we aim to identify key outcomes (violet boxes). Our studies are organized into the chapters outlined below.

1.2.1 Understanding Developers' Expectation from Bug Localization Tools

Automating the process of identifying errors and tracing them to their root cause has the potential to significantly enhance developer productivity [84]. Despite the development of various automated **BL** tools in previous research [181, 347, 188, 374, 390], their integration into the software development lifecycle remains limited [253]. This limited adoption indicates a potential disconnection between research efforts and developers' actual expectations.

This thesis identifies practical challenges faced by developers in adopting **BL** tools through a combination of user surveys and qualitative interviews. These insights will ensure that the proposed **BL** tools address practical challenges faced by developers.

Chapter 4: *Understanding Developers' Expectations*

The discrepancy between research objectives in bug localization and developers' expectations undermines the productivity benefits these tools are

designed to provide. To investigate this gap, we conducted a mixed-methods study comprising qualitative interviews with six senior software practitioners and a quantitative survey of 95 developers. Our study provides valuable insights into developers’ views on **BL** tools, including practical challenges, performance expectations, use cases, and perceived benefits or limitations. A significant concern raised by developers is the low accuracy of current tools, which hampers their widespread adoption. In particular, they worry that these tools struggle in cases where historical data is insufficient or contains noise, further diminishing their reliability. Additionally, concerns about potential intellectual property (**IP**) leakage during the training process also contribute to practitioners’ reluctance to adopt such tools.

1.2.2 Exploring Design Choices for Building Bug Localization Tools

Deep learning (**DL**)-based techniques [124, 123, 317, 135, 388, 344] have recently surpassed traditional methods in **BL** by using semantic similarity. However, the semantic and lexical gap between **NL** and **PL** remains a persistent challenge.

Previous approaches, such as Word2Vec [215] and FastText [33], have made progress in narrowing the gap by mapping bug reports and source code into a shared vector space. However, these methods fall short by not fully capturing the contextual nuances of the text, which limits their overall effectiveness. The advent of more sophisticated models like BERT [66], which incorporate both semantic and contextual information, offers a promising new direction. Yet, these models introduce complex design considerations—such as model architecture, pre-training strategies, and data sources—that significantly influence the quality of the resulting embeddings. To develop effective **BL** tools that address the practical challenges identified in Chapter 4, it is crucial to understand how different design choices influence model performance.

Chapter 5: *Exploring Design Choices*

To investigate the impact of design choices, our research focused on three critical factors: the use of domain-specific data, pre-training methodology, and sequence length, and their influence on the performance and generalization of embedding models in **BL** tasks. The study involved training and evaluating 28 transformer-based models across two datasets using three different training methods. This chapter presents an analysis of how these design choices affected the performance of the models in **BL** tasks.

1.2.3 Addressing Challenges in the Use of Bug Localization Tools

Traditional BL tools [317, 135, 388, 344] predominantly rely on supervised learning approaches [189], which depend on labeled data for accurate bug identification and localization. However, these tools often face issues with limited generalizability [327], requiring retraining for each new project before they can be applied effectively. This retraining poses significant challenges in practice: new projects frequently lack sufficient labeled data for proper model training [56], and training separate models for each project can become economically taxing and lead to long-term maintenance problems, as continual retraining and updates are required [142]. In Chapter 4, we have identified developers’ concerns about the low accuracy of these tools, particularly in cases of limited or noisy historical data, restricting their real-world applicability. Additionally, concerns about IP leakage during training present further obstacles to adoption.

To address these challenges, we propose two BL tools: RLocator and BLAZE. These tools are designed to enhance generalizability and reduce the dependency on extensive retraining for new projects, offering a more scalable and maintainable solution for BL in diverse and dynamic software environments.

Chapter 6: *Reinforcement Learning-Based Bug Localizer*

The scarcity of labeled data presents a challenge for applying BL tools in new projects. Reinforcement Learning (RL) can effectively address this issue. By allowing an RL agent to learn directly from user feedback, the reliance on labeled data is eliminated [302]. In this chapter, we address the BL problem by formulating it as a Markov Decision Process and utilizing RL techniques for bug localization. Our evaluation demonstrates that this approach not only addresses the lack of labeled data but also directly responds to one of the key concerns developers have regarding the adoption of bug localization tools: accuracy. The proposed method improves performance by a significant margin of up to 38.3% in terms of Mean Average Precision (MAP), offering a dual solution that tackles both data scarcity and accuracy challenges commonly encountered in BL tools.

Chapter 7: *Enhancing Generalizability Through Hard Example Learning*

The limited generalizability of BL tools poses a significant maintainability challenge and increases overhead costs, ultimately diminishing the productivity gains these tools are designed to offer. In this chapter, we investigate the use of hard example learning to improve generalizability. Furthermore, we propose methods to overcome challenges related to modeling lengthy

source code files. Our approach enables bug localization in new projects without the need for additional training, effectively alleviating developers’ concerns about potential IP leakage, as highlighted in Chapter 4. Furthermore, our evaluation on 31k bugs demonstrates that the proposed solution outperforms six state-of-the-art BL tools by up to 144% in terms of MAP.

1.3 Thesis Contribution

This thesis demonstrates that:

1. Although developers are open to adopting BL tools, significant challenges remain due to concerns about accuracy, performance, and the risk of intellectual property (IP) leakage. BLAZE addresses this by eliminating the need for training on proprietary code, thus ensuring no risk of IP leakage (Chapter 4, and 7).
2. BL tools are particularly beneficial for junior developers, serving as an effective aid in training and skill development, especially for those with limited experience (Chapter 4).
3. Training embedding models with project-specific data greatly improves the performance of BL models, with models trained on relevant datasets showing up to a 76% improvement compared to those trained on generic datasets (Chapter 5).
4. Longer sequence transformers, like LongCodeBERT, have demonstrated improved performance in BL tasks. However, the performance gains vary across different projects, suggesting the need for careful resource allocation (Chapter 5).
5. Modeling the BL problem as a reinforcement learning (RL) problem can effectively reduce the tools’ dependency on labeled data, offering a more flexible approach to training (Chapter 6).
6. Directly optimizing performance metrics through RL can enhance the effectiveness of BL tools, achieving up to a 38% improvement in key performance indicators (Chapter 6).
7. We contribute the largest cross-language and cross-project BL dataset to date, encompassing over 26,000 bugs from 29 projects across five programming languages, providing a valuable resource for future research (Chapter 7).

8. We propose a hard example learning technique tailored for the software engineering domain, specifically for bug localization. Our technique demonstrates a significant performance boost, achieving up to a 120% increase in Top-1 accuracy, underscoring its effectiveness in challenging scenarios. (Chapter 7).
9. Dynamic chunking is introduced as a new method to overcome the context window limitations of Large Language Models (LLM). It significantly enhances the ability to process long source code files while minimizing continuity loss (Chapter 7).

1.4 Thesis Organization

The structure of this thesis is as follows: Chapter 2 begins by establishing the foundational definitions and key terms necessary for understanding the subsequent work. Chapter 3 reviews related research, situating this thesis within the broader academic context. In Chapter 4, we delve into developers' expectations of bug localization tools, marking the start of the main body of the thesis. Chapter 5 addresses another practical challenge by examining the impact of design choices on training embedding models for bug localization. Chapter 6 explores the application of reinforcement learning to bug localization tasks. In Chapter 7, we assess the benefits of learning from hard examples in enhancing the generalizability of bug localization tools. Finally, Chapter 8 concludes the thesis and suggests directions for future research.

Chapter 2

Background and Definitions

This chapter will discuss the various types of **BL** tools, outline the generalized methodologies employed by these tools, and provide essential information necessary for a comprehensive understanding of **BL** mechanisms.

2.1 Bug Localization Terms

To lay the foundation for the rest of this thesis, we define key terms related to **BL** process in this section.

2.1.1 Bug report

A bug report is a documented record submitted by users, testers, or developers that describes an issue or defect in a software application. It typically includes details such as a summary of the problem, steps to reproduce the issue, the expected versus actual behavior, and any relevant system information or logs [391]. The goal of a bug report is to provide enough information for developers to identify, diagnose, and fix the underlying problem in the software. Figure 2.1 represents a sample bug report¹ from the AspectJ project. It mainly includes a bug ID, title, textual description of the bug, and version of the codebase where the bug exists.

¹https://bugs.eclipse.org/bugs/show_bug.cgi?id=134471

Bug 134471 - adding whitespace to an aspect loses crosscutting → Title

Bug ID Status: RESOLVED FIXED Reported: 2006-04-03 05:32 EDT by Helen Beeken
 Alias: None Modified: 2012-04-03 14:13 EDT (History)
 CC List: 0 users

Product: AspectJ
 Component: Compiler (show other bugs)
 Version: DEVELOPMENT
 Hardware: PC Windows XP

See Also: Version of the codebase

Importance: P1 critical (vote)
 Target Milestone: 1.5.2
 Assignee: Andrew Clement
 QA Contact:

Helen Beeken 2006-04-03 05:32:01 EDT Description

Given the following aspect:

```
package pkg;
public aspect A {
    pointcut p() : call(* pkg.*(..));
    before() : p() {
    }
}
```

Example code snippet

Textual description

Adding a whitespace (or anything else in the aspect) and saving results in an incremental build and all crosscutting information is lost. Everything is regained after a full build.

This is a regression from AspectJ 1.5.0 and could be related to [bug-133532](#).

Figure 2.1: An example bug report.

2.1.2 Pull Request

After a solution for a bug is identified, a contributor referred to as the submitter, submits a request to integrate the change into the target branch [378]. This request is known as a Pull Request (PR) or Merge Request (MR) and requires approval from project maintainers, who have the authority to merge the code, often called mergers [378].

2.1.3 Bug Localization

Software bugs are deviations from intended behavior, leading to unwanted outcomes [159]. While bug localization and fault localization are two close concepts, there are subtle differences [112]. Fault localization involves identifying specific parts of the software responsible for failures by analyzing test results, code coverage, and execution data. In contrast, bug localization maps bug reports to relevant code sections using information retrieval or ma-

chine learning techniques. Consequently, bug localization tools can be applied even when testing information is not available.

2.1.4 Types of Bug Localization

In a broader sense, software bug localization can be divided into seven categories [396],

- **Spectrum-based fault localization (SBFL)** [352, 2, 104]: using test coverage information.
- **Mutation-based fault localization (MBFL)** [221, 245]: using test results from mutating the program.
- **Dynamic program slicing** [275, 5]: using dynamic program dependencies.
- **Stack trace analysis** [339, 340]: using crash reports.
- **Predicate switching** [377]: using test results from mutating the results of conditional expressions.
- **Information-retrieval-based bug localization (IR-based BL)** [383]: using bug report information.
- **History-based fault localization** [147, 263]: using the development history.

The first five techniques rely on test cases; however, it is possible for bugs to remain undetected even after these test cases have passed. Additionally, the effectiveness of these techniques diminishes when test coverage is low or the test cases are insufficiently rigorous. Furthermore, Zou et al. [396] observed that these five techniques are time-intensive, as they necessitate processing the entire test set. The time required for these approaches ranges from 80 seconds to 4800 seconds, in contrast to information retrieval and history-based techniques, which require approximately 10 seconds per data point.

2.1.5 Document Vectorizer

The IR-based bug localization tools use vector similarity to determine the similarity between a query and a document. In the first step, the document is tokenized after necessary pre-processing, and after that, they are vectorized. The most common approaches for vector conversion are TF-IDF, BM25, and their variants.

Term frequency-inverse document frequency (TF-IDF)

Term frequency represents the relative frequency of a particular term t in a document d . It is calculated using the following formula,

$$tf(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \quad (2.1)$$

Inverse document frequency measures how common a particular term t in a document set D . It is calculated using the following formula,

$$idf(t, D) = \log \frac{|D|}{|d \in D : t \in d|} \quad (2.2)$$

Finally, the TF-IDF of a document d and term t is calculated by multiplying the term frequency and the inverse document frequency.

$$tfidf(t, d) = tf(t, d) \times idf(t, D) \quad (2.3)$$

To use the TF-IDF-based similarity, the IR system vectorizes each document by replacing each token with its corresponding TF-IDF score. Finally, the IR tool calculates similarity using cosine similarity between two vectors.

BM25

BM25 is a more robust approach that uses a slightly different formula with parameters $K1$ and b . IR systems use the BM25 scoring with a pipeline similar to TF-IDF.

$$idf'(t, D) = \ln \frac{|D| - |d \in D : t \in d| + 0.5}{|d \in D : t \in d| + 0.5} + 1 \quad (2.4)$$

$$BM25(t, d) = idf'(t, D) * \frac{tf(t, d) \times (K1 + 1)}{tf(t, d) + K1 \times (1 - b + b \times \frac{length(d)}{avg. length of D})} \quad (2.5)$$

2.2 Overview of Bug Localization Process

Figure 2.2 provides an overview of BL tool's working process. Below, we describe each step of this process:

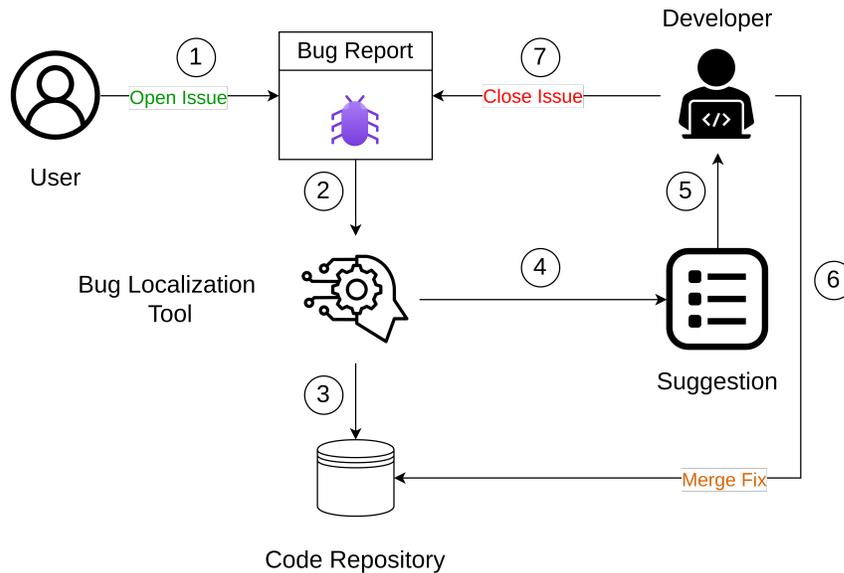


Figure 2.2: Overview of bug localization tools' working process.

1. **Bug Reporting:** When a user, either a client or developer, identifies a discrepancy in the existing system, they file a bug report. This report documents the unexpected behavior and may include relevant details such as stack traces, logs, and steps to reproduce the issue. Platforms like GitHub, GitLab, and Bugzilla are commonly used to manage and track these reports. In Figure 2.2, step 1 illustrates the bug reporting stage.
2. **Initiation of Bug Localization Tools:** Upon submission of a bug report, a BL tool will be triggered. Utilizing the information provided in the report, the tool will analyze the code repository to pinpoint the source code files likely responsible for the bug. This process may involve leveraging ad-hoc data, such as historical bug records, previous file modifications, and the user's past reports. The tool will then generate suggestions for the developer, with recommendations ranging from the file level to specific methods or lines of code. In Figure 2.2, steps 2–4 illustrate the suggestion generation stage.
3. **Developer Resolution:** Based on the suggestions provided by the BL tool, the developer will proceed to create a fix for the bug. This fix may involve updates to one or multiple files within the code repository. The fix will be submitted as a PR, which the repository maintainers review for code quality and compatibility with existing test cases. Once the PR is merged into the repository, the developer

will update the bug report status from ‘open’ to ‘closed.’ In Figure 2.2, steps 5–7 illustrate the bug report resolution stage.

2.3 Chapter Summary

In this chapter, we presented an overview of BL tools and their role in the bug resolution process. Additionally, we defined key terms relevant to this thesis. With this foundational understanding of the essential concepts and terminology, we proceed to the next chapter, where we discuss related studies to highlight the gap that this thesis aims to address.

Chapter 3

Related Work

In this chapter, we review the relevant research on bug localization systems. We categorize the work into three major groups: data and embeddings, learning and models, experiments, and applications.

3.1 Data and Embedding

3.1.1 Dataset

Available Dataset

In the development of [ML/DL](#)-based model, the dataset performs a crucial role. Models trained on a good-quality dataset will learn to generalize better. Kim et al. [\[145\]](#) have identified five unique datasets for [IR](#) based [BL](#) by investigating 50 different studies since 2012. These datasets are comprised of various programming languages. A short description of the datasets is presented in [Table 3.1](#). We can observe that Bench4bl [\[173\]](#) contains the largest number of projects, and the dataset proposed by Ye et al. [\[363\]](#) contains the largest number of bug reports. Moreover, most of the datasets cover only Java language.

Trustworthiness of the datasets

Kim et al. [\[145\]](#) have identified that most of the datasets used for [IR](#)-based [BL](#) are not trustworthy. By untrustworthy, they mean the ground truths are not accurate. To measure

Table 3.1: Summary of the dataset.

Dataset	# Projects	# Bug Report	Languages	# Source Files			
				Min	Avg.	Med	Max
Zhou et al. [383]	4	3479	Java	391	4665	2986	12298
Dit et al. [74]	5	417	Java	531	2845	1069	10607
Sisman et al. [293]	2	5053	Java, C/C++	19326	21433	21400	23540
Moreno et al. [224]	13	599	Java	301	2366	2866	4457
Ye et al. [363]	6	22747	Java	382	5413	4692	10546
Bench4BL [173]	46	9459	Java	23	1510	654	14529

Table 3.2: Trustworthiness of the dataset.

Dataset	Ambiguous Report	
	Number	Rate (%)
Zhou et al. [383]	201	5.8
Dit et al. [74]	18	4.3
Sisman et al. [293]	1031	20.4
Moreno et al. [224]	264	44.1
Ye et al. [363]	2260	9.9
Bench4BL [173]	4001	42.3

the quality of the ground truth, they sampled data from each of the datasets mentioned in Table 3.1 and manually checked the ground truth from the corresponding repository. Kim et al. have defined a new term, “ambiguous report.” According to their definition, an ambiguous report is a bug report associated with both production and test source code. They have argued that ambiguous bug report potentially has the wrong ground truth as there is a low possibility that a bug is associated with both production and test source code. The findings of Kim et al. have been presented in Table 3.2. Table 3.2 shows that Bench4Bl contains the highest percentage of ambiguous reports, and the dataset proposed by Ye et al. [363] contains the lowest rate of ambiguous reports. Kim et al. have also

estimated the effect of ambiguous reports on the performance of an IR-based BL system. They have found that the ambiguous bug report can falsely increase the performance of the IR-based BL systems from 12.2% to 13.1%.

Data used in bug localization

All the research work we have found has used bug report-source code pairs for localizing bugs. However, some studies have used metadata (such as bug fixers’ information, time) [165, 348] and history (including bug frequency and bug recency) [217, 341, 165, 348] alongside bug reports to enhance the performance of localization models. While previous studies claimed that incorporating history and metadata improves performance, there has not been any controlled study to measure this impact. For example, history from bug reports can be used to identify source code files that were updated to fix previous bugs similar to the current one, thus prioritizing these files in the candidate set.

In addition to metadata, some studies have employed *code structural information*, such as Abstract Syntax Tree (AST) and Control Flow information. The AST is a tree representation of the syntactic structure of source code, while the control flow represents the execution order of statements. ASTs are widely used to detect syntactic similarity in code [15, 37]. For instance, CAST [188] employs a modified AST by removing library method invocations based on the intuition that user-defined methods are more likely to contain bugs. This pruning simplifies the problem for deep learning models. They then extracted Word2Vec embeddings for each AST node and used a Tree-based CNN (TBCNN) [226] to represent the entire AST. Similarly, Kim et al.[148] included method invocation and “import” statements as special edges, using TBCNN for representation. Xiao et al. [348] trained Word2Vec embeddings for both bug reports and source codes, vectorizing AST nodes and applying a CNN model to estimate similarity. Wang et al.[326] used a Deep Belief Network to extract syntactic information from an AST, while CG-CNN [122] used control flow graphs to extract structural information from code, applying CNN and LSTM to estimate source code representation from the control flow graph. However, in this thesis, we focused on only bug reports and source code files and used them as text.

3.1.2 Representation Learning

The first step of DL-based bug localization techniques is learning the representation of text (source code, bug report). Bug reports are in natural language (NL), whereas source codes are in the programming language (PL). Thus before applying any tools, we need

to transfer them to the representation. The representation function is often referred as an ‘Embedding’. Embeddings project both NL and PL in the same embedding space (a multi-dimensional space). In the curated list of literature, we have observed three major categories of embeddings.

Statistical Embeddings

TF-IDFs [217, 367, 317, 376, 344, 165, 348] and N-grams [96, 326] are examples of statistical embeddings. In this approach, the studies tried to estimate the statistical property of word(s) in a sentence and used that as the representation of that specific word. We can say that the statistical value represents the word if the statistical property is calculated on a large text dataset. One major drawback of this kind of representation is that they do not consider context or inter-word relations. 39% of the literature used statistical embedding in their bug localization (BL) model.

Vector Embedding

Word2Vec [53, 148, 124, 376, 188, 181, 347, 249, 348], Doc2Vec [217, 249, 341], GloVe [195], and FastText is example of vector-based embedding. Word2Vec and FastText generate the representation of each word, whereas Doc2Vec generates the representation for each document from a neural model. For similarity comparison, we need the representation of the whole document (bug report, source code). Thus, studies used several approaches like CNN, average, or weighted average of the word representation to get the document representation if they were using Word2Vec or FastText. 57% of the literature used vector-based embedding in their model.

Embeddings from autoencoder

Embeddings generated from BERT, GPT, or other Transformer [39] based systems are often used as embeddings in BL models. We can observe that only a handful of the studies have used embeddings generated from an autoencoder in their model. Yang et al. [357] have used an autoencoder along with CNN to get the similarity score. However, some of the studies have guessed that context-based embedding (embeddings from the transformer) may contribute to increasing the performance. Kanade et al. [140] compared the performance of the BERT-based language model with other embedding techniques (e.g., Word2Vec). They found that the performance of the BERT-based language model is higher than the other embedding techniques. Feng et al. [83] proposed a multi-modal BERT-based

language model [66] for both programming language and natural language. The proposed language model supports multiple programming languages like Go, Java, Php, Javascript, Python, and Ruby. For training the model, Feng et al. used a code-documentation pair collected from Github. GraphCodeBERT is a follow-up work of CodeBERT by Guo et al. [99]. In this model, they have incorporated data flow diagrams with text data (NL and PL). Ahmad et al. [6] have proposed an encoder-decoder model for the PLUG (Program and Language Understanding and Generation) task. Their model has an encoder to encode information from NL and PL data and a decoder for generating text or code based on the task. A BERT-based transformer model was used as the encoder, and the encoder was trained on GitHub and Stack Overflow data. Lu et al. [201] proposed a large dataset for ten tasks related to program understanding and generation. Moreover, they have included baseline systems for these tasks. In their baseline system to understand the programs, they have used CodeBERT [83] as the encoder and CodeGPT as the decoder. Both the encoder and the decoder have been trained on the CodeSearchNet [126] dataset. Previous studies proposed language models for programming language following different architectures of transformers and training techniques. However, those studies do not answer the impact of various transformer architectures and training techniques on the generated embeddings or the performance of a bug localization system. Wang et al. [328] have used the T5 architecture to develop the encoder-decoder model CodeT5. Their proposed model uses token-type information to generate the code representation. There are two other autoencoder models for source code. One is the GPT-based model Codex [47], and the other one is the seq2seq-based model SPT-Code [239].

Thus, in this thesis, we conducted an empirical study to understand the impact of a design choice on the performance of embedding models in BL task.

3.1.3 Embedding Granularity

Based on the granularity of embedding, Chen et al. [52] have classified the embeddings of source codes into four levels

Embeddings of Token

. These types of embeddings [103, 334, 21, 51] are generally generated from a Word2Vec model. First, the source code is tokenized, and then it is fed into a Word2Vec model to generate embeddings. The generated embeddings are used in various applications, including bug localization/fault localization, automatic program repair, code clone detection, and code search.

Embeddings of Function/Method

Embeddings of function/method are mostly generated from code structure such as AST [15, 68, 37] or control flow graph [65]. Alone et al. [15] have used AST and methods path to generate the embedding of a method. Lu et al. [200] have proposed to generate embedding from Function Call Graph (FCG). From Ricci’s curvature of the FCG, they identified the underlying geometry of the FCG, which is a hyperbolic space. Later, they used the Poincaré model to learn the embeddings. One problem with using the function/method level embedding in bug localization is the lack of structural data in the input. In bug localization, we mainly use source code and bug reports as input. The source code contains structural information, whereas the bug report does not contain such information. Embedding of functions/methods mainly encodes the code’s structure in a multi-dimensional space. Thus this kind of embedding may not be impactful in IR-based bug localization using bug reports.

Embeddings of Sequence

Pradel et al. [256] have trained a Word2Vec-based model to generate the embedding of source codes. In their training methodology, they created artificial buggy code and later trained embeddings to increase the difference between buggy and non-buggy codes’ embeddings. Finally, those embeddings are used to detect name-based bugs in source codes.

Embeddings of Binary Code

Xu et al. [354] have converted binary functions into a control flow graph and later used Struct2Vec to generate embeddings of binary code. Zuo et al. [397] have used a Siamese network of LSTM to calculate the embedding of binary blocks. First, they have developed binary instructions for different architectures (say, X86 and ARM). Later, they are fed into the model. The study has optimized the model so that the representation of the code in two platforms should have a minimum distance.

While prior studies have proposed embedding at different granularities, this thesis has focussed on embedding at file and chunk levels. Our selection of chunks as granularity allows us to change the granularity with higher flexibility.

3.2 Learning and Models

3.2.1 Types of models

We can observe the use of two major model types in the BL task. In the first type of model, the model is trained to estimate the similarity/probability that a source code file is associated with a bug report. The other major type is the Learning to rank (LTR) model, where the model predicts the rank of the candidate files. The two types of models are described in detail below.

Similarity/probability Estimation Model

In our observation, most of the existing BL models [348, 344, 165, 53, 148, 317, 124] follow this strategy. This approach first transforms the report and source code into a vector. Next, they are fed into a deep neural network (CNN, LSTM, GRU, MLP) and associated data (hierarchical information, history, or metadata). The output from the deep neural network estimates the similarity between the bug report and source code or the probability that the particular source code file is buggy. Finally, the source code files are ranked based on similarity/probability. We have seen in earlier sections that word-vector-based embeddings (e.g., Word2Vec, GloVe, FastText) are the most popular embedding technique in previous studies. The BL tools need to find a way to generate the representation of the entire document from the word embeddings. Studies have used CNNs for that purpose. Most of the studies [300, 367, 148, 317, 124, 194, 188, 181, 347, 344, 348] have used CNNs as an intermediate layer for representation generation. However, some studies have extracted hierarchical information from the source code using ASTs. They have used a specialized CNN model to capture the hierarchical information for that purpose. Kim et al. [148] used CNN4TFIDF, Li et al. [181] used Graph-CNN, and Liang et al. [188] used Tree-based CNN in their studies. Studies have used [53, 341, 326] deep neural network (multi-layer perceptron) as the final layer or to combine different kinds of information such as metadata, history, and hierarchical information with textual information. Li et al. [181] and Xiao et al. [344] have used recurrent units (e.g., LSTM, GRU) to generate the embedding of the whole document in their study. Later, they are passed through an MLP to generate the final prediction. Some studies have incorporated the encoding-decoding technique in their tool to identify buggy files. Xiao et al. [344] have used LSTM-based encoder-decode architecture. First, they used embeddings of the words in a bug report in LSTM to generate the representation of the full bug report. Later, the bug report embedding has been used as context (incorporated with subsequently generated context) and fed into an LSTM unit

along with the word embeddings of a source code to generate the final predictions. In another study, they [345] used a machine translation-based approach and LSTM in an encoder-decoder architecture.

Learning To Rank Models

We have observed only “similarity estimation” models in our list of literature. However, another popular model category, “Learning to Rank” (LTR), which is widely used in NLP, is not present here. Liu et al. [196] have categorized LTR models into three categories. *Pointwise techniques* transform the ranking problem into a regression problem, whereas *pairwise techniques* approximate the problem by a classification problem: creating a classifier for classifying item pairs according to their ordinal position. *Listwise techniques* take ranking lists as input and evaluate the ranking lists directly by the loss functions. However, due to the large source code files and hardware size limitation, the listwise ranking may not be viable for BL. For example, say we have 753 files in a repository, and following the listwise method, we have to load 753 files in memory/GPU. In this method, the size requirement of memory/GPU will be proportional to the number of files. In the similarity estimation models, authors tried to estimate the similarity between a bug report and source code files, and after that, they ranked them accordingly. Le et al. [22] have used the listwise technique to rank buggy methods. However, they have used the clustering technique to shortlist the candidate methods for their learning-to-rank model.

In this thesis, we proposed two BL tools, RLocator and BLAZE. RLocator is a RL based agent, and BLAZE is an embedding-based model. However, regarding types, RLocator is the LTR model, and BLAZE is the similarity estimation model.

3.3 Experiments and Applications

3.3.1 Experimental Setting

We have observed two experimental settings in previous studies. The two experimental settings are in-project and cross-project training.

In-Project Settings

In-project settings refer to scenarios where a model’s training and testing data are sourced from the same project. This approach aligns with the ML principle that both train-

ing and testing data should come from the same distribution. Most of the tools we reviewed [53, 367, 317, 376, 195, 188, 341, 347, 249, 344, 345, 165, 348, 326, 22] have adopted in-project settings, meaning they trained a separate model for each project in their dataset. A preliminary step in applying DNN techniques involves projecting text into a multi-dimensional space using an embedding model. A handful of studies have employed both project-specific embeddings and models. However, the methodology often makes it unclear whether project-specific embeddings were used. Only Globug [217] explicitly mentioned using a global embedding for their models.

Cross-project Settings

Most of the proposed BL tools require project-specific training before they can be used effectively. These models need to be trained on a particular project to perform well. However, a significant challenge arises when these tools are applied in cross-project settings, where their performance can drop dramatically (approximately 50%). This issue makes project-specific training impractical in many situations. Cross-project models, on the other hand, are designed to be trained on one project but used across different projects without a loss in performance. Four studies have attempted to tackle this problem. Miryeganeh et al. [217] addressed it by using a global embedding—an embedding trained from source code across all projects—in their dataset. Kim et al. [148] approached the problem by representing source code as a graph. They extracted tokens, AST, control flow graph, and call graph, then added extra edges and nodes to combine the AST with the control flow and call graphs, ultimately creating an embedding of the combined graph. TRANP-CNN [124] used features from both in-project and out-project source code to train a CNN model. However, because they partially used data from other projects during training, it can be considered a partial cross-project BL tool rather than a fully cross-project one. Li et al. [181] used both global and local contexts to determine whether a particular method is buggy. Their tool outperforms the state-of-the-art in method-level BL. The global context includes a program dependency graph and data flow graph, while the local context consists of the AST. They combined these local and global contexts using an attention mechanism to predict whether a method is buggy.

In this thesis, we identify that developers believe the inability to perform cross-project is one of the major shortcomings of BL tools. To overcome this, we propose a cross-project cross-language bug localizer BLAZE.

3.3.2 Evaluation Metrics

There are two major criteria for measuring the performance of **BL** tools. The first one is retrieval performance which measures how good the tool is in retrieving a relevant document. The second one is the latency/inference performance which measures how fast the tool can identify buggy files. To be used by developers, both types of performance are required. A tool with good retrieval performance will not work if it takes too long to identify the buggy files. The following sections will discuss the use of evaluation metrics in previous studies.

Retrieval Performance

Most studies have approached the **BL** problem as an information retrieval challenge, with only a few treating it as a classification problem. Common metrics used to evaluate information retrieval-based solutions include Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), and Top-K [363]. MRR focuses on the best (smallest) retrieved rank, calculating the inverse of that rank, while MAP takes into account the ranks of all related files to assess performance. For instance, if multiple buggy files are related to a bug report, MAP evaluates the overall ranking of these files, not just the top one. This approach highlights the importance of considering the rank of all relevant files in **BL** evaluation. On the other hand, classification-based studies have typically used precision, recall, and F1 scores for evaluation. On the other hand, Top-K calculates the ratio of the cases where a buggy file has been ranked in a position less than or equal to k.

Inference Performance

In addition to retrieval performance, it is crucial to consider inference performance in **BL** tools. Many tools rely on similarity or probability estimation approaches, which may require evaluating a large number of files—such as the 753 files in the dataset provided by Ye et al. [363]—to identify the relevant source code files. Therefore, the time performance of a **BL** tool is as important as its retrieval accuracy. However, only a few studies have reported the time required for prediction per bug report. The inference times vary significantly across different tools, ranging from 0.53 to 294.3 seconds, depending on the complexity of the tool. For instance, simpler tools like MD-CNN [317] tend to have shorter inference times, while more complex tools like CAST [188], which use AST to capture the structural properties of code, require longer time for inference. Interestingly, increased complexity does not always correlate with better retrieval performance, but it often results in longer inference times.

In this thesis, our evaluation focuses solely on retrieval performance, excluding inference performance, as the latter often relies on the underlying hardware and the complexity of the dataset.

3.3.3 Bug Localization System in Industry

Among forty studies in our literature set, only one study has proposed a solution that is used in commercial settings. Scaffle [255] offered a BL tool to localize bugs from the stack trace. It has been used in the Facebook codebase. One major problem with localizing bugs from stack-trace is the stack trace is long, and there is a lot of unnecessary information in the stack trace. To solve the issue, Scaffle [255] uses a trace line model to identify the stack traces’ essential lines. The trace line model has been developed using Word2Vec embedding and GRU. After identifying the critical lines, Scaffle uses an IR tool to retrieve the relevant source code files. Scaffle [255] uses a proprietary dataset for evaluation, and in that dataset, they have achieved MRR up to 0.8. Bug2Commit [229] is another tool that has been proposed by Facebook and used by Facebook to localize bugs. However, the goal of Bug2Commit [229] is to localize buggy commits, not the buggy files. Thus, it is out of the scope of this study. In the proposed tool, they used FastText embedding and calculated the similarity between embedding a bug report and commit diff. They have achieved the highest MRR of 0.41 in a proprietary dataset. In this thesis, we have identified the adoption rate of BL tools is low. The observation is further supported by this analysis of the use of BL tools in industry.

3.3.4 Complete tool

In our review of the literature, we did not find a comprehensive tool designed for developers. By “comprehensive tool,” we mean a BL tool that can be seamlessly integrated into any software project via a version control system or custom hook without requiring additional training or infrastructure. We also conducted searches on platforms like GitHub, GitLab, and the Git marketplace using keywords such as “Bug Localization” and “Fault Localization.” Among these, we found only one tool, IncBL¹, which provides an IR-based BL service. IncBL leverages VSM similarity to localize bugs [359]. Similar to the previous section, this observation bolsters our findings regarding the low adoption rate of BL tools in the industry.

¹<https://github.com/apps/incbl>

3.3.5 Chapter Summary

This chapter reviewed studies on [BL](#), embeddings, and the quality of [BL](#) datasets, all of which are relevant to the core chapters of this thesis. It also positioned our work within the broader research context. In the next chapter, we begin the main body of the thesis by presenting a study on the expectations from [BL](#) tools and addressing the associated challenges.

Chapter 4

Bug Localization Tools in Practice: Developer Perspectives and Expectations

Note. The work is prepared to be submitted at an upcoming software engineering conference.

4.1 Introduction

A bug localization tool aims to boost productivity by automating the process of identifying bugs. While previous studies have proposed various tools for automated bug localization [181, 347, 188, 374, 390], their integration into the software development lifecycle has been limited [253]. This limited adoption suggests a possible disconnection between research efforts and developers' needs.

Several studies have investigated developers' expectations for fault localization tools [316, 158]. However, fault localization differs slightly from bug localization [112]. Fault localization involves identifying specific parts of the software responsible for failures by analyzing test results, code coverage, and execution data. In contrast, bug localization maps bug reports to relevant code sections using information retrieval or machine learning techniques. Consequently, bug localization tools can be applied even when testing information is not available. Furthermore, these studies are somewhat outdated and may not accurately represent the evolving expectations of developers.

Like many other fields, software development has been significantly impacted by the recent emergence of Artificial Intelligence (AI) and Large Language Models (LLMs) [139]. AI assistants, such as GitHub Copilot [61] and Visual Studio IntelliCode [299], have become invaluable assets. These tools assist developers in various tasks, including code generation [242], bug fixing [29], refactoring [59], and testing [307]. They support nearly every aspect of the software development lifecycle. Therefore, developers' perceptions and expectations may have been influenced by the emergence of these new technologies.

Thus, in this study, we aim to understand developers' perspectives on bug localization tools and their evolving expectations. We conduct a mixed-methods study combining qualitative and quantitative approaches. We begin with in-depth interviews with six senior software practitioners to understand their views on these tools. After transcribing the interviews, we use open coding and card sorting to identify key themes.

To validate these themes, we design and distribute a survey to a broader community of software developers, collecting 95 responses. This combination of qualitative interviews and quantitative surveys allows us to comprehensively understand developers' expectations and perceived challenges, providing valuable insights for future improvements in bug localization tools. Our study revolves around the following research questions:

RQ1. What are developers' current approaches and perspectives on bug localization tools? This research question aims to understand the methods developers currently use to localize bugs and their views on the advantages and disadvantages of using bug localization tools. Understanding the current approaches will serve as a baseline, helping us to better recognize developers' expectations of bug localization tools.

RQ2. What are developers' expectations from automated bug localization tools? Understanding these expectations is vital for enhancing bug localization tools' usefulness and user satisfaction. This research question aims to understand developers' expectations and thereby identify the reasons for the lack of adoption of bug localization tools in the software development lifecycle. Furthermore, it will highlight areas where current tools are particularly effective and where they require enhancements, offering guidance for future development.

RQ3. Who are the primary users of bug localization tools, and what are the tools' focus areas? Identifying the target users and use cases will help researchers design more effective tools and identify existing drawbacks. This question explores the roles (e.g., developers, QA engineers) and experience levels (e.g., senior, junior) of users, as well as the complexity of the bugs addressed. Understanding

these aspects is crucial for tailoring tools to meet user needs and improve their functionality.

RQ4. What features do developers value in automated bug localization tools?

Previous studies suggest that developers' low adoption of any technology may be due to a lack of expected features in the tools [218]. To address this gap and guide future research, we aim to identify the specific features developers value in bug localization tools. Understanding these preferences will help us design these tools to better meet developers' needs and improve their adoption rates.

Our study finds that while developers are willing to adopt bug localization tools, they face significant challenges. Approximately 76% of respondents expressed willingness to use automated bug localization tools. The primary concern is the performance of these tools in accurately localizing bugs, with 51% of respondents uncertain about the correctness of outputs. Developers also expressed concerns about potential intellectual property (IP) leaks, with 30% of respondents highlighting this issue.

Compared to prior studies, our findings suggest that developers' performance expectations have evolved to be more relaxed, with the expected timeframe for results ranging from one to fifty-nine minutes and a median of eight minutes. Despite believing that these tools can improve productivity, developers remain skeptical about their performance and the dependency they might create. Notably, only 27.5% of developers are currently familiar with advanced AI tools for bug localization.

This paper makes the following contributions:

- We conducted in-depth interviews and surveys with 95 software developers, providing a robust understanding of their real-world challenges and needs.
- Our study identified critical features like an explanation of suggestion and workflow integration and highlighted adoption barriers, including accuracy, performance, and IP concerns.
- We offer insights for researchers and suggest future research directions based on practitioners' expectations and perceived challenges with bug localization.

4.2 Methodology

As illustrated in Figure 4.1, this study employs both qualitative and quantitative method-

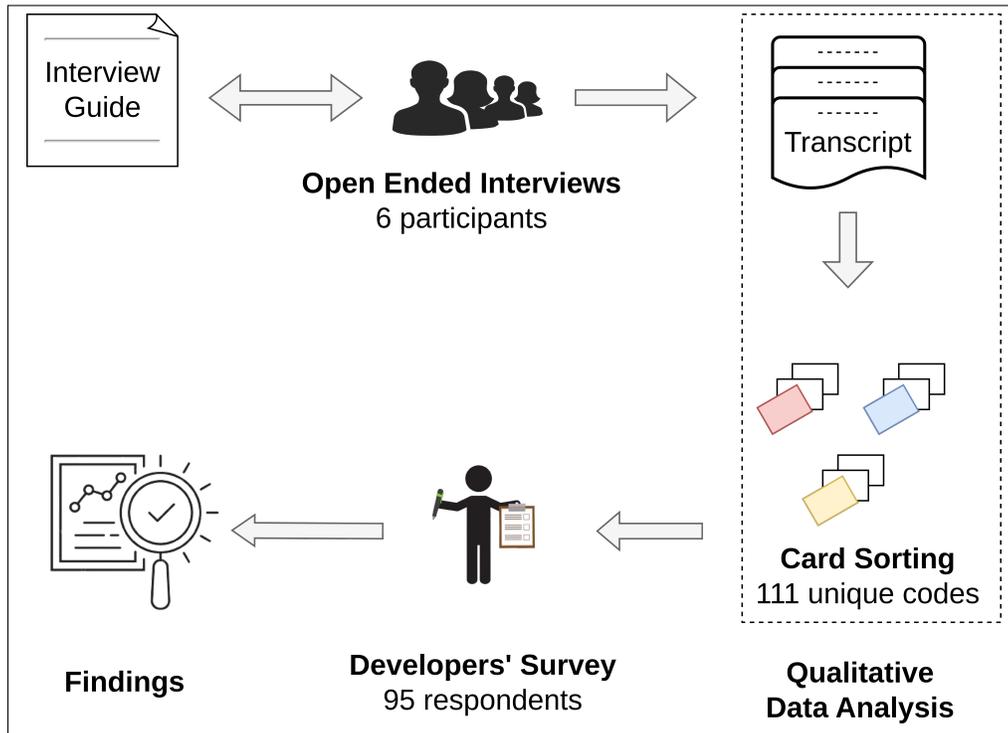


Figure 4.1: Research methodology.

ologies to comprehensively understand developers’ perspectives and their expectations of bug localization tools. The data collection process includes two primary sources: (1) in-depth interviews with six senior software practitioners and (2) survey responses from 95 software practitioners. To ensure transparency and facilitate further research, the interview transcripts, survey questionnaire, and the corresponding responses are publicly accessible¹.

4.2.1 Interview guide

Before these interviews, we create an interview guide covering broad topics and questions. After each interview, we refine this guide, clarifying any confusing questions to ensure that they are clearer and easier to understand for the next participants.

¹<https://t.ly/d9KtT>

4.2.2 Open Ended Interview

The following sections detail the protocol, participant selection, and data analysis methodologies employed in our open-ended interviews.

Protocol

We conduct six face-to-face interviews with senior software practitioners to understand their views on bug localization tools. The interview consists of three distinct parts. In the first part, we ask demographic questions about the interviewees, covering their age, gender, educational background, and experience in software development and leadership roles (e.g., senior software developer and software development manager). In the second part, we introduce the concept of bug localization to ensure that interviewees' understanding is consistent with our definition. Here, we pose open-ended questions to capture the interviewees' experiences with bug localization tools, allowing them to freely discuss their thoughts without bias. The questions include:

- What is your perception of a bug localization tool?
- Which feature of the tool do you appreciate the most?
- Which features or behavior of the tool do you dislike the most?

In the third part, we continue with open-ended questions to explore the interviewees' expectations of bug localization tools. We ask them:

- What do you consider the lowest acceptable performance for a bug localization tool?
- What type of output do you prefer from these tools?

At the end of each interview, we thank the interviewee and briefly outline our next steps.

Participant Selection

We select our interviewees from the authors' personal networks and professional networking sites like LinkedIn. Our recruitment criteria include at least ten years of professional experience and active involvement in software development. We continued the recruitment process until we reached data saturation. Following these criteria, we have six interviewees with diverse experiences. In this paper, we refer to these participants as P1 to P6.

Table 4.1 displays each interviewee's gender, the highest level of education, and years of experience.

Table 4.1: Gender, educational level, and experience of the six interviewees.

ID	Gender	Highest level of education	# of years of experience in SE
P1	Male	Masters	12
P2	Male	Bachelors	21
P3	Female	Masters	20
P4	Female	PhD	12
P5	Male	Bachelors	17
P6	Male	Bachelors	17

4.2.3 Qualitative Data Analysis

To gather insights from the interviews, we follow a structured process starting with transcribing the recorded interviews. We use the Otter auto-transcription tool² to transcribe the recordings after each interview. An author then reviews these transcriptions for accuracy by comparing them to the recorded videos and sends the transcription to the interviewees for verification.

After completing all interviews, we code the transcriptions using NVivo software³. This step generates 111 cards, with each interview yielding between 46 and 66 cards. Following prior studies [170, 316], we employ open card sorting to categorize these cards into common themes. After three sorting sessions, we identify 34 distinct themes in the interviews. The themes are available in the Appendix A.

4.2.4 Developers' Survey

To validate the themes identified in the interviews, we conduct a survey targeting a broader community of software developers. We adhered to Kitchenham and Pfleeger's guidelines [152] for personal opinion surveys and employed an anonymous survey to boost response rates [310]. As an expression of our appreciation, participants had the opportunity

²<https://otter.ai/>

³<http://portal.mynvivo.com/>

to enter a draw for one of 20 gift cards valued at 20 CAD each. We distribute the survey via our personal networks, the interviewees' connections, and professional networking sites, resulting in a total of 95 responses. The questions included in the survey are detailed in Table 4.2. To extract insights from these responses, we employ statistical analysis for the quantitative data and open card sorting for the open-ended questions.

Table 4.2: Survey Questions.

#	RQ	Question Text	#	RQ	Question Text
Q1	D	How old are you?	Q12	RQ1	What are the concerns in adopting a bug localization tool?
Q2	D	How do you identify your gender?	Q13	RQ2	What is the expected timeframe (in minute) for the tool to identify the files responsible for a bug?
Q3	D	What is your highest level of formal education?	Q14	RQ2	How many incorrect suggestions will you tolerate before a correct suggestion?
Q4	D	Did you receive your highest level of formal education in the field of computer science?	Q15	RQ4	Where do you want to integrate a bug localization tool?
Q5	D	How many years of experience do you have in the software engineering field?	Q16	RQ3	Which category of developers do you believe would benefit the most from the bug localization tool?
Q6	D	How many years of experience do you have in leading teams? (e.g., Senior developer, project manager etc.)	Q17	RQ3	For which software development role do you think the bug localization tool would be most advantageous?
Q7	RQ1	What is your current approach to bug localization?	Q18	RQ3	Which types of bugs do you believe the tool should prioritize and focus on?
Q8	RQ1	Have you ever used an LLM based tool like copilot or code whisperer for bug localization?	Q19	RQ3	Which error do you consider to be more severe than the other?
Q9	RQ1	On average, how long does it take you to locate bugs in source code files manually?	Q20	RQ4	What level of granularity would be the most effective for you when using the tool?
Q10	RQ1	What do you think will be the benefits of a bug localization tool?	Q21	RQ4	Please prioritize the following additional features of a bug localization tool in the order of their significance (most to least significant).
Q11	RQ1	What do you think will be the harm of using a bug localization tool?	Q22	RQ1	Would you ever use a bug localization tool?

4.3 Demographics

To provide a proper context for the results, this section describes the demographics of the interviewees and survey participants.

4.3.1 Interviewee Demographics

The interviewees were aged between 37 and 46 years, with a median age of 41. In terms of gender distribution, 66.6% were male, and 33.4% were female. Regarding educational qualifications, 50% held a bachelor's degree, 33.3% had a master's degree, and 16.6% possessed a Ph.D., all specifically in computer science. Their professional experience in software engineering ranged from 12 to 21 years, with a median of 16.5 years. Additionally, their leadership experience varied from 4 to 17 years, with a median tenure of 11 years in team leadership roles.

4.3.2 Survey Demographics

The age of survey respondents ranged from 24 to 70 years, with a median age of 34. The demographic breakdown showed that 80% were male, 17.78% female, and 2.22% non-binary or third gender. In terms of education, 49.45% reported a master's degree as their highest level of formal education, followed by 43.95% with a bachelor's degree and 6.95% with a Ph.D. A majority (81.81%) completed their highest level of education in computer science.

Survey participants also displayed a wide range of experience in the software engineering domain, spanning from one to 40 years, with a median experience of 9.5 years. The range of experience in leading teams was similarly broad, from one to 39 years, though the median experience in this area was shorter, at three years.

4.4 Results

The following subsections present the results of our survey, addressing the four research questions introduced in Section 4.1. Please note that respondents could select multiple options for some survey questions. Additionally, in the qualitative analysis, each open-ended response could correspond to multiple codes. As a result, the total percentages may exceed 100%.

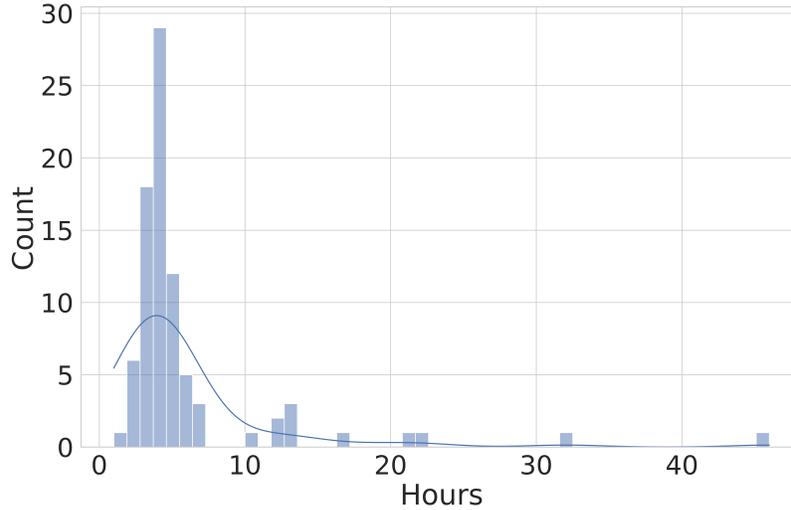


Figure 4.2: Time required to localize a bug.

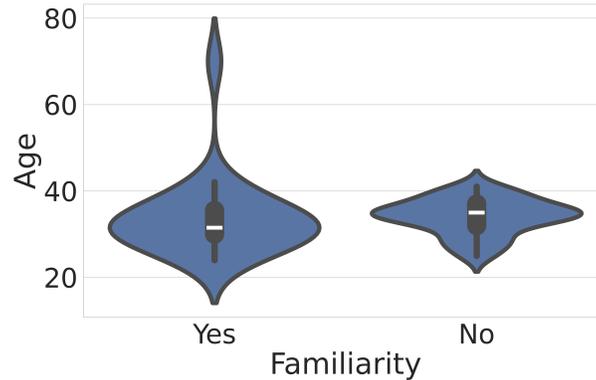
4.4.1 RQ1: What are developers’ current approaches and perspectives on bug localization tools?

Our first research question investigates how developers currently localize bugs and their views on using bug localization tools in software development.

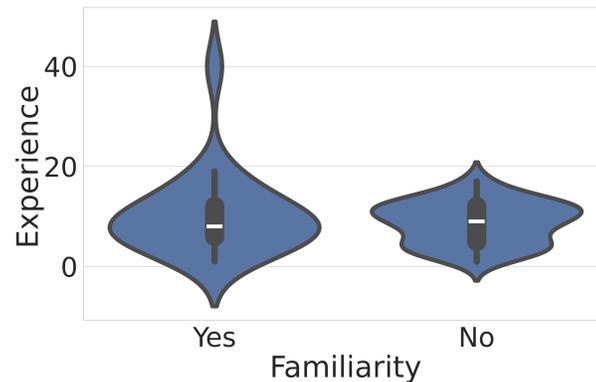
Current Approach: According to the survey, the most popular method for manual bug localization is reproducing the failing case, used by 62% of developers. Log analysis is the second most common method at 44%, followed by using text search tools such as ‘grep,’ employed by 30% of developers. Additionally, 29% of developers use assertions for bug localization, and 14% rely on tools like LLVM, GDB, and hexdecoder. A smaller percentage, 3% of developers, reported using other tools, including Sentry, a log-based error tracking tool, and Logstash, a large-scale log processor.

Developers also noted that using their current methods, they can manually localize a bug in one (minimum) to 46 hours (maximum), with a median time of four hours. Figure 4.2 shows the distribution of time required to localize a bug using these approaches.

Familiarity with code assistance tools: Recent developments in artificial intelligence [79] and their application in code assistance tools [171, 40, 88] have not yet significantly influenced developers’ practices. Only 27.5% of developers reported familiar with LLM-based tools. Figure 4.3 illustrates the distribution of age and experience in relation



(a) Familiarity vs. Age.



(b) Familiarity vs. Experience.

Figure 4.3: Respondents’ familiarity with LLM-based tools in relation to age and experience.

to familiarity with these tools. The figure indicates that younger respondents, both in terms of age and experience, tend to be more familiar with these modern tools. To compare familiarity across different categories, we applied two-tailed unpaired Mann-Whitney U tests [208] with a significance level of $\alpha = 0.05$. The tests showed significant differences for age vs. familiarity ($p = 0.042$) but not for experience vs. familiarity ($p = 0.43$). ***These results suggest that a developer’s age can significantly influence their familiarity with AI-based code assistance tools, such as automated bug localizers.***

Benefits and harms of using automated bug localization tools: Regarding the



Figure 4.4: Perceived harms of bug localization tools by experience.

benefits of using a bug localization tool, the majority of respondents (72%) believe that the primary advantage would be an increase in productivity. Additionally, 41% of respondents feel that these tools would be helpful in training new developers. A small percentage (1%) mentioned other benefits, such as automating manual steps in bug localization.

On the potential harms of using a bug localization tool, most respondents (67%) perceive that the greatest risk is developers adopting shortsighted solutions due to a limited understanding of the problem. Furthermore, 42% of respondents are concerned that reliance on these tools might hinder the development of their skills. In this context, P2 mentioned,

“... the first few years, whenever you’re part of a new team, the best way to engage a developer initially is to give them a couple of bugs. They solve the bugs, and walk through the code, and then they get to learn about the code base and different functionalities.” [# P2]

Furthermore, 22% of developers anticipate that these tools could negatively impact the job market for developers. A small fraction (1%) of respondents noted that blindly trusting these tools without verifying edge cases or considering downstream use cases can lead to long-term consequences. P2 expressed a similar concern.

“... developer will not be aware, which can be caused in some other area. So, you

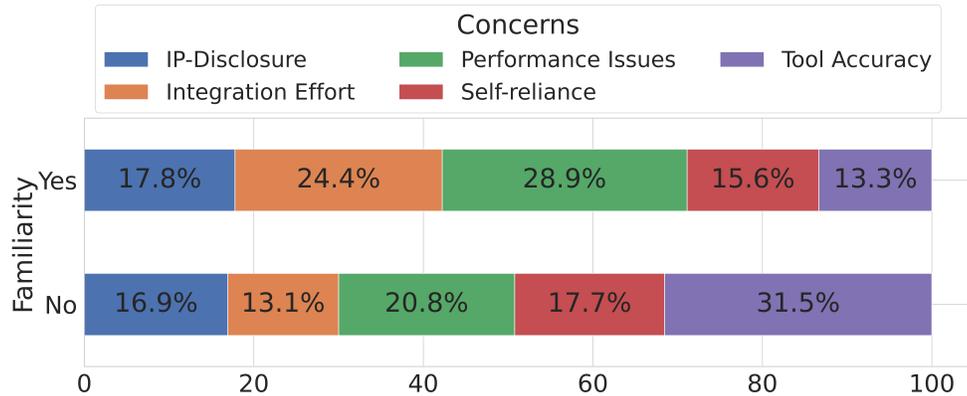


Figure 4.5: Concerns of adopting bug localization tools by familiarity with code assistance tools.

know, there is a possibility of more of those incidents, so probably they will fix this bug pretty quickly, but they are likely to introduce some other bugs” [# P2]

To understand how developers of varying experience levels perceive the harms of using a bug localization tool, we present the data in Figure 4.4. The figure indicates that younger developers are more concerned with the threat posed by automated tools to the job market, whereas senior developers are more focused on shortsighted solutions and the potential underdevelopment of developers’ skills.

Concerns regarding adopting bug localization tools: The primary concern about adopting bug localization tools relates to their accuracy, with 51% of respondents uncertain about the correctness of outputs. This specific concern was also observed among interviewees, as illustrated by P2’s comment:

“I would use it, as long as I find it to be reliable.” [# P2]

The next significant concern, shared by 43% of respondents, is the tool’s performance, particularly the time taken to deliver results. Additionally, 30% of respondents worry about the potential disclosure of intellectual property (IP). This issue was also a major concern among the interviewees, with half of them mentioning it. As P3 said,

“[I will use it] as long as it is not like consulting a server outside and like throwing information out to somebody else.” [# P3]

Another 30% believe that experienced developers may not use these tools due to confidence in their own abilities. P4 also echoed this sentiment.

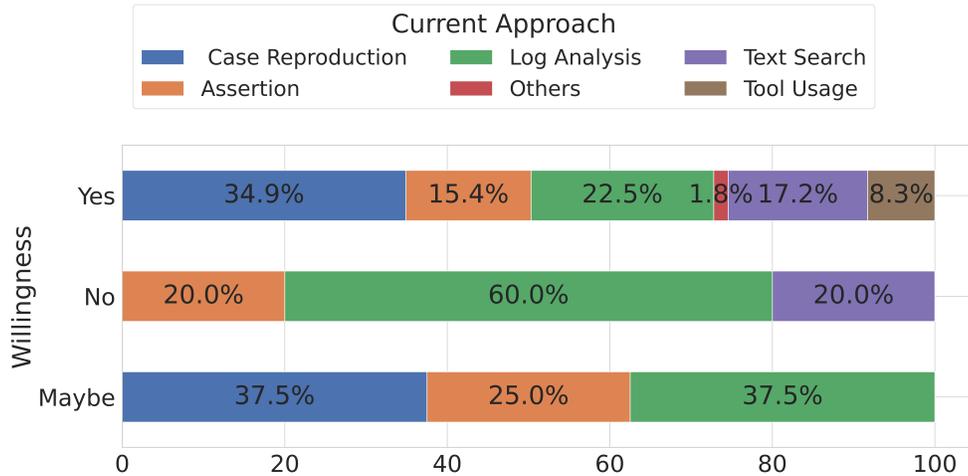


Figure 4.6: Current practices vs willingness to adopt.

“... giving them [senior developers] a tool sometimes makes them feel like, probably, other people do not have enough confidence in their abilities to track bugs.” [# P4]

Additionally, 29% of respondents are concerned about the effort required to integrate these tools into the existing software development pipeline.

Another concern is that these tools may perform well when the codebase adheres to best practices and conventions but poorly when those practices are lacking. However, in a well-maintained codebase, the need for such tools is minimal. Developers would benefit more if these tools delivered good results even in poorly maintained codebases. On this point, P6 mentioned:

“... a well-maintained codebase may not need such a [bug localization] tool. They may have been able to localize the bug in a viable time without the help of any tool. On the other hand, the tool may not work properly in a poorly maintained codebase.” [# P6]

To determine if familiarity with coding assistance tools leads to different concerns, we present the data in Figure 4.5. This figure shows that among developers who are unfamiliar with these tools, the most common concern is their accuracy. In contrast, developers who are familiar with the tools are mostly concerned about performance issues (e.g., latency). ***This observation suggests that while AI-based assistance tools (e.g., bug localizers) may be accurate enough, their performance is not yet sufficient to convince developers.***

Willingness to adopt: The majority (76%) of respondents indicated they are willing

to use an automated bug localizer. However, 5% are uncertain about using such a tool, and 4% are not interested.

We also investigated the willingness to adopt these tools in conjunction with current developer practices, as illustrated in Figure 4.6. The figure shows that developers inclined to adopt a bug localization tool typically rely on the manual and time-consuming process of reproducing cases to localize bugs. ***This observation highlights a significant pain point for developers and suggests a potential use case for bug localizers. By automating the process of reproducing failing cases, bug localizers can enhance developers' productivity and increase adoption rates.*** Among developers unwilling to adopt the tool, the most common practice is using logs to identify bugs, often with built-in search tools like 'grep' in file editors or operating systems. The availability and familiarity of these tools likely contribute to their lack of interest in using a bug localizer.

Takeaway 1: Developers mainly rely on reproducing failing cases and log analysis for bug localization. Despite the availability of advanced AI tools, only 27.5% of developers are familiar with them. Their primary concerns about adopting these tools are reliability, output quality, and potential IP leakage.

4.4.2 RQ2: What are developers' expectations from automated bug localization tools?

In this research question, we investigate the performance expectations of developers from bug localization tools.

Expected Timeframe for Bug Localization Tools: We asked respondents about the expected timeframe for a bug localization tool to deliver results, particularly considering its use in an interactive setting. The responses indicated an expected range of one to 59 minutes, with a median of eight minutes being deemed acceptable. Interviewees echoed similar expectations, emphasizing the importance of timely results for usability. As one interviewee noted,

“One hour is a pretty decent time to wait for. Yeah, like a day to fix anything on our side is a pretty big time to just pay for.” [# P4]

We calculate the Pearson correlation [86] between the manual time developers take to localize bugs and the time they expect an automatic bug localizer to deliver results. The correlation is 0.76 ($p < 0.001$). ***This strong positive correlation suggests that***

developers expect an automatic bug localizer to perform at a similar level to human efforts.

Expected results in TopK: Bug localization tools generate a ranked list of source code files likely responsible for a given bug report. During interviews and surveys, we presented a scenario with an example and asked respondents about the maximum acceptable rank for a buggy file (Top-K). The results revealed that respondents expect buggy files to appear between the first and 20th positions on the ranked list, with a median rank of fifth. Interviewees showed a similar preference, with 66% agreeing they would check up to the fifth rank for the buggy files, ideally between the first and the 10th positions.

Takeaway 2: Developers expect bug localization tools to deliver results within eight minutes and rank the correct files within the top five positions. These performance expectations closely align with the time and accuracy of manual efforts.

4.4.3 RQ3: Who are the primary users of bug localization tools, and what are the tools' focus areas?

In this research question, we examine the user base of bug localization tools and identify the types of problems these tools should prioritize.

Users of bug localization tools: Bug localization tools can be used by both software quality assurance (QA) engineers and software developers, depending on the hierarchy and domain of the work. However, 77% of respondents believe these tools are most beneficial for developers, while 26% think they are most useful for QA engineers. Interviewees have mixed views on the potential users: 50% believe the tools are primarily useful for developers, whereas another 33.3%, including interviewee P6, believe they will benefit most the QA engineers,

“A lot of people would rather prefer the QA engineer [to] file the bug report with all the information. So the developer does not spend more time.” [# P6]

Only one interviewee believes that both roles could benefit from a bug localization tool, depending on the organizational hierarchy.

Experience level of users: Survey responses indicate that the majority (77%) believe new or junior developers will benefit the most from these tools, while 43% think experienced or senior developers will gain significant advantages. Interviewees share a similar belief, as noted by P4.

“... also for fresh developers who have less experience in programming or software engineering, they haven’t built that muscle memory on where to look for when a bug comes in.” [# P4]

Severity of mistakes: Concerning errors made by bug localization tools, the majority of respondents (46%) believe that marking a non-buggy file as buggy (false positive) and marking a buggy file as non-buggy (false negative) are equally severe. However, 45.78% of respondents consider false negatives to be more severe, while only 7.22% regard false positives as more severe. Interviewees also perceive false negatives as more critical, noting that such errors can undermine the trustworthiness of the tool; as P1 said,

“I think false negatives are much worse than false positives. To make you trust the tool, being very sure of the negatives is quite important.” [# P1]

Focus areas of the tool: Respondents express a preference for bug localization tools to prioritize complex bugs. Specifically, 54.11% of respondents believe the tool should focus on complex bugs, while 45.44% think it should target easier bugs. Interviewees also had mixed opinions on this matter: some argued that the tool should address easier bugs due to their higher frequency, while others contended that it should concentrate on complex bugs because they require more time to localize.

Takeaway 3: Bug localization tools are considered to be most beneficial for developers, particularly junior ones, due to their limited experience in identifying bugs. The tools should prioritize complex bugs, as these are more time-consuming to localize manually.

4.4.4 RQ4: What features do developers value in automated bug localization tools?

In this research question, we explore the additional features developers expect in bug localization tools.

Integration point: Bug localization tools can function either as assistants to developers or as automated tools with different integration points. These points can range from the developer’s machine (e.g., Integrated Development Environments (IDE)) to Continuous Integration/Continuous Deployment (CI/CD) pipelines. The majority of respondents (65%) prefer integrating bug localization tools into their CI/CD pipeline, while 35% favor integration with the IDE. Interviewees also had mixed opinions, with some supporting CI/CD integration and others advocating for IDE integration.

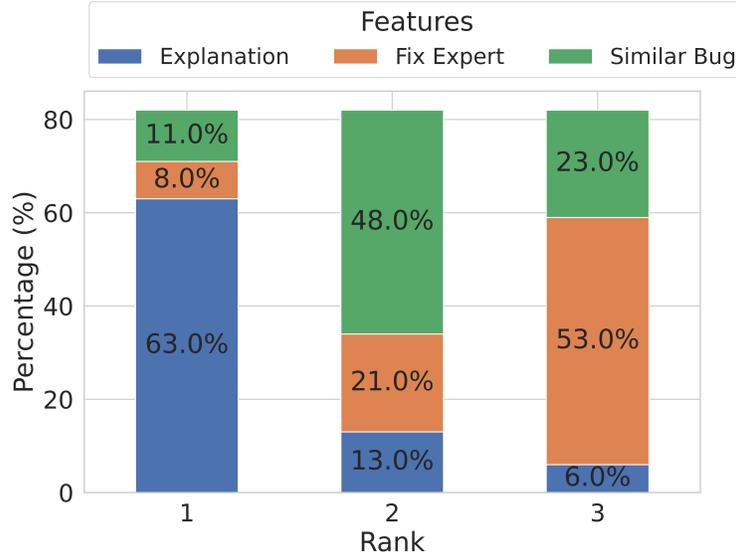


Figure 4.7: Additional features in bug localization tools.

Granularity of suggestion: Bug localization tools are available in various granularities, such as line-level, method-level, and file-level localization. Respondents indicated that a line-level bug localizer would be the most beneficial, with 63.41% supporting this preference. Following this, 34.1% of respondents believed a method-level bug localizer would be most useful. Only 2.4% of respondents preferred a file-level bug localizer. *However, most existing bug localization tools [347, 188, 53, 102, 374] primarily offer file-level granularity. This mismatch between user preferences and the solutions provided may be one reason for the low adoption rate of bug localization tools.*

Expected additional features: Interviewees indicated that their current bug localization tools lack certain features, such as transparency through explanation. In this regard, P3 mentioned,

“if you give me two files, and then info [explanation in this case]. How you arrived at it would help.” [# P3]

Additionally, interviewees highlighted other desired features, such as suggestions for similar bugs that have been previously fixed and recommendations for the best team member to address a bug.

Respondents were asked to rank these additional features based on their perceived benefits (a lower rank indicates higher benefit). Figure 4.7 shows that the majority of respondents (63%) believe explanations would be the most beneficial feature. Furthermore, 48% think the suggestion of similar bugs would be the second most beneficial, while 21% consider the recommendation of an expert team member to fix a bug as the second most beneficial. Finally, 53% of respondents ranked the suggestion of a team member as the third most beneficial feature. One reason developers may not find expert suggestions highly beneficial is because of the potential unavailability of team members.

Takeaway 4: Developers expect bug localization tools to be integrated into CI/CD pipelines (65%) and prefer line-level suggestions (52%) from the tool. Furthermore, as an additional feature, they prefer the transparency of the output by generating an explanation.

4.5 Discussion

Several studies have surveyed developers to understand their perspectives on defect prediction [316], fault localization [158], and the utilization of AI tools [288]. However, these surveys were conducted more than five years ago. Since then, recent advancements in artificial intelligence [79] have introduced a wide array of AI tools [88, 171] for developers, potentially altering their perspectives. Therefore, in this section, we compare the results of our survey with prior findings to observe any changes.

Kochhar et al. [158] conducted a survey involving 403 respondents to understand their perspectives on fault localization tools, while Wan et al. [316] surveyed 395 respondents to examine developers' perceptions of defect prediction. For our comparison, we utilize the publicly available survey responses from both studies^{4,5}.

4.5.1 Timeframe of results

In the survey by Kochhar et al., more than 50% of practitioners expected fault localization tools to deliver results in less than a minute. In contrast, our findings indicate that developers now have a broader tolerance, expecting the timeframe to range from one to

⁴<https://github.com/smysis/automated-debugging>

⁵https://github.com/zhiyuan-wan/defect_prediction_survey

fifty-nine minutes, with a median of eight minutes. We tested this observation using the Mann-Whitney U test and found that the hypothesis of increased tolerance levels is statistically significant ($p < 0.001$). ***This shift suggests a growing recognition of the increasing complexity of modern software systems [94, 11], where more detailed analysis might take longer.*** Nonetheless, a quick turnaround remains crucial for adopting bug localization tools in the software development lifecycle.

4.5.2 Expected results in TopK

Kochhar et al. also observed that over 80% of practitioners expected buggy files to be within the top five positions of the ranked list. The observation underscores the need for highly accurate tools to reduce manual bug localization efforts. In our findings, developers preferred buggy files to appear between the 1st and 20th positions, with a median preference for the 5th position. Using the Mann-Whitney U test, we found this observation to be statistically significant ($p < 0.001$). ***This indicates a slightly broader tolerance for the top ranks while still emphasizing the need for high accuracy.***

4.5.3 Granularity of suggestion

Wan et al.’s survey indicated that developers prefer feature-level granularity the most. In contrast, Kochhar et al. found that respondents preferred method and statement-level granularity almost equally, with file-level granularity following behind. Our findings, however, show that respondents predominantly prefer line-level granularity. ***This shift towards finer granularity underscores the increasing complexity [11] and the size of modern codebases [270], where more detailed localization is required to reduce debugging time effectively. Additionally, this evolution highlights a growing demand for precision in bug localization tools.***

4.5.4 Willingness to adopt

In Kochhar et al.’s survey, nearly all respondents were willing to adopt a fault localization tool if it met criteria such as trustworthiness, scalability, and efficiency. Unwillingness was mainly due to disbelief in the tool’s potential success and the need for more information. Similarly, Wan et al. found that the primary reason for not adopting defect prediction tools was disbelief in their efficacy. Another prominent reason for not adopting bug localization tools is the perceived lack of need; a theme echoed in a recent survey on AI tools [288]. Our

findings show that 76% of developers are willing to use an automated bug localization tool, with concerns focusing on accuracy, performance, and integration effort. This suggests a significant market potential for improved bug localization tools, provided they meet performance and reliability standards. *The recurring “lack of need” theme implies that developers may not see current tools as solving critical problems or may be unconvinced of their benefits. This underscores the importance of not only creating robust tools but also effectively communicating their value to the developer community.*

Takeaway 5: Developers’ expectations have evolved to tolerate more relaxed performance standards for bug localization tools. Despite advancements, adoption remains low. However, there is a strong willingness to adopt these tools if they meet the necessary performance and reliability criteria.

4.6 Implication

In this study, we focus on understanding developers’ perceptions and expectations of using bug localization tools. The results of our survey highlight areas for improvement and suggest future research directions from the developers’ perspective. In this section, we discuss the implications of these findings.

4.6.1 What Needs Our Focus Now?

Developer Productivity and Training: In this study, we identify that the primary benefit of bug localization tools is their potential to increase developers’ productivity significantly. These tools reduce the time spent on debugging, thereby allowing developers to concentrate on writing new features and enhancing overall software quality [319]. Furthermore, 41% of our respondents believe bug localization tools can serve as effective training aids for junior developers, facilitating their understanding of the codebase and improving their debugging skills efficiently. This dual role of enhancing productivity and providing training support can be leveraged to promote the broader adoption of bug localization tools.

Accuracy and Performance Concerns: A major barrier to the adoption of bug localization tools is the concern regarding their accuracy and performance. The majority of the respondents (51%) were uncertain about the correctness of the tools. Enhancing

the precision and reliability of these tools is, therefore, essential. Future research should prioritize refining algorithms to improve accuracy and ensure that the tools deliver reliable results promptly. Addressing these concerns is vital for increasing developer confidence and encouraging broader adoption.

Integration with Existing Workflows: The success of software tools largely hinges on their integration points and usage patterns [329]. However, no prior studies have specifically examined the integration points of bug localization tools. In this study, we found that approximately 29% of respondents expressed concerns about the integration effort required to incorporate these tools into their existing workflows. A potential solution to this issue is to embed the tools within the development pipeline. Our findings indicate a strong preference for integrating bug localization tools into CI/CD pipelines (65% of respondents) and IDEs. Such integration could simplify workflows, reduce the learning curve, and promote tool adoption by enabling developers to utilize them in familiar environments. Future development efforts should prioritize seamless integration to enhance both usability and effectiveness.

Transparency and Customization: Transparency in bug localization tools is crucial for building trust among developers. According to our survey, 67% of respondents believe these tools can lead to shortsighted solutions, and 42% think they may hinder skill development. Generating explanations for the tool’s suggestions can mitigate these concerns, preventing shortsighted fixes and serving as a training aid for newer developers to improve their skills. Additionally, tools that clearly explain their processes and how they reach conclusions are more likely to gain trust and adoption [133, 116]. Furthermore, offering customization options—such as adjusting the granularity of bug localization (line, method, or file level)—enables developers to tailor the tools to their specific needs, enhancing both utility and user satisfaction.

4.6.2 What Needs to Change in the Future?

Enhancing Tool Accuracy and Contextual Understanding: Addressing the issue of accuracy and contextual understanding in bug localization tools is crucial [133]. Our findings reveal that while AI-powered tools, such as bug localizers, may achieve a reasonable level of accuracy, their current performance is still not convincing enough for developers. According to our survey, 4% of respondents expressed no interest in using bug localization tools. For these tools to be effective, they must be capable of understanding the broader context of the codebase, including project-specific documentation and internal knowledge bases. Enhancing the tools’ capacity to handle large volumes of information and deliver contextually relevant results will be a significant advancement. Previous studies have

shown that, in addition to the codebase, other sources such as commit messages, communication channels, design documents, and documentation can provide essential information for bug localization [116, 141, 54]. Therefore, besides improving the underlying algorithms, researchers should ensure that tools can access and process all relevant information for accurate bug localization.

Addressing Legal and Ethical Considerations: Legal and ethical considerations, such as data privacy and security, are crucial [119, 361, 250]. In our survey, 30% of the respondents are concerned about the security of their code and the potential for sensitive information to be exposed. Developing local models that can operate on user devices without sending data over the internet could alleviate these concerns. Additionally, ensuring compliance with company policies and non-disclosure agreements is crucial for the acceptance of these tools in practical environments [179].

Providing extensible information to Developers and Promoting Tool Benefits: Another important area is providing information to developers about the capabilities and limitations of bug localization tools. A recurring theme observed in both this study and previous research is the perceived “lack of need” for bug localization tools. This perception likely stems from an incomplete understanding of the tools’ potential benefits or how to integrate them effectively into existing workflows. Providing comprehensive training and resources can help bridge this knowledge gap and promote wider adoption. Additionally, clear communication about the tools’ advantages and improvements can foster trust among developers and encourage their use [178, 112].

4.7 Threats to Validity

Internal Validity Threats. The primary threats to validity are related to sample selection. As in previous studies [158], we recruited participants from professional networking sites. This may result in a sample that does not fully represent all software developers, potentially impacting the generalizability of opinions on bug localization tools. However, we mitigated this by including interviewees with at least ten years of experience. Their long-term experience will be able to cover developers’ perspectives from a variety of viewpoints. Similarities in responses from both interviews and surveys further validate our findings.

Construct Validity Threats. Our results may be influenced by non-response bias, where the views of participants differ from those who chose not to respond. However, the 95 responses we collected provide meaningful insights. Another potential threat is if our methods do not accurately capture key constructs like perceptions and expectations

of bug localization tools. To mitigate this, we followed prior studies [316, 41] and refined our interview guide after each session. We also designed our survey questionnaire based on established frameworks [91]. By combining qualitative interviews with quantitative surveys, we ensured a well-rounded understanding of practitioners' views. The convergence of results from both methods further supports the validity of our conclusions.

External Validity Threats. We interviewed six participants and surveyed 95 software engineering professionals. Although the sample size is small, it yielded valuable insights. Additionally, all interviewees had at least ten years of experience, which helps mitigate concerns about the limited sample size by ensuring depth and detail in their responses.

4.8 Conclusion

In this study, we systematically investigate developers' perceptions and expectations of bug localization tools through interviews and surveys. Our findings reveal that, while most developers are willing to adopt bug localization tools, they rely on manual methods for bug localization. The primary expectations of developers include rapid and accurate bug localization, particularly for complex issues. However, developers also expressed concerns about the potential negative impact of bug localization tools on developers' skill development, as well as the risk of intellectual property leaks.

Furthermore, insights from this study underscore the importance of improving developer training to bridge the gap between research advancements and real-world usage. Specifically, developers emphasized the need for transparency in tool operations and customization options to address specific project requirements. Therefore, future research should focus on addressing these challenges and promoting seamless integration to enhance the effectiveness and acceptance of bug localization tools in software development.

4.9 Chapter Summary

In this chapter, we explore developers' expectations for bug localization tools, focusing on key areas of interest and desired features. Our findings indicate that while developers are interested in utilizing these tools, they have significant concerns about their performance and accuracy. Additionally, developers expressed doubts about the effectiveness of these tools in non-standard cases, such as when there is no labeled data or when the data is noisy. The next chapter will delve into the challenges of training these tools, examining the design choices and their impact on model performance.

Chapter 5

Exploring Design Choices in Multi-Modal Transformer-Based Embedding for Bug Localization

Note. An earlier version of the work in this chapter appears in the Proceedings of the IEEE/ACM International Workshop on Natural Language-Based Software Engineering (NLBSE 2024).

5.1 Introduction

DL-based bug localization techniques have achieved comparatively higher performance than other techniques in recent years [348, 344, 165, 53, 148, 317, 124]. Most DL-based techniques use similarity measurement to identify buggy files. However, the bug reports are in Natural Language (NL), and source codes are in Programming Language (PL). Thus, there is a semantic and lexical gap between them. Previous studies used well-known NLP techniques such as Word2Vec or FastText to project the source code and the bug report in the same vector space. This vector representation was successful in bridging the aforementioned gap to some extent. However, one of the drawbacks of such techniques is that they do not consider the context in generating vector representation of a source code file or a bug report. In recent times state-of-the-art large pre-trained models such as BERT [66], RoBERTa [197] are used in Natural Language Processing (NLP) for capturing the semantic and contextual representation of text as they consider the context

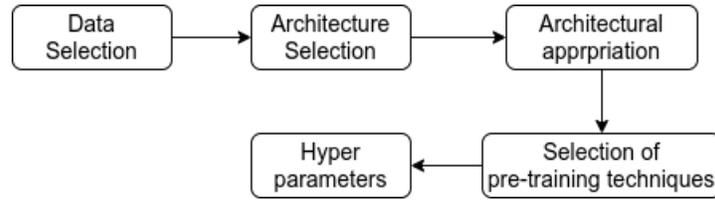


Figure 5.1: Steps of training a transformer.

in generating vector representation. The advantage of using those models is that they are more context-sensitive than before.

However, there are many design choices in creating a transformer-based embedding, such as model architecture, pre-training strategies, and data sources for training the embedding model. Figure 5.1 presents the steps of training a transformer. These design choices can impact the quality of the embeddings generated by an embedding model. In the first step of training transformers, we have to identify the appropriate data for pre-training. In the second and third steps, we have to select the proper architecture of the transformers and make the appropriate changes to fit the use case (e.g., sequence size, attention head). In the third step, we have to identify the pre-training technique. So far, there are different pre-training techniques Masked Language Model (MLM), Question-Answering (QA), Next Sentence Prediction (NSP). However, the performance of the pre-training technique is specific to the use case [197]. The final step is the selection of hyper-parameters such as learning rate and batch size. Ontañón et al. [241] have focussed on the second step and explored the impact of different types of positional encodings, different types of decoders, and weight sharing. Zhu et al. [387] have also focussed on identifying the impact of design choices in the second step. In their study, they used reinforcement learning to identify the best design choices, for example, the number of layers in the transformer model and the type of activation function. The performance of the bug localization model is dependent on the embedding quality. There are too many types of architectures and training methods for transformer-based models. Thus, knowing which architecture and training methods produce the best quality embedding for bug localization task is important.

With the increment of the complexity of the modern bug-localization model, the resource (GPU size, training time) requirement is also increasing. Thus, a project-specific bug localization model is a less popular choice than a cross-project bug localization model, which can localize bugs in different projects. However, the performance of cross-project bug localization models is comparatively lower than project-specific bug localization models. Thus, it is required to quantify the impact of project-specific data on bug localization models' performance to understand the trade-off and make better design decisions.

This study aims to understand the impact of three design choices on embedding models' performance and the generalization capability of those embedding models. The choices are the use of domain-specific data, pre-training methodology, and the sequence length of the embedding. Though the design space for a language model (embedding) is not limited, we can say that the type of training data, model architecture, and training strategies are the primary design choices. Prior studies have already identified some other design choices such as learning rate, weight sharing [241], and other hyperparameters [132, 259]. Thus, in this study, we intend to identify the impact of three design choices by answering the following research questions.

RQ1. Do we need data familiarity to apply the embeddings?

In this research question, we will test whether project-specific data is needed for the embedding models or not. In the NLP domain, we have observed the use of transfer learning. However, previous studies in the domain of bug localization used project-specific embeddings [187] for their models. Using project-specific embeddings may not be a viable approach in commercial settings. Thus we need to verify whether data familiarity is required in using transformer-based embedding models. We trained embedding models on two datasets to answer the question. We found that pre-trained embedding models using project-specific datasets perform better in bug localization tasks than those that are not pre-trained using those data.

RQ2. Do pre-training methodologies impact embedding models performance?

We will analyze whether pre-training impacts the performance of the embedding model and which pre-training technique is better for bug localization in this research question. We have seen different pre-training techniques such as MLM, QA, NSP, etc. However, pre-training techniques have domain-specific use. Peter et al. [252] found that feature extraction depends on the similarity of the pre-training and target tasks. Liu et al. [197] found that the individual sentence NSP (Next Sentence Prediction) task hurts the performance of the transformer model in a question-answer task. No other studies before us have verified the effectiveness of the pre-training techniques in the domain of bug localization which uses both programming language and natural language. Thus we need to know the impact of pre-training techniques and identify the best technique for bug localization tasks. For this, we pre-trained the embedding models using several pre-training strategies. We found that certain pre-training methodologies, such as ELECTRA (Efficiently Learning an Encoder that Classifies Token Replacements Accurately) create better embedding models than others.

RQ3. Do transformers with longer maximum input sequence lengths perform well compared to the models with shorter maximum input sequences?

We test whether the length of the input sequences of an embedding model impacts the model’s performance in the bug localization task. Source codes are typically long (~ 2000 tokens), whereas transformers typically support 512 tokens. Typical transformer architectures can not be extended to higher sequence lengths as the action is computationally expensive ($O(n^4)$, where n is sequence length [26]). Thus we need to understand the performance gain over using a long input sequence transformer in bug localization tasks. We trained embedding models with different lengths and compared their performance in the bug-localization task. In most experimental settings we conducted, the embedding model with longer input sequences outperformed the embedding model with shorter input sequences.

We used two datasets to train the embedding and bug localization models. We collected the first one by mining the bugs/issues from the Apache project and Github. The other one has been offered by Ye et al. [363]. We have evaluated 14 transformer-based embedding models on the bug-localization task using those two datasets.

Overall, our study finds a significant difference between the in-project and cross-project performance of the embedding models. We also found that pre-training methodologies impact embedding models’ performance. Our research found that in most cases, adversarial techniques like ELECTRA pre-trained models performed better than others. This study trained embedding models using bug report-source code pairs. Compared to the documentation-source code pair-trained model, we found bug report-source code-trained embedding models performed better. The dataset and code used in this study are publicly available here ¹.

5.2 Dataset

We have used two different datasets for this study. The first one consists of mined projects of the Apache project and public Github repositories. The following steps are followed to extract data from JIRA (bug report repository of Apache projects).

1. **Candidate project selection:** In this step, we listed all Apache projects except those included in the dataset prepared by Ye et al. [363] or Bugzbook [9]. The reason for the exclusion is that we intend to use those datasets to test the proposed embedding’s effectiveness in the future.

¹<https://zenodo.org/record/6760333>

2. **Extraction of bug reports:** After creating the candidate project list, we have extracted all the fixed bug reports. For that purpose, we have used Jira Query Language (JQL). However, not all Apache projects selected in the previous step have enough bug reports. We have filtered out the project if that has less than ten bug reports in JIRA. The list of the projects is available in the online Appendix ¹.
3. **Extraction of source code:** In this step, each bug report in JIRA, identifiable by a unique id, was linked to its corresponding pull request in Git/Github using the bug id and a regular expression heuristic, as done in prior studies [294]. Following this, the commit hash id (SHA) of the fix, along with the pre- and post-fix file versions for each change in the commit, were extracted.

For public repositories of Github, we first listed all repositories of Java language sorted by the number of stars in descending order. Like the steps followed for extracting JIRA data, we also excluded some of the repositories in this step. After that, we followed the before-mentioned steps to extract bug reports and source codes from those repositories.

For this study, we have used only the Java language source code files and bug reports. After filtering by language, we have 7,970 bug reports from 21 projects. From now on, we will refer to this dataset as Bug Localization Dataset (BLDS). This dataset has been used only for pre-training the language model.

The second dataset we used has been offered by Ye et al. [363, 362]. This dataset contains 22.7K bug reports from Six popular Apache projects (AspectJ, Birt, Eclipse UI, JDT, SWT, and Tomcat) and associated source codes. From now on, we will refer to this dataset as the Benchmark Dataset for Bug Localization (Bench-BLDS). We select this dataset as prior study [145] showed that this dataset contains the lowest number of false positive or negative cases among other datasets [173, 223] in the literature. We have used Bench-BLDS only for training and testing the bug localization model. The Bench-BLDS dataset contains 12480 bug reports from six projects. The length distribution of the source code files of BLDS and Bench-BLDS, along with the description of the projects, is available in the online Appendix ¹.

5.3 Methodology

In this study, we created 12 different transformer-based embedding models. Previous studies have presented one multi-modal language model [83], and we have updated that language model by extending the maximum supported sequence length. Next, we evaluated

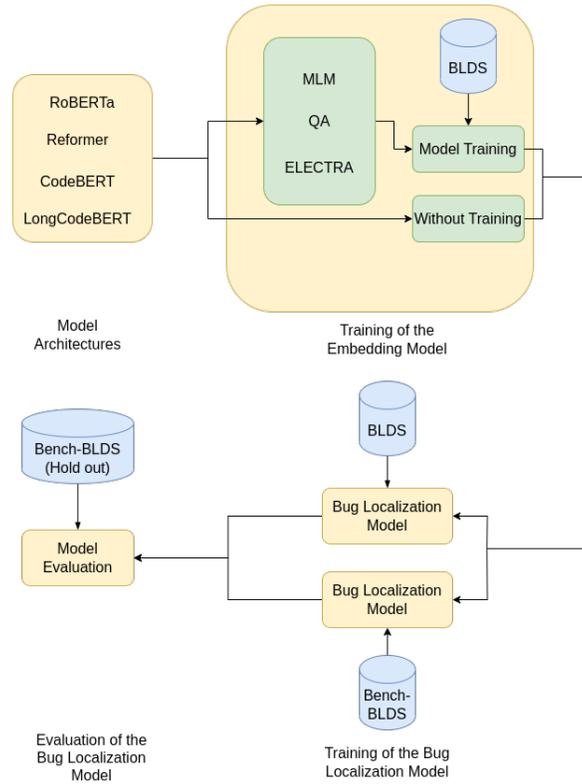


Figure 5.2: Multi-modal embedding training pipeline.

14 models (our twelve models along with one model from the previous study and the extended version of that model) in the bug localization task. The embedding models differ from each other in terms of the architecture and the pre-training methodology. The methodology of this study has been presented in Figure 5.2.

Model Architectures. The first step of our methodology is the selection of the architecture for the embedding model. We have used two different architectures, RoBERTa and Reformer. The reason for selecting these two embedding models is that the methodology for calculating attention is fundamentally different for these two architectures. RoBERTa is a popular architecture; based on this architecture, many domain-specific embedding models have been proposed in NLP. On the other hand, Reformer presented a new attention calculation method that reduces the complexity of attention calculation. The weights of the models are initialized randomly. Besides these two architectures, we have also used CodeBERT [83], which is a variant of the RoBERTa model trained on multimodal (NLP) data. As CodeBERT has already been trained, we have used the trained weights for

CodeBERT. Furthermore, we extended the CodeBERT model by increasing the maximum sequence length. In this process, we have followed the approach of Beltagy et al. [26] The extended model will have a new maximum sequence length while using the trained weights of CodeBERT. From now on, we will call this extended CodeBERT model LongCodeBERT. We have selected CodeBERT [83] as it is the state-of-the-art multi-modal embedding model for software engineering tasks. The reason behind using LongCodeBERT is it extends the maximum sequence length of the original CodeBERT embedding model while not changing the weights. After this step, we have four different architectures of the embedding model, two of which (CodeBERT, LongCodeBERT) have already been trained in a previous study.

Training of the Embedding Model. The second step is the pre-training of the embedding models. For pre-training the embedding models, we followed three methods: Masked Language Modeling (MLM) [66], ELECTRA [58], and QA [92], and used the BLDS dataset. After this step, we will have twelve different embedding models (four architecture trained using three different methods). Moreover, we also wanted to know the performance of the embedding models proposed in previous studies. Thus, we have included the CodeBERT and LongCodeBERT embedding models without further fine-tuning/pre-training steps, which increases the number of embedding models from twelve to fourteen (four architecture trained using three different methods along with two architectures without any further training).

Training of the Bug Localization Model. In this study, we referred to the language models (DL models that generate multi-modal data embedding) as *embedding models* and bug detection models (DL models that use embedding models to detect bugs) as *emphbug localization models*. In the third step, we have used the embeddings generated from the embedding models to train bug localization model. Following the practices of previous studies [125, 317, 123, 344], we have used a Convolutional Neural Network (CNN) based architecture for the bug localization model. This study aims not to find the best bug localization model but the best embedding training technique for a bug localization model. We have used the bug localization models as a performance measurement for the embedding models. The bug localization model consists of three convolutional layers followed by a perceptron layer. In training the bug-localization models, we have used two different datasets. The input to the bug localization models is a vector of the bug report and the source code corresponding to the bug report, concatenated together. The bug localization task has been aimed for file-level bug localization and has narrowed down to the task of a binary classification problem. The model is trained to learn whether the given pair of bug reports and code snippet is a match or not. All the layers/weights of the embedding model are unchanged (frozen) while training the bug-localization model. After this step, we will have two bug localization models for each of the fourteen embedding models, which means

we will have 28 bug localization models.

Evaluation of the Bug Localization Model. In the fourth step, we evaluated the bug localization models’ performance on a held-out test dataset created from the Bench-BLDS. The performance has been measured in terms of Mean Reciprocal Rank (MRR). We have mentioned before that the BLDS and Bench-BLDS datasets has no common projects. Thus, the model trained on the BLDS dataset and evaluated on the Bench-BLDS dataset can be considered a cross-project bug localization model. All the models are trained for ten epochs in our training setup with 32 samples per GPU separately. Later the performance is evaluated on the held out Bench-BLDS dataset.

We have used all combinations of model architectures, training techniques, and data sources to understand the impact and identify the best design choices. However, testing all combinations requires time. For example, to complete one row of Table 5.4 requires sixteen days of training in a single GPU machine (Nvidia V100 Volta 32G GPU). This translates to almost 2 months for the entire set of combinations reported in Table 1. Now, for any small tweak that we make, we need to rerun the whole experiment again, which would take anywhere between 4-56 days. Thus another contribution of this study is that future studies do not have to go through the same time-consuming process and test all combinations to find good design choices for embeddings.

5.4 Results

Table 5.2, 5.3 and 5.4 represents the average performance whether the bug localization model is trained on project-specific data or not, the average performance of each pre-training technique, and average performance of each architecture respectively. Each column with the project name (AspectJ, Birt, JDT, SWT, and Tomcat) represents the models’ performance in those respective projects. The *overall* column represents the overall performance of the model in these six projects. Table 5.2, 5.3 and 5.4 have been calculated from Table A.2 and Table A.3, which are available in the online appendix [17]. Tables A.2 and A.3 represent the performance of all the bug localization models in terms of MRR and MAP, respectively. Besides, we have also presented the performance of previous IR-based studies in Table 5.1. However, those studies have not used Birt, Eclipse UI, and Tomcat. Thus, the performance of the tools on those projects was not presented.

Table 5.1: Performance of the IR-based systems.

Study	AspectJ	JDT	SWT
MRR			
BugLocator [383]	0.38	0.26	0.50
BRTracer [339]	0.25	0.33	0.62
BLUiR [283]	0.41	0.38	0.59
AmaLgam [323]	0.33	0.33	0.62
MAP			
BugLocator	0.21	0.16	0.44
BRTracer	0.14	0.24	0.55
BLUiR	0.22	0.27	0.52
AmaLgam	0.20	0.24	0.55

Table 5.2: Average performance with varying data familiarity.

Training Data (CNN Model)	AspectJ	Birt	Eclipse	JDT	SWT	Tomcat	Overall
MRR							
BLDS	0.28	0.26	0.28	0.27	0.25	0.25	0.27
Bench-BLDS	0.37	0.32	0.37	0.37	0.35	0.30	0.35
MAP							
BLDS	0.19	0.18	0.20	0.18	0.17	0.16	0.17
Bench-BLDS	0.28	0.23	0.29	0.27	0.27	0.21	0.25

Table 5.3: Average performance in different pre-training techniques.

Embedding Training Strategy	AspectJ	Birt	Eclipse	JDT	SWT	Tomcat	Overall
MRR							
ELECTRA	0.37	0.28	0.34	0.35	0.33	0.30	0.33
MLM	0.36	0.29	0.34	0.37	0.31	0.28	0.33
MLM and QA	0.27	0.30	0.32	0.28	0.27	0.27	0.29
Without training	0.27	0.27	0.28	0.23	0.27	0.25	0.27
MAP							
ELECTRA	0.29	0.20	0.26	0.26	0.25	0.21	0.23
MLM	0.27	0.21	0.25	0.28	0.23	0.19	0.23
MLM and QA	0.19	0.22	0.24	0.19	0.19	0.18	0.19
Without training	0.18	0.18	0.20	0.14	0.19	0.16	0.17

5.4.1 RQ1. Do we need data familiarity to apply the embeddings?

In the natural language domain, pre-trained embedding models are used in various downstream tasks such as question-answering, sentiment detection, etc. To use pre-trained embedding models in downstream tasks, one needs to add a task-specific model (head) on top of the embedding models. From a language model, we only receive the vector representation of the text in the embedding space. However, in the context of source code, the use of libraries, comment style, and coding style are not the same in all projects. Thus, the relation between vector representation and bugginess may vary from project to project. This variation may pose a problem if we intend to use a bug localization model in another project without further training. The head is not familiar with the relation between the vector of the source code-bug report pair and the bugginess of a file. Thus, the performance may vary in a new project.

Let’s think about bug localization in commercial settings where a tool is used to localize bugs from hundreds of different projects of an organization. Project-specific training seems like a less viable approach. It will require a high amount of resources to maintain (train

and deploy) a specific head for each project. The scalability of a bug localization system depends on whether project-specific training is required or not. Past NLP research [117] seems to indicate that transformer-based models are good at predicting when they are not pre-trained on similar data. Thus, in this research question, we will test whether or not project-specific training is necessary for the heads.

Table 5.2 represents the average performance in two cases, whether or not the bug localization model is trained on project-specific data. From Table 5.2 we can observe that the models trained on the Bench-BLDS dataset performed better than the model trained on the BLDS dataset. To verify whether the observation is statistically significant, we conducted a pairwise Mann-Whitney test between the model trained on BLDS and Bench-BLDS. We found that the observation that Bench-BLDS trained models perform better is statistically significant ($p < 0.001$) for both MRR and MAP. The observation may point out that the embedding model used in the understanding of programming language is learning more project-specific features than the ones in the NLP domain. For example, in the text classification task using of ULMFit [117] without pre-training achieved a 5.63% error, or the pre-trained embedding model with project-specific data achieved a 5.00% error on the IMDB dataset. However, in this study, we have observed a maximum 76% drop in performance. From a high-level understanding, we can say that the heads trained on specific data representations struggle with generalization across projects, indicating a need for more adaptable architecture for heads.

Takeaway 6: Data familiarity has an impact on embedding models' performance. Training the embeddings with project-specific data (fine-tuning) can enhance the performance of the bug localization models.

5.4.2 RQ2. Do pre-training methodologies impact embedding models' performance?

In NLP, many pre-training methodologies are used to understand the language model. Methodologies like MLM, NSP, and dynamic MLM are widely used in NLP. However, we do not know how the pre-training methodologies contribute to understanding programming language. Though several pre-training methodologies are used for natural language processing, we do not have any pre-training methods for a specific SE task (such as bug localization).

Moreover, Liu et al. [197] have found that only some NSP pre-training is appropriate

Table 5.4: Average performance with varying model architecture.

Model Name	AspectJ	Birt	Eclipse	JDT	SWT	Tomcat	Overall
MRR							
CodeBERT	0.24	0.24	0.30	0.25	0.28	0.22	0.26
Long CodeBERT	0.40	0.32	0.36	0.35	0.29	0.29	0.34
Long RoBERTa	0.31	0.31	0.33	0.35	0.33	0.32	0.32
Reformer	0.35	0.28	0.33	0.34	0.32	0.29	0.32
MAP							
CodeBERT	0.16	0.16	0.21	0.16	0.19	0.13	0.16
Long CodeBERT	0.31	0.23	0.28	0.26	0.21	0.20	0.25
Long RoBERTa	0.22	0.23	0.24	0.25	0.24	0.23	0.22
Reformer	0.26	0.19	0.24	0.25	0.24	0.21	0.22

for the question-answering task in the NLP domain. Furthermore, individual sentence NSP hurts the performance of the model. However, we do not have any data about the impact of pre-training in bug localization tasks.

Thus it is essential to know which pre-training methodology works best for creating a joint (natural language, programming language) embedding space and how pre-training methodologies impact embedding models performance. Table 5.3 shows the average performance of the bug localization model where the embedding models are trained using different pre-training methodologies. We observed that overall, ELECTRA pre-trained embedding models produced a better result in the bug-localization task. To test the observation, we compared the performance of the models using the pairwise Mann-Whitney U test. The performance comparison between ELECTRA, QA methodologies was statistically significant for both MRR and MAP. For MRR, it was significant with $p = 0.013$ (Bonferroni corrected α was $0.05/3 = 0.016$) and for MAP it was significant with $p = 0.009$ (Bonferroni corrected α was $0.05/3 = 0.016$). However, the difference between ELECTRA and MLM ($p = 0.3$ for MRR and $p = 0.38$ for MAP) and MLM and QA ($p = 0.97$ for MRR and $p = 0.98$ for MAP) is not statistically significant. Though no specific pre-training technique performed better (statistically significant) than other techniques, ELECTRA pre-trained

Table 5.5: Average performance of the embeddings in six projects sorted by sequence length.

Model Name	Sequence Length	AspectJ	Birt	Eclipse	JDT	SWT	Tomcat	Overall
MRR								
CodeBERT	512	0.24	0.24	0.30	0.25	0.28	0.22	0.26
Long RoBERTa	1536	0.31	0.31	0.33	0.35	0.33	0.32	0.32
Reformer	2048	0.35	0.28	0.33	0.34	0.32	0.29	0.32
Long CodeBERT	4096	0.40	0.32	0.36	0.35	0.29	0.29	0.34
MAP								
CodeBERT	512	0.16	0.16	0.21	0.16	0.19	0.13	0.16
Long RoBERTa	1536	0.22	0.23	0.24	0.25	0.24	0.23	0.22
Reformer	2048	0.26	0.19	0.24	0.25	0.24	0.21	0.22
Long CodeBERT	4096	0.31	0.23	0.28	0.26	0.21	0.20	0.25

embedding models achieved the highest MRR and MAP in 48% of cases. In contrast, MLM and QA-trained embedding models achieved the highest MRR and MAP in 28% and 23% of cases, respectively. For CodeBERT and LongCodeBERT, we have embedding models offered by a previous study trained in a separate dataset. We have used those embedding models as it is (without training). However, the performance difference between embedding models without training and embedding models trained with ELECTRA ($p = 0.02$ for MRR and MAP) or QA ($p = 0.87$ for MRR and $p = 0.83$ for MAP) is not statistically significant. Only the difference between embedding models without training and models trained with MLM is statistically significant ($p = 0.0001$ for both MRR and MAP). One reason for the statistically insignificant difference can be the low number of data points for comparison. As ELECTRA pre-training is a discriminative pre-training approach, embedding models trained by this technique generate a more generalized representation. The reason for more generalization is that ELECTRA is defined over all input tokens, whereas the MLM task is defined over a small subset of tokens. Because of the generalizability, we believe the bug localization models that used ELECTRA trained representation performed better.

Takeaway 7: Pre-training has an impact on the embedding model’s performance. Generally, the ELECTRA pre-trained embedding models performed better than the other two pre-training techniques (MLM, QA) in bug localization tasks.

5.4.3 RQ3. Do transformers with longer maximum input sequence lengths perform well compared to the models with shorter maximum input sequences?

Before using transformers for embedding, input texts have to go through some pre-processing and tokenization steps. The text is split into tokens in the tokenization step, and a unique ID is assigned to each token. The list of IDs is called “sequence”. Typical transformers support short input sequences—for example, BERT, RoBERTa, and CodeBERT support sequences up to 512 tokens. Nevertheless, source code files are usually long. The token length distribution of the source code files in our dataset is available in the online appendix [17]. On the other hand, transformer architectures that support long input sequences require high GPU resources. The number of parameters will increase quadratically [26] with the increase of maximum sequence length. For example, the RoBERTa model has 124M trainable parameters (sequence length 512), whereas, for Reformer, it is 149M (sequence length 4096). A higher number of parameters implies that the model will require higher computing resources or longer training time (with the same computing

resource). Since short input sequence transformers cannot use the complete source code, it is a common assumption that they produce low-quality embedding.

Ding et al. [70] have studied the use of BERT for long input sequences and identified the attention decay of a typical transformer model. A typical attention mechanism requires high resources and becomes less effective over a long sequence. Thus, we may not achieve higher performance even after using higher resources.

Therefore, we need to know whether the cost of long input sequence transformers is justified. In this research question, we will investigate whether long input sequence transformers produce better embedding or not. This study has used transformer models with different sequence sizes. The maximum sequence size supported by those models is presented in Table 5.5 along with the average performance of all the bug-localization of models' performance in six projects. We can observe that there is no trend among the models' performance except that CodeBERT performed poorly than all other models. To check the observation, we conducted a pairwise Mann-Whitney test among the models. We found that the MRR and MAP of the CodeBERT model is less than the MRR and MAP of Long RoBERTa ($p < 0.001$), Long CodeBERT ($p = 0.002$ for MRR and $p = 0.004$ for MAP) and Reformer ($p < 0.001$ for both MRR and MAP) model (Bonferroni corrected α was $0.05/6 = 0.008$). The comparison among the other models was not statistically significant. The significant difference in performance among models may stem from their sequence lengths, which are 2 to 5 times longer than CodeBERT's and even 8 times longer in LongCodeBERT's case. This implies a higher computational cost for LongCodeBERT without notable performance improvements. This observation may help the developers of a bug localization tool when they have to balance between resource usage and performance gain.

Takeaway 8: Maximum sequence length has a mixed impact on generated embeddings' quality. The performance of transformers varies from project to project.

5.5 Threats to Validity

This section discusses potential threats to the validity of our case studies.

Internal Validity. In our study, we have trained embedding models on the BLDS dataset. For creating the BLDS dataset, we have to link bug reports with appropriate fixes. For linking bug reports with fixes (pull requests), we have followed the approach of Liwerski et al. [294]. However, this methodology is based on a heuristic, and a bug report might be

associated with wrong pull requests and source code files. Moreover, in some cases, issues such as coding style violations are also reported as bug reports. However, to mitigate the issues, we verified the link between a bug report and pull requests by checking the pull request’s attachments (if they exist). In our manual check, we found that typically non-software bug-related pull requests update many files at once. Thus we have filtered out all the pull requests that updated more than ten files. In this study, bug localization was approached as a binary classification problem using a CNN model on top of embedding models. Although ranking-based models are an option, their comparison with classification-based models requires further investigation, which is beyond the scope of this study.

External Validity. A possible threat to external validity is that we evaluated the performance of the embedding on only six projects. Moreover, all of these six projects are from the same community (Apache). Thus, it is possible that the performance may not represent the actual performance. However, bug reports from these six projects are often used to benchmark the performance of the bug localization model. Thus, even if the result is not generalizable, we can compare the result with other studies.

5.6 Conclusion

Our key takeaway is that the design choices in embedding creation significantly affect the performance of deep learning (DL) similarity-based bug localization models. In this study, we explored various design options for constructing multi-modal embeddings and evaluated the model’s performance in both in-project and cross-project settings. Notably, we found that larger transformers do not always yield better performance.

Future research could investigate alternative techniques, such as chunking, to handle longer source code files effectively. Additionally, enhancing the performance of cross-project bug localization models presents a valuable direction, especially given the high cost associated with project-specific training.

5.7 Chapter Summary

Before addressing developers’ concerns and designing bug localization tools, it was essential to understand the design choices and their impact. This chapter examines the design decisions involved in training embedding models and their influence on performance. Our findings highlight that data familiarity is a critical factor in achieving strong performance. While this aligns with current practices, it also presents the challenge of needing to retrain

the model for each new project, especially when labeled data is scarce or unavailable. In the next chapter, we will explore how reinforcement learning techniques can overcome this limitation and improve bug localization tools.

Chapter 6

RLocator: Reinforcement Learning for Bug Localization

Note. An earlier version of the work in this chapter appears in the IEEE Transactions on Software Engineering (TSE) Journal.

6.1 Introduction

To increase developer productivity and automate bug localization, various tools have been proposed in previous studies (e.g., Deeplocator [344], CAST [188], KGBugLocator [374], BL-GAN [390]). A common characteristic of these methods is their reliance on similarity-based approaches for identifying bugs. These techniques assess the similarity between bug reports and source code files using different methods such as cosine distance [54], Deep Neural Networks (DNN)[163], and Convolutional Neural Networks (CNN)[188]. The source code files are then ranked according to their similarity scores. During the training phase, the model is designed to optimize these similarity metrics. In contrast, in the testing phase, the model is tested with ranking metrics (e.g., Mean Reciprocal Rank (MRR) or Mean Average Precision (MAP)).

While most of these approaches showed promising performance, they optimize a metric that indirectly represents the performance metrics. Prior studies [330, 10, 354, 368] found that direct optimization of evaluation measures substantially contributes to performance

improvement of ranking problems. Direct optimization is also efficient compared to optimizing indirect metrics [368]. Hence, we argue that it is challenging for the solutions proposed by prior studies to sense how a wrong prediction would affect the performance evaluation measures [330]. In other words, if we use the retrieval metrics (e.g., MAP) in the training phase, the model will learn how each prediction will impact the evaluation metrics. A wrong prediction will change the rank of the source code file and ultimately impact the evaluation metrics.

Reinforcement Learning (RL) is a sub-category of machine learning methods where labeled data is not required. In RL, the model is not trained to predict a specific value. Instead, the model is given a signal about a right or wrong choice in training [302]. Based on the signal, the model updates its decision. This allows RL to use evaluation measures such as MRR and MAP in the training phase and directly optimize the evaluation metrics. Moreover, because of using MRR/MAP as a signal instead of a label, the problem of overfitting will be less prevalent. Markov Decision Process (MDP) is a foundational element of RL. MDP is a mathematical framework that allows the formalization of discrete-time decision-making problems [89]. Real-world problems often need to be formalized as MDP to apply RL.

In this paper, we present *RLocator*, an RL technique for localizing software bugs in source code files. We formulate RLocator into an MDP. In each step of the MDP, we use MRR and MAP as signals to guide the model to optimal choice. We evaluate RLocator on a benchmark dataset of six Apache projects and find that, compared with existing state-of-the-art bug localization techniques, RLocator achieves substantial performance improvement. While pinpointing the exact reasons for RL’s superior performance over other supervised techniques can be challenging, RL learns more generalizable approaches, especially in dynamic and complex environments. In comparison to supervised learning, it learns approaches that are more adaptable to a variety of situations [81, 153], which is a form of generalization. Additionally, RL demonstrates proficiency in scenarios where the optimal solution is not clearly defined, showcasing its versatility across various tasks and domains [302]. These factors can contribute to the superior performance of RLocator.

The main contributions of our work are as follows:

- We present RLocator, an RL-based software bug localization approach. The key technical novelty of RLocator is using RL for bug localization, which includes formulating the bug localization process into an MDP.
- We provide an experimental evaluation of RLocator with 8,316 bug reports from six Apache projects. When RLocator can localize, it achieves an MRR of 0.49 - 0.62,

MAP of 0.47 - 0.59, and Top 1 of 0.38 - 0.46 across all studied projects. Additionally, we compare RLocator’s performance with state-of-the-art bug localization methods. RLocator outperforms FLIM [187] by 38.3% in MAP, 36.73% in MRR, and 23.68% in Top K. Furthermore, RLocator exceeds BugLocator [383] by 56.86% in MAP, 41.51% in MRR, and 26.32% in Top K. In terms of Top K, RLocator shows improved performance over BL-GAN [390], with gains ranging from 55.26% to 3.33%. The performance gains for MAP and MRR are 40.74% and 32.2%, respectively.

6.2 Motivation

Reinforcement Learning (RL) stands out for its ability to learn from feedback, a characteristic that empowers models to self-correct based on the outcomes of their actions. This feature finds widespread application, exemplified by platforms like Spotify, an audio streaming service using RL to learn user preferences [211]. The model evolves and adapts by presenting music selections and refining recommendations through user interactions. The versatility of RL extends beyond entertainment; various companies [51] and domains [366] leverage its capacity for iterative learning and adjustment.

The proficiency required for bug localization is often acquired through experience, with seasoned developers exhibiting a faster bug-finding aptitude than their less experienced counterparts [338]. Recognizing the significance of experience in bug localization, we propose the integration of reinforcement learning into this domain. By employing RL, a model can present developers with sets of source code files as possible causes for a bug and learn from their feedback to enhance their skill in localizing bugs in the software. In contrast to conventional machine learning approaches, which rely solely on labeled data and lack easy adaptability, reinforcement learning presents two distinct advantages: firstly, the ability to learn from developer feedback, and secondly, the elimination of the requirement for labeled data in real-world scenarios. Therefore, our research aims to incorporate reinforcement learning into bug localization, leveraging its capacity to adapt and enhance performance through iterative feedback.

6.3 Background

In this section, we describe terms related to the bug localization problem, which we use throughout our study. Also, we present an overview of reinforcement learning.

6.3.1 Bug Localization System

A typical bug localization system utilizes several sources of data, e.g., bug reports, stack traces, and logs, to identify the responsible source code files. One particular challenge of the system is that the bug report contains natural language, whereas source code files are written in a programming language.

Typically, bug localization systems identify whether a bug report relates to a source code file. To do so, the system extracts features from both the bug report and the source code files. Previous studies used techniques such as N-gram [326, 217] and Word2Vec [148, 46] to extract features (embedding) from bug reports and source code files. Other studies (e.g., Devlin et al. [67]) introduced the transformer-based model BERT which has achieved higher performance than all the previous techniques. One of the reasons transformer-based models perform better in extracting textual features is that the transformer uses multi-head attention, which can utilize long context while generating embedding. Previous studies have proposed a multi-modal BERT model [83] for programming languages, which can extract features from both bug reports and source code files.

A bug report mainly contains information related to unexpected behavior and how to reproduce it. It mainly includes a bug ID, title, textual description of the bug, and version of the codebase where the bug exists. The bug report may have an example of code, stack trace, or logs. A bug localization system retrieves all the source code files from a source code repository at that particular version. For example, assume we have 100 source code files in a repository in a specific version. After retrieving 100 files from that version, the system will estimate the relevance between the bug report and each of the 100 files. The relevance can be measured in several ways. For example, a naive system can check how many words of the bug report exist in each source code file. A sophisticated system can compare embeddings using cosine distance [321]. After relevance estimation, the system ranks the files based on their relevance score. The ranked list of files is the final output of a bug localization system that developers will use.

6.3.2 Reinforcement Learning

In Reinforcement Learning (RL), the agent interacts with the environment through observation. Formally, an observation is called ‘State’, S . In each state, at time t , S_t , the agent takes action A based on its understanding of the state. Then, the environment provides feedback/reward \mathfrak{R} and transfers the agent into a new state S_{t+1} . The agent’s strategy to determine the best action, which will eventually lead to the highest cumulative reward, is referred to as *policy* [87, 302].

The cumulative reward (until the goal/end) that an agent can get if it takes a specific action in a certain state is called *Q value*. The function that is used to estimate the *Q value* is often referred as *Q function* or *Value function*.

In RL, an agent starts its journey from a starting state and then goes forward by picking the appropriate action. The journey ends in a pre-defined end state. The journey from start to end state is referred to as *episode*.

From a high level, we can divide the state-of-the-art RL algorithms into two classes. The first is the model-free algorithms, where the agent has no prior knowledge about the environment. The agent learns about the environment by interacting with the environment. The other type is the model-based algorithm. In a model-based algorithm, the agent uses the reward prediction from the model instead of interacting with the environment.

The bug localization task is quite similar to the model-free environment as we cannot predict/identify the buggy files without checking the bug report and source code files (without interacting with the environment). Thus, we use model-free RL algorithms in this study. Two popular variants of model-free RL algorithms are:

- *Value Optimization*: The agent tries to learn the Q value function in value optimization approaches. The agent keeps the Q value function in memory and updates it gradually. It consults the Q value function in a particular state and picks the action that will give the highest value (reward). An example of the value optimization-based approach is Deep Q Network (DQN) [302].
- *Policy Optimization*: In the Policy optimization approach, the agent tries to learn the mapping between the state and the action that will result in the highest reward. The agent will pick the action based on the mapping in a particular state. An example of the policy optimization-based approach is Advantage Actor-Critic (A2C) [236, 302].

A2C is a policy-based algorithm where the agent learns an optimized policy to solve a problem. In Actor-Critic, the actor-model picks action. The future return (reward) of action is estimated using the critic model. The actor model uses the critic model to pick the best action in any state. Advantage actor-critic subtracts a base value from the return in any timestep. A2C with entropy adds the entropy of the probability of the possible action with the loss of the actor model. As a result, in the gradient descent step, the model tries to maximize the entropy of the learned policy. Maximization of entropy ensures that the agent assigns almost an equal probability to an action with a similar return.

6.4 RLocator: Reinforcement Learning for Bug Localization

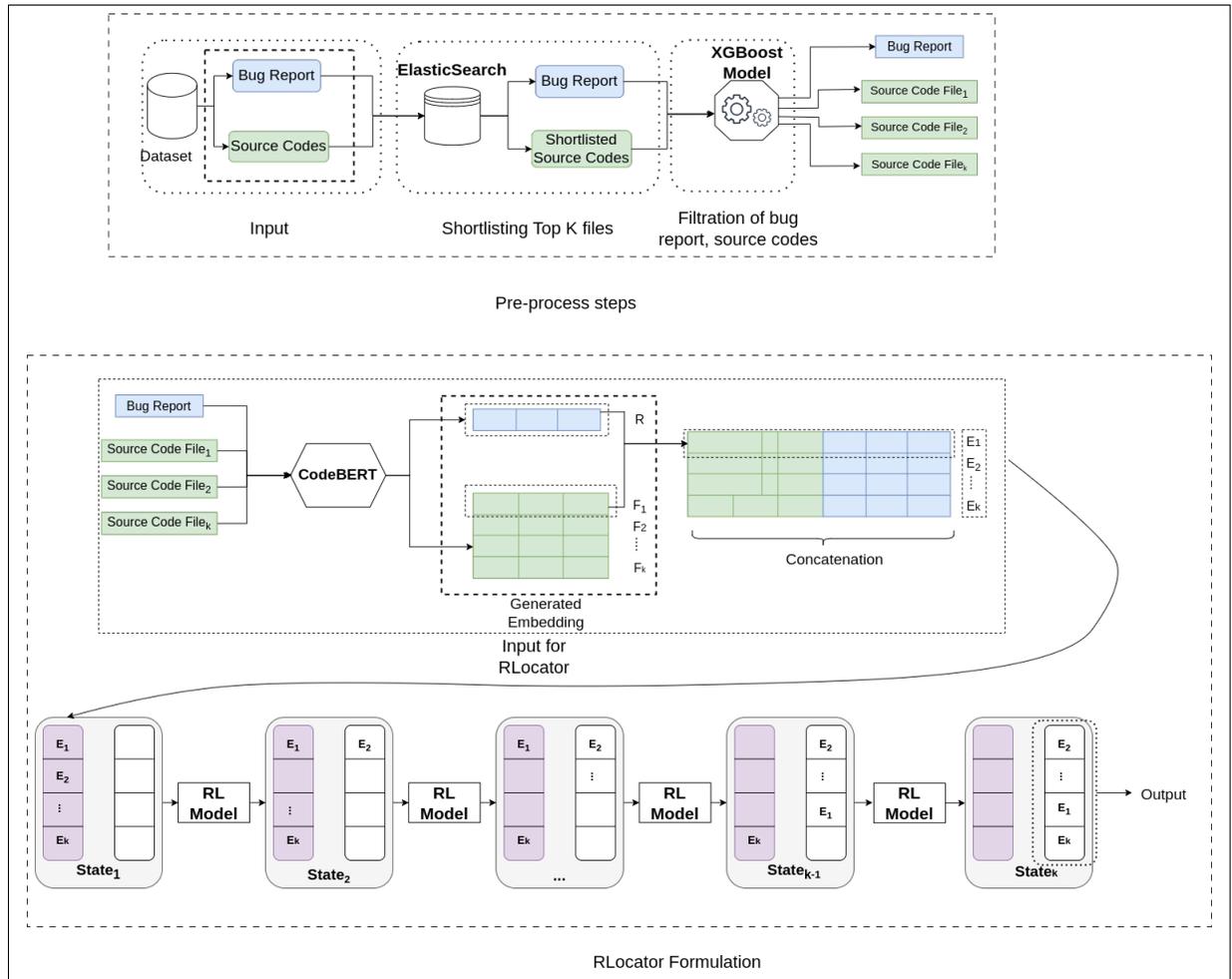


Figure 6.1: Bug Localization as Markov Decision Process.

In this section, we discuss the steps we follow to use RLocator. First, we explain (in Section 6.4.1) the pre-processing step required for using RLocator. Then, we explain (in Section 6.4.2) the formulation steps of our design of RLocator. We present the overview of our approach in Figure 6.1.

6.4.1 Pre-process

Before using the bug reports and source code files to train the RL model, they undergo a series of pre-processing steps. The steps are described in this section.

Input:The inputs to our bug localization tool are bug reports and source code files associated with a particular version of a project repository. Software projects maintain a repository for their bugs or issues (e.g., Jira, Github, Bugzilla). The first component, the bug report, can be retrieved from those issue repositories.

Each bug report is typically linked to a specific version or commit SHA of the project’s repository. First, we identify the buggy version mentioned in the bug report and collect all source code files from that version of the repository. Next, we extract and concatenate the title and description of the bug report. Finally, the concatenated title and description are used as a query to retrieve the second component: the source code. In the training phase, we compile bug reports and source code files into a dataset for subsequent usage. Our dataset contains a set of bug reports where each bug report has its own set of source code files. In real-world usage, RLocator directly accesses bug reports and source code files from the repository. In Figure 6.1, we illustrate the input stage where we get the bug report and source code files from the dataset.

Shortlisting source code files: The number of source code files in different versions of the repository can be different. All of the source code files can be potentially responsible for a bug. In this step, we identify K source code files as candidates for each bug. We limited the candidates to K as we cannot pass a variable number of source code files to the RL model. Moreover, given that RLocator primarily learns from developers’ feedback, its usage can prove challenging for a developer with many candidate source code files. To illustrate the issue, consider a repository with 700 files. RLocator presents files to the developer one by one for relevance verification. This sequential approach significantly prolongs the time taken to find a relevant file, resulting in a waste of developers’ time. Consequently, it is crucial to limit the number of files shown to developers by providing a shortlisted set for assessment.

To identify the K most relevant files, we use Elasticsearch (ES). ES is a search engine based on the Lucene search engine project. It is a distributed, open-source search and analytic engine for all data types, including text. It analyzes and indexes words/tokens for the textual match and uses BM25 to rank the files matching the query. We use the ES index for identifying the topmost k source code files related to a bug report. Following the study by Liu et al. [192] (who used ES in the context of code search), we build an ES index using the source code files and then queried the index using the bug report as the query. Then, we picked the first k files with the highest textual similarities with the bug report. We want to note that the goal of bug localization is to get the relevant files to be ranked

as close to the 1st rank as possible. Hence, metrics like MAP and MRR can measure the performance of bug localization techniques. While one can argue why we not only rely on ES to rank the relevant files, we find that the MAP and MRR of using ES are poor. Our RL-based technique learns from feedback and aims to rerank the output from ES to get higher MAP and MRR scores. In Figure 6.1, we illustrated the candidate refinement step where we query ElasticSearch using the bug report and use outputs to refine the candidate source code files.

Filtration of bug report and source code files: One limitation of ES is that it sometimes returns irrelevant files among the top k most relevant source code files. When there are no relevant files in the first k files, it hinders RLocator training using developer feedback and introduces noise. Therefore, we use an XGBoost-based binary classifier [50] to identify cases where ES may return no relevant files in the top k files. The rationale for using XGBoost is twofold: (1) to optimize developer time by not presenting irrelevant files and (2) to filter out noise during training.

ES-based filtering is not used because its similarity values are not normalized, and cosine similarity is inapplicable to text data. We provide the XGBoost model with the bug report and the top k files retrieved by ES to determine if any are relevant. If the XGBoost model predicts no relevant files in the set, we exclude those bug reports and their associated files. Each bug report is associated with its unique set of source code files, so filtering one does not impact others.

To build the model, we study the most important features associated with the prediction task. We consult the related literature on the field of information retrieval [206, 205, 177] and bug report classification [82] for feature selection. The list of computed features is presented in Table 6.1. For our dataset, we calculate the selected features and trained the model using 10-fold cross-validation. The results show that our classifier model has a precision of 0.78, a recall of 0.93, and an F1-score of 0.85. Additionally, the model is able to correctly classify 91% of the dataset (there will be relevant source code files in the top k files returned by ES).

After filtration, we pass each bug report and its source code files to RLocator. In Figure 6.1, we have depicted the operational procedure of RLocator. The workflow commences with a curated dataset containing bug reports and source code files. Subsequently, we index the source code files into the ES index. From ES, we obtain bug reports and shortlisted K source code files linked to those bug reports. Following this shortlisting, we employ the XGBoost model to predict the presence of a relevant file within the top K files. If at least one relevant file exists, we proceed to the next step by passing the bug reports and filtered source code files.

Table 6.1: Description and rationale of the selected features.

Feature	Description	Rationale
Bug Report Length	Length of the bug report.	Fan et al. [82] found that it is hard to localize bugs using a short bug report. A short bug report will contain little information about the bug. Thus it will be hard for the ElasticSearch to retrieve source code file responsible for this bug.
Source Code Length	Median length of the source code files associated with a particular bug. Note that we calculate the string length of the source code files after removing code comments.	Prior studies [206, 177] found that calculating textual similarity is challenging for long texts. Length of source code may contribute to the performance drop of ElasticSearch.
Stacktrace	Availability of stack trace in the bug report.	Schroter et al. [286] found that stacktraces in bug reports can help the debugging process as they may contain useful information. Availability of stacktraces may improve the performance of ElasticSearch.
Similarity	Ratio of similar tokens between a bug report and source code files	Similarity indicates the amount of helpful information in the bug report. We calculate the similarity based on the equation presented in Section 7.4.

6.4.2 Formulation of RLocator

In the previous step, we have pre-processed the dataset for training the reinforcement learning model. We shortlist k most relevant files for each bug report. After that, we identify the bug reports for which there will be no relevant files in top k files and filter out those bug reports. Finally, we pass the top k relevant files to RLocator. In this section, we explain how RLocator employs Reinforcement Learning for its bug localization. Our approach is grounded in the belief that each bug report contains specific indicators, such as terms and keywords, aiding developers in pinpointing problematic source code files. For example, in Java code, we can get a nested exception (the exception that leads to another exception, and another; an example is available in the online Appendix [19]). A developer can identify the root exception and which method call (or any other code block) caused that. After that, they can go for the implementation of that method in the source code files. The process indicates that developers can identify and use the important information from a bug report. Following prior studies [23, 315], we formulate RLocator into a Markov Decision Process (MDP) by dividing the ranking problem into a sequence of decision-making steps. A general RL model can be represented by a tuple $\langle S, A, \tau, \mathfrak{R}, \pi \rangle$, which is composed of states, actions, transition, reward, and policy, respectively. Figure 6.1 shows an overview of our RLocator approach. Next, we describe the formulation steps of each component of RLocator.

States: S is the set of states. The RL model moves from one state to another state until it reaches the end state. To form the states of the MDP, we apply the following steps:

Input:

The input of our MDP comprises a bug report and the top K relevant source code files from a project repository. We use CodeBERT [83], a transformer-based model, to convert text into embeddings, representing the text in a multi-dimensional space. CodeBERT is chosen for its ability to handle long contexts, making it suitable for long source code files where methods may be declared far from their usage. Unlike Word2Vec, which generates static embeddings for words, CodeBERT generates dynamic embeddings for sequences, capturing context during inference. This is crucial in source code files where variable use depends on scope.

CodeBERT, trained on natural language and programming language pairs, handles both programming and natural languages. Its self-attention mechanism assesses the significance of individual terms, helping link bug reports to relevant source code files. For example, in a Java nested exception, developers can identify the main exception and pinpoint the responsible code block. RLocator relies on this self-attention mechanism to identify and leverage these informative cues effectively.

In our approach, as shown in Figure 6.1, the embedding model processes bug reports and source code files, generating embeddings for the source code files F_1, F_2, \dots, F_k , and the bug report R .

Concatenation: After we obtain the embedding for the source codes and the bug report, we concatenate them. As prior studies [109, 372] suggest combining distinct sets of features through concatenation and processing them with a linear layer enables effective interaction among the features. Furthermore, feature interaction is fundamental in determining similarity [388, 143]. Thus, with the goal of calculating the similarity between a bug report and a source code file pair, we concatenate their embedding. Given our example in Figure 6.1, we concatenate the embedding of F_1, F_2, \dots, F_k with the embedding of bug report R independently. This step leads us to obtain the corresponding concatenated embedding E_1, E_2, \dots, E_k , as shown in Figure 6.1.

Note that each state of the MDP comprises two lists: a candidate list and a ranked list. The candidate list contains the concatenated list of embedding. As shown in our example in Figure 6.1, the candidate list contains E_1, E_2, \dots, E_k . In the candidate list, source code embeddings (code files and bug reports embedding concatenated together) are ranked randomly. The other list is the ranked list of source code files based on their relevance to the bug report R . Initially (at $State_1$), the candidate list is full, and the ranked list is empty. In each state transition, the model moves one embedding from the candidate list to the ranked list based on their probability of being responsible for a bug. In the final state, the ranked list will be full, and the candidate list will be empty. We describe the process of selecting and ranking a file in detail in the next step.

Actions: We define *Actions* in our MDP as selecting a file from the candidate list and moving it to the ranked list. Suppose at the timestep t ; the RL model picks the embedding E_1 , then the rank of that particular file will be t . In Figure 6.1 at the timestamp 1, the model picks concatenated embedding of file F_2 . Thus, the rank of F_2 will be 1. As in each timestamp, we are moving one file from the candidate list to the ranked list; the total number of files will be equal to the number of states and the number of actions. For identifying the potentially best action at any timestamp t , we use a deep learning (DL) model (indicated as *Ranking Model* in Figure 6.1), which is composed of a Convolutional Neural Network (CNN) followed by a Long Short-Term Memory (LSTM) [113]. Following [244, 125, 121], we use CNN to establish the connection between source code files and bug reports and extract relevant features. As mentioned earlier, developers acquire the ability to recognize cues and subsequently employ them to establish the association between source code files and bug reports. The CNN facilitates the second stage of bug localization, which involves extracting important features. The input of the CNN is the concatenated embedding of both bug reports and each source code file, and the output of CNN is extracted features

from the combined embedding of bug reports and source code files. The features are later used to calculate relevance.

On the other hand, LSTM [107] intends to make the model aware of a restriction, which we call *state awareness*. That is, in each timestamp, the model is allowed to pick the potentially best embedding that has not been picked yet, i.e., if a file is selected at $State_i$, it cannot be selected again in a later state (i.e., $State_{i+j}; j \geq 1$). The LSTM retains the state and aids the RL agent in choosing a subsequent action that does not conflict with prior actions. Thus, following previous studies [108, 25], we use an LSTM to make the model aware of previous actions. The LSTM takes a set of feature vectors as input and outputs the id of the source code file most suitable for the current state.

Transition: $\tau(S, A)$ is a function $\tau : S \times A \rightarrow S$ which maps a state s_t into a new state s_{t+1} in response to the selected action a_t . Choosing an action a_t means removing a file from the candidate list and placing it in the ranked list.

Reward: A reward is a value provided to the RL agent as feedback on their action. We refer to a reward received from one action as *return*. The RL technique signals the agent about the appropriate action in each step through the *Reward Function*, which can be modeled using the retrieval metrics. Thus, the RL agent can learn to optimize the retrieval metrics through the reward function. We consider two important factors in the ranking evaluation: the position of the relevant files and the distance between relevant files in the ranked list of embedding. We incorporated both factors in designing the reward function shown below.

$$\mathfrak{R}(S, A) = \frac{M * file\ relevance}{\log_2(t + 1) * distance(s)}; \text{if } A \text{ is} \\ \text{an action that has not been selected before} \quad (6.1)$$

$$\mathfrak{R}(S, A) = -\log_2(t + 1); \text{otherwise} \quad (6.2)$$

$$distance(S) = Avg.(Distance\ between\ currently \\ \text{picked\ subsequent\ related\ files}) \quad (6.3)$$

In Equations 6.1 and 6.2, t is the timestamp, S is State and A is Action. Mean reciprocal rank (MRR) measures the average reciprocal rank of all the relevant files. In Equation 6.1, $\frac{file\ relevance}{\log_2(t+1)}$ represents the MRR. The use of a logarithmic function in the equation is

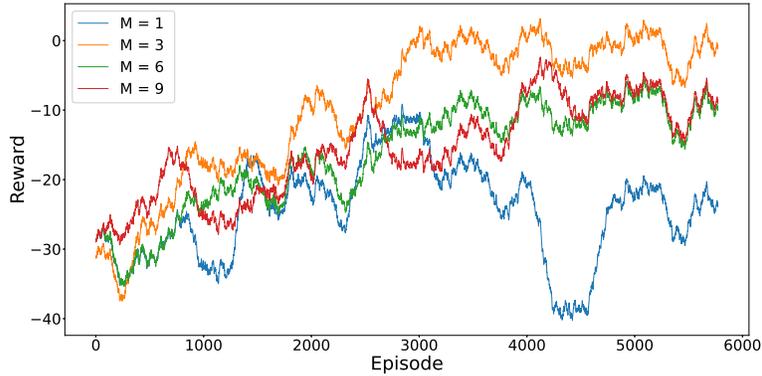


Figure 6.2: Effect of M in the reward-episode graph.

motivated by previous studies [100, 318], which found that it leads to a stable loss. When the relevant files are ranked higher, the average precision tends to be higher. To encourage the reinforcement learning system to rank relevant files higher, we introduce a punishment mechanism if there is a greater distance between two relevant files. By imposing this punishment on the agent, we incentivize it to prioritize relevant files in higher ranks, which in turn contributes to the Mean Average Precision (MAP).

We illustrate the reward functions with an example below. Presuming that the process reaches State S_6 and the currently picked concatenated embeddings are $E_1, E_2, E_3, E_4, E_5, E_6$ and their relevancy to the bug report is $\langle 0, 0, 1, 0, 1, 1 \rangle$. This means that these embeddings (or files) ranked in the 3rd, 5th, and 6th positions are relevant to the bug report. The position of the relevant files are $\langle 3, 5, 6 \rangle$, and the distance between them is $\langle 1, 0 \rangle$. Hence, $distance(S_6) = Avg.\langle 1, 0 \rangle = 0.5$. If the agent picks a new relevant file, we reward the agent M times the reciprocal rank of the file divided by the distance between the already picked related files. In our example, the last picked file, E_6 's relevancy is 1. Thus, we have the following values for Equation 6.1: $distance(S_6) = 0.5$; $\log_2(6 + 1) = 2.8074$; $file\ relevance = 1$. Note that M is a hyper-parameter. We find that three as the value of M results in the highest reward for our RL model. We identify the best value for M by experimenting with different values (1, 3, 6, and 9). Figure 6.2 shows the resulting reward-episode graph using different values of M. Hence, given $M = 3$, the value of the reward function will be $\mathfrak{R}(S, A) = \frac{3*1}{2.8074*0.5} = 2.14$. The reward can vary between M to ~ 0 . A higher value of the reward function indicates a better action of the model. Finally, in the case of optimal ranking, the $distance(S)$ will be zero. We handle this case by assigning a value of 1 for $distance(S)$. Even though we are using MRR and MAP as optimization goals we do not require labeled data. Instead, it learns from developers' feedback. It presents a limited set of files to the developer, seeking their feedback. If a developer deems a specific

file as relevant, they can click on it. This click feedback signifies the file’s relevance within the set. RLocator leverages this input at Equation 6.1 and learns the process of bug localization. Incorporating developers’ feedback may cause some inconvenience. However, all machine learning models are prone to data drift [260, 128, 12], where initial training data no longer matches current data, leading to declining performance. RLocator addresses this by continuously updating its learning based on developers’ feedback. Please note that if developers do not provide any feedback, we will not be able to use that particular session for training.

We limit the number of actions to 31 in RLocator. Since the number of states equals the number of actions, we also limit the number of states to 31. The prediction space of a reinforcement learning agent cannot be variable, and the number of source code files is variable. Thus, we must fix the number of actions, k , to a manageable number that fits in memory. We use an Nvidia V100 16 GB GPU and found that with more than 31 actions, training scripts fail due to out-of-memory errors. Therefore, we set $K = 31$ to keep the state size under control. As mentioned in Section 6.4.1, we select the top 31 relevant source code files from ES and pass them to RLocator, ensuring the number of states and files remains the same.

6.4.3 Developers’ Workflow

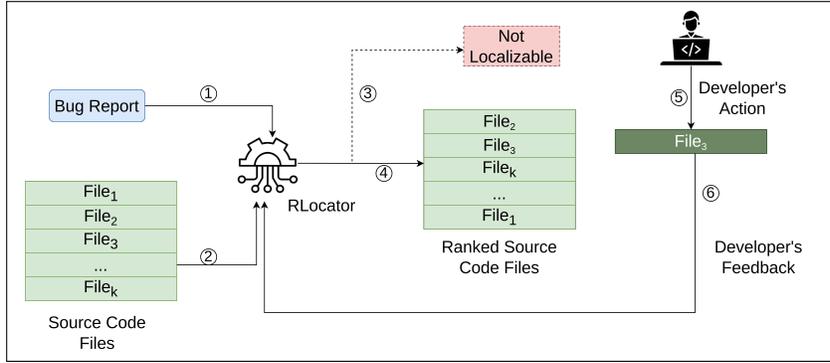


Figure 6.3: Developer interaction flow.

Figure 6.3 illustrates the interaction flow of developers using RLocator, which is represented as a central black box in the diagram. Details of RLocator are presented in Figure 6.1. The process begins with RLocator receiving two primary inputs: a bug report and all the source code files, labeled as 1 and 2, respectively, in the figure. After processing these inputs, RLocator outputs a ranked list of 31 source code files, indicated by step #4.

Developers then review this list to identify and select files that may contain bugs; an example is shown when a developer selects *File₃*, noted as step #5. This selection serves as feedback to RLocator, marked by step #6, aiding in refining its bug localization strategy. The feedback from developers is expressed as a binary value: files that developers open are marked with a 1, and all other files are marked with a 0. Additionally, the system can also indicate its inability to localize a bug for a given report, as shown by step #3. This ongoing loop allows RLocator to stay updated with changes in techniques and patterns, enhancing its bug localization performance.

6.5 Dataset and Evaluation Measures

In this section, we discuss the dataset used to train and evaluate our model (section 6.5.1). Then, we present the evaluation metrics we use for evaluation (section 6.5.2).

6.5.1 Dataset

In our experiment, we evaluate our approach on six real-world open-source projects [363], which are commonly used benchmark datasets in bug localization studies [187, 374]. Prior work has shown that this dataset has the lowest number of false positives and negatives compared to other datasets [173, 73, 383, 223, 293]. Following previous studies, we train our RLocator model separately for each of the six Apache projects (AspectJ, Birt, Eclipse Platform UI, JDT, SWT, Tomcat). Table 6.2 shows descriptive statistics on the datasets.

The dataset contains metadata such as bug ID, description, report timestamp, commit SHA of the fixing commit and buggy source code file paths. Each bug report is associated with a commit SHA/version, and we use a multiple version set matching approach to exclusively utilize the source code files linked to each specific report [173]. This approach closely resembles the bug localization process done by developers and reduces noise in the dataset, improving tool performance.

We identify the version containing the bug from the commit SHA and collect all relevant source code files from that version, excluding the bug-fixing code. This ensures our bug localization system closely mimics real-world scenarios.

For training and testing, we use 91% of the data, sorting the dataset by the date of bug reports and splitting it 60:40 for training and testing, respectively. Unlike previous studies [347] that used a 60:20:20 split, we repurpose validation data for testing to shorten the training duration.

Table 6.2: Dataset statistics.

Project	# of Bug Reports	Avg. # of Buggy Files per Bug
AspectJ	593	4.0
Birt	6,182	3.8
Eclipse UI	6,495	2.7
JDT	6,274	2.6
SWT	4,151	2.1
Tomcat	1,056	2.4

6.5.2 Evaluation Measures

The dataset proposed by Ye et al. [363] provides ground truth associated with each bug report. The ground truth contains the path of the file in the project repository that has been modified to fix a particular bug. To evaluate RLocator performance, we use the ground truth and analyze the experimental results based on three criteria, which are widely adopted in bug localization studies [383, 188, 374, 390, 54].

- **Mean Reciprocal Rank (MRR):** To identify the average rank of the relevant file in the retrieved files set, we adopted the Mean Reciprocal Rank. MRR is the average reciprocal rank of the source code files for all the bug reports. We present the equation for calculating MRR below, where A is the set of bug reports.

$$MRR = \frac{1}{|A|} \sum_A \frac{1}{\text{Least rank of the relevant files}} \quad (6.4)$$

Suppose we have two bug reports, $report_1$ and $report_2$. For each bug report, the bug localization model will rank six files. For $report_1$ the ground truth of the retrieved files are $[0, 0, 1, 0, 1, 0]$ and for $report_2$ the ground truth of the retrieved files are $[1, 0, 0, 0, 0, 1]$. In this case, the least rank of relevant files is 3 and 1, respectively, for $report_1$ and $report_2$. Now, the $MRR = \frac{1}{2}(\frac{1}{3} + \frac{1}{1}) = 0.67$

- **Mean Average Precision (MAP):** To consider the case where a bug is associated with multiple source code files, we adopted Mean Average Precision. It provides a measure of the quality of the retrieval [383, 287]. MRR considers only the best rank of relevant files; on the contrary, MAP considers the rank of all the relevant files in the retrieved files list. Thus, MAP is more descriptive and unbiased than MRR. Precision means how noisy the retrieval is. If we calculate the precision on the first two retrieved files, we will get precision@2. For calculating average precision, we have to figure precision@1, precision@2,... precision@k, and then we have to average the precision at different points. After calculating the average precision for each bug report, we have to find the mean of the average precision to calculate the MAP.

$$MAP = \frac{1}{|A|} \sum_A AvgPrecision(Report_i) \quad (6.5)$$

We show the MAP calculation for the previous example of two bug reports. The Average precision for $report_1$ and $report_2$ will be 0.37 and 0.67. So, the $MAP = \frac{1}{2}(0.36 + 0.67) = 0.52$

- **Top K:** For fare comparison with prior studies [121, 125] and to present a straightforward understanding of performance we calculate Top K. Top K measures the overall ranking performance of the bug localization model. It indicates the percentage of bug reports for which at least one buggy source appears among the top K positions in the ranked list generated by the bug localization tool. Following previous studies (e.g., [121, 125]), we consider three values of K: 1, 5, and 10.

Table 6.3: RLocator performance.

Project	Model	Top 1		Top 5		Top 10		MAP		MRR	
		91%	100%	91%	100%	91%	100%	91%	100%	91%	100%
AspectJ	RLocator	0.46	0.40	0.69	0.63	0.75	0.70	0.56	0.46	0.59	0.50
	BugLocator	0.36	0.28	0.50	0.45	0.56	0.51	0.33	0.31	0.49	0.48
	FLIM	0.51	0.36	0.65	0.60	0.72	0.67	0.41	0.35	0.47	0.45
	CodeBERT	0.4	0.35	0.59	0.55	0.65	0.61	0.49	0.39	0.51	0.44
	BL-GAN	0.41	0.38	0.6	0.55	0.71	0.65	0.33	0.31	0.42	0.39
Birt	RLocator	0.65	0.25	0.46	0.41	0.53	0.48	0.47	0.38	0.49	0.41
	BugLocator	0.61	0.15	0.27	0.21	0.34	0.29	0.30	0.30	0.39	0.38
	FLIM	0.49	0.18	0.39	0.34	0.47	0.42	0.29	0.25	0.31	0.28
	CodeBERT	0.33	0.22	0.39	0.35	0.46	0.43	0.41	0.33	0.42	0.35
	BL-GAN	0.17	0.16	0.33	0.3	0.46	0.42	0.32	0.29	0.4	0.37
Eclipse Platform UI	RLocator	0.45	0.37	0.69	0.63	0.78	0.73	0.54	0.42	0.59	0.50
	BugLocator	0.45	0.33	0.54	0.49	0.63	0.58	0.29	0.30	0.38	0.35
	FLIM	0.48	0.41	0.72	0.67	0.80	0.75	0.51	0.48	0.52	0.53
	CodeBERT	0.39	0.32	0.6	0.55	0.68	0.62	0.47	0.36	0.52	0.44
	BL-GAN	0.34	0.31	0.53	0.49	0.66	0.61	0.32	0.3	0.4	0.36
JDT	RLocator	0.44	0.33	0.67	0.61	0.78	0.75	0.51	0.44	0.53	0.45
	BugLocator	0.34	0.21	0.51	0.45	0.60	0.55	0.22	0.20	0.31	0.28
	FLIM	0.40	0.35	0.65	0.60	0.82	0.77	0.42	0.41	0.51	0.49
	CodeBERT	0.38	0.29	0.59	0.54	0.68	0.66	0.44	0.38	0.46	0.39
	BL-GAN	0.3	0.27	0.53	0.48	0.64	0.59	0.35	0.32	0.44	0.41
SWT	RLocator	0.40	0.30	0.57	0.51	0.63	0.58	0.48	0.42	0.51	0.44
	BugLocator	0.37	0.25	0.50	0.45	0.56	0.51	0.42	0.40	0.46	0.43
	FLIM	0.51	0.37	0.70	0.65	0.83	0.78	0.43	0.43	0.48	0.50
	CodeBERT	0.34	0.27	0.5	0.45	0.54	0.51	0.42	0.37	0.45	0.39
	BL-GAN	0.31	0.29	0.53	0.48	0.6	0.55	0.37	0.34	0.44	0.4
Tomcat	RLocator	0.46	0.39	0.61	0.55	0.73	0.68	0.59	0.47	0.62	0.51
	BugLocator	0.40	0.29	0.43	0.38	0.55	0.50	0.31	0.27	0.37	0.35
	FLIM	0.51	0.42	0.70	0.65	0.76	0.71	0.52	0.47	0.59	0.60
	CodeBERT	0.39	0.34	0.53	0.49	0.62	0.6	0.51	0.41	0.53	0.44
	BL-GAN	0.38	0.35	0.61	0.55	0.65	0.61	0.43	0.4	0.55	0.5

6.6 RLocator Performance

We evaluate RLocator on the hold-out dataset using the metrics described in Section 6.5.2. As there has been no RL-based bug localization tool, we compare RLocator with three state-of-the-art bug localization tools: BugLocator, FLIM, and BL-GAN.

A short description of the approaches is presented below.

- BugLocator [383]: an IR-based tool that utilizes a vector space model to identify the potentially responsible source code files by estimating the similarity between source code file and bug report.
- FLIM [187]: a deep-learning-based model that utilizes a large language model like CodeBERT.
- BL-GAN [390]: uses generative adversarial strategy to train an attention-based transformer model.

We use the original implementations to assess the performance of BugLocator [383] and FLIM [187]. Additionally, we fine-tune a CodeBERT [83] model as a baseline to demonstrate the benefits of using reinforcement learning. For tools like CAST [188], KGBugLocator [374], and BL-GAN [390], which lack replication packages, we refer to their respective studies. These studies show that KGBugLocator outperforms CAST, and BL-GAN outperforms KGBugLocator. Consequently, we replicate BL-GAN based on its study descriptions.

Regarding FBL-BERT [54], a recent technique, we do not compare it with RLocator. This is because FBL-BERT performs bug localization at the changeset level, and applying it to our file-level dataset would disadvantage FBL-BERT, as it is designed for shorter documents. Therefore, comparing it with RLocator would be unfair.

Furthermore, other studies, such as DeepLoc [347], bjXnet [102], CAST [188], KGBugLocator [374], and Cheng et al. [53], also propose deep learning-based approaches but do not provide replication packages. Although these studies evaluate similar projects, the lack of available code or pre-trained models prevents further comparison. However, to ensure comprehensive information, we include a table in our online appendix [19] displaying their performance alongside RLocator.

6.6.1 Retrieval performance

We use $k=31$ relevant files in RLocator, allowing us to rerank files for 91% of the bug reports. Table 6.3 shows RLocator’s performance on 91% and 100% of the data. RLocator is not designed for 100% data as it cannot rerank files if no relevant files are in the top k files. For such cases, we estimate performance assuming zero contribution, providing a lower bound for RLocator’s effectiveness. This conservative approach ensures we do not overestimate the technique’s effectiveness. **Table 6.3 showcases that RLocator achieves better performance than BugLocator and FLIM in both MRR and MAP across all studied projects when using the 91% data.** On 91% data, RLocator outperforms FLIM by 5.56-38.3% in MAP and 3.77-36.73% in MRR. Regarding Top K, the performance improvement is up to 23.68%, 15.22%, and 11.32% in terms of Top 1, Top 5, and Top 10, respectively. Compared to BugLocator, RLocator achieves performance improvement of 12.5 - 56.86% and 9.8% - 41.51%, in terms of MAP and MRR, respectively. Regarding Top K, the performance improvement is up to 26.32%, 41.3%, and 35.85% in terms of Top 1, Top 5, and Top 10, respectively. The results indicate that RLocator consistently outperforms BL-GAN in 91% settings across all metrics. Specifically, in the TopK measurements, RLocator’s performance exceeded that of BL-GAN, with improvements ranging from 55.26% to 3.33%. Additionally, RLocator achieved performance gains of 40.74% in MAP and 32.2% in MRR, respectively.

The results point out that RLocator outperforms BL-GAN across all the metrics in 91% settings. Specifically, in TopK, RLocator achieved better performance than BL-GAN, ranging from 3.33% to 55.26%. The performance gain is 40.74% and 32.2% for MAP and MRR, respectively.

Compared to the CodeBERT model trained as a classifier (CodeBERT), RLocator achieves better performance across all the metrics. CodeBERT model archives consistently lower performance across all the metrics. The performance drops up to 17.65%, 15.63%, 17.95%, 17.14%, and 16.67% for Top1, Top5, Top 10, MAP, and MRR, respectively.

When we consider 100% of the data, RLocator has better MAP results than FLIM in three out of the six projects (AspectJ, Birt, and JDT) by 6.82-34.21%, equal to FLIM in one project (Tomcat) and worse than FLIM in 2 projects (Eclipse Platform UI and SWT) by 2-14%. The underperformance of RLocator in the Eclipse Platform UI and SWT projects can be linked to the poor and inconsistent quality of bug reports, which creates a significant lexical gap between the reports and the source code. By applying the IMaChecker [296] approach, we discovered that AspectJ reports are of the highest quality, whereas those for Eclipse Platform UI, SWT, and Tomcat rank among the lowest. For a detailed analysis of bug report quality, please refer to the online Appendix [19]. In terms of MRR, RLocator is

better than FLIM in 2 projects (AspectJ and Birt) by 10-31.71% and worse than FLIM in the remaining four projects (Eclipse platform UI, JDT, SWT, and Tomcat) by 6-18%. In terms of Top K, RLocator ranks 4.29-12.5% more bugs in the top 10 positions than FLIM in two projects. On the other hand, in the rest of the four projects, FLIM ranks more bugs in the top 10 positions, ranging between 2.74 - 34.48%. When comparing RLocator with BugLocator for the 100% data along MAP, we find that the RLocator is better in five of the six projects and similar in just the Tomcat project. With respect to MRR, RLocator is better than BugLocator in all six projects. In terms of Top K, RLocator ranks more bugs than BugLocator in the top 10 position, where the improvement ranges between 12.07 -39.58%. The results demonstrate that RLocator consistently surpasses BL-GAN in all metrics in the 100% setting. Specifically, in the TopK metric, RLocator’s performance was better than BL-GAN, with improvements ranging from 3.33% to 36%. The performance enhancements for RLocator are 32% in MAP and 1.96% in MRR, respectively.

It is important to note that MAP provides a more balanced view than MRR and top K since it accounts for all the files that are related to a bug report and not just one file. Additionally, in our technique, we optimize to give more accurate results for most of the bug reports than give less accurate results on average for all the bug reports. Thus, **by looking at the MAP data for the 91%, we can see that RLocator performs better than the state-of-the-art techniques in all projects. Even if we consider 100% of the data, RLocator is still better than other techniques in the majority of the projects.** Only with 100% of the data and when using MRR as the evaluation metric, RLocator does not perform better than the state-of-the-art in most projects.

RLocator performs the worst in the Birt project, with a performance drop of 10.47% in MAP, 11.71% in MRR, and 41.42% in the top 10 compared to its average on 91% of the data. Despite this, RLocator outperforms FLIM by 38.3% in MAP, 36.73% in MRR, and 11.32% in the top 10. It also surpasses BugLocator by 36.17% in MAP, 20.41% in MRR, and 35.85% in the top 10. Factors like bug report quality, amount of information in the bug report, and source code length may contribute to the performance drop. We measure helpful information in bug reports using a *similarity* metric, which calculates the ratio of similar tokens between source code files and bug reports, indicating the potential usefulness of the report for bug localization. The metric is defined in equation 6.6.

$$Similarity = \frac{Bug\ Report\ Tokens \cap File\ Tokens}{\#\ of\ Unique\ Tokens\ in\ Bug\ Report} \tag{6.6}$$

The median similarity scores for the Birt, Eclipse Platform UI, and SWT projects are 0.29, 0.30, and 0.33, respectively, making them the lowest among the six projects. This observation suggests that the lower quality of bug reports (reflected in their similarity to

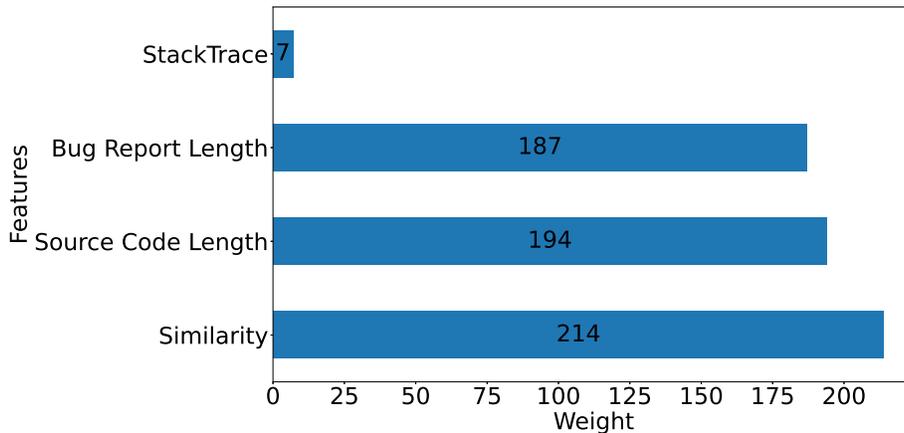


Figure 6.4: Feature importance of classifier model.

source files) may contribute to the decreased performance of RLocator in these projects.

To effectively use RLocator in real-world scenarios, we employ an XGBoost model (Section 6.4.1) to filter out bug reports where relevant files do not appear in the top K ($=31$) files. We then compute the importance of features listed in Table 6.1 using XGBoost’s built-in module. The importance score indicates each feature’s contribution to the model, with higher values signifying greater importance. Figure 6.4 shows that similarity is the most crucial feature, followed by source code length and Bbg report length. These findings highlight the importance of similarity in text-based search systems and suggest that high-quality bug reports can influence localization performance.

6.6.2 Entropy Ablation Analysis: Impact of Entropy on RLocator performance

We conduct an ablation study to gain insights into the significance of each component of RLocator. The main two components of RLocator are the ES-based shortlisting step and the reinforcement learning step.

In RL, we have used the A2C with entropy algorithm. Entropy refers to the unpredictability of an agent’s actions. A low entropy indicates a predictable policy, while high entropy represents a more random and robust policy. An agent in RL will tend to repeat actions that previously resulted in positive rewards while learning the policy. The agent may become stuck in a local optimum due to exploiting learned actions instead of exploring

Table 6.4: RLocator performance with and without Entropy for A2C.

Project	Model	Top 1	Top 5	Top 10	MAP	MRR
AspectJ	ES	0.15	0.20	0.28	0.23	0.27
	A2C	0.27	0.39	0.48	0.40	0.52
	A2C with Entropy	0.46	0.69	0.75	0.56	0.59
Birt	ES	0.10	0.14	0.17	0.18	0.23
	A2C	0.21	0.30	0.43	0.31	0.42
	A2C with Entropy	0.38	0.46	0.53	0.47	0.49
Eclipse Platform UI	ES	0.09	0.15	0.19	0.25	0.31
	A2C	0.25	0.38	0.51	0.39	0.51
	A2C with Entropy	0.45	0.69	0.78	0.54	0.59

new ones and finding a higher global optimum. This is where entropy comes useful: we can use entropy to encourage exploration and avoid getting stuck in local optima [7]. Because of this, entropy in RL has become very popular in the design of RL approaches such as A2C [134]. In our proposed model (Section 6.6.1), we use A2C with entropy to train the RLocator aiming to rank relevant files closer to each other. As entropy is part of the reward, the gradient descent process will try to maximize the entropy. Entropy will increase if the model identifies different actions as the best in the same state. However, those actions must select a relevant file; otherwise, the reward will be decreased. Thus, if there are multiple relevant files in a state, the A2C with entropy regularized model will assign almost the same probability in those actions (actions related to selecting those relevant files). This means that when the states are repeated, a different action will likely be selected each time. This probability assignment will lead to a higher MAP.

The observed performance of RLocator in achieving higher MAP can be interpreted due to two factors: 1) the way we design our reward function, given that we define a function that aims to encourage higher MAP; 2) the inclusion of entropy, as entropy regularization is assumed to enable the model to achieve higher MAP.

Hence, to provide a better understanding of our model, we measure the performance of three different steps of our model separately. The ES-based shortlisting step, the A2C-based RL model (without entropy), and the A2C with entropy model. Due to resource (time and GPU) limitations, we limit our evaluation to half of the total projects in our dataset, i.e., AspectJ, Birt, and Eclipse Platform UI. We observe a similar trend in those three projects. Thus, we believe our results will follow a similar trend in the remaining projects.

Table 6.4 presents the performance of the three choices (i.e., ES, A2C only, and A2C with Entropy). Table 6.4 shows that ES archives the baseline performance, which is 53-61% lower than the A2C with entropy model in terms of MAP and 47-54% lower in terms of MRR. We also find that the MRR and MAP of the models without entropy are lower than those of the A2C with entropy models. Table 6.4 shows that in terms of MAP, the performance of A2C with entropy models is higher than A2C models by a range of 27.78-34.04%. In MRR and the top 10, the A2C with entropy model achieves higher performance by a range of 11.86-13.56% and 18.87 - 36%, respectively. Such results indicate that entropy could substantially contribute to the model performance regarding MAP, MRR, and Top K. Moreover, this shows that the use of entropy encourages the RL agent to explore possible alternate policies [219]; thus, it has a higher chance of getting a better policy for solving the problem in a given environment than the A2C model.

6.7 Threats to Validity

RLocator has a number of limitations as well. We identify them and discuss how to overcome the limitations below.

Internal Validity. One limitation of our approach is we are not able to utilize 9% of our dataset due to the limitation of text-based search. One may point out that we exclude the bug reports where we do not perform well. But our the XGBoost model in our approach automatically identifies them and we say that we would rather not localize the source code files for these bug reports than localize them incorrectly. Hence, developers need to rely on their manual analysis only for the 9%. Moreover, as a measure of full transparency, we estimate the lower bound of RLocator performance for the 100% data and show that the difference is negligible.

External Validity. The primary concern for the external validity of the RLocator evaluation stems from its limitation to a small number of bugs in six varied, real-world open-source

projects, potentially impacting its broad applicability. However, those projects are from different domains and used by prior studies [344, 188, 387, 374, 163, 317]. Furthermore, the A2C without entropy model was only evaluated on three projects because of the substantial resources required for training—taking about four days on an Nvidia V100 16GB GPU. The uniform outcomes across these projects indicate that similar results could be expected in the remaining projects. Additionally, due to the absence of a replication package, we replicated BL-GAN based on its description in the original study, which may lead to slight performance deviations. Nevertheless, after experimenting with various hyperparameters, we selected a set that achieves comparable performance to that reported in the original study.

Construct Validity. Finally, our evaluation measures might be one threat to construct validity. The evaluation measures may not completely reflect real-world situations. The threat is mitigated by the fact that the used evaluation measures are well-known [54, 187, 363, 383] and best available to measure and compare the performance of information retrieval-based bug localization tools.

6.8 Conclusion

In this paper, we propose RLocator, a reinforcement learning-based (RL) technique to rank the source code files where the bug may reside, given the bug report. The key contribution of our study is the formulation of the bug localization problem using the Markov Decision Process (MDP), which helps us to optimize the evaluation measures directly. We evaluate RLocator on 8,316 bug reports and find that RLocator performs better than the state-of-the-art techniques when using MAP as an evaluation measure. Using 91% bug reports dataset, RLocator outperforms prior tools in all the project in terms of both MAP and MRR. When using 100% data, RLocator outperforms all prior approaches in four of six projects using MAP and two of the six projects using MRR. RLocator can be used along with other bug localization approaches to improve performance. Our results show that RL is a promising avenue for future exploration when it comes to advancing state-of-the-art techniques for bug localization. Future research can explore the application of advanced reinforcement learning algorithms in bug localization. Additionally, researchers can investigate how training on larger datasets impacts the performance of tools in low-similarity contexts.

6.9 Chapter Summary

This chapter introduces a reinforcement learning-based approach aimed at reducing bug localization tools' reliance on labeled data. By learning directly from developer feedback, our method becomes more effective for new projects, particularly those lacking high-quality or sufficient labeled data. This addresses a key concern from developers regarding the performance of bug localization tools in such scenarios. In the next chapter, we will explore how to scale bug localization tools for cross-project and cross-language applications by enhancing the model's generalizability.

Chapter 7

Bug Localization in Cross-Language and Cross-Project Settings

Note. An earlier version of the work in this chapter is under submission in the IEEE Transactions on Software Engineering (TSE) Journal.

7.1 Introduction

While previous works (e.g., [184, 347, 188, 374, 390]) have introduced numerous bug localization tools, they rely heavily on labeled data and often require retraining for each new project, which limits their cross-project applicability [156, 284]. Although progress has been made with tools that leverage transfer learning to generalize across different projects (e.g., [125, 388]), they still require labeled data from new projects for retraining. Furthermore, these tools have primarily been tested in single-language environments, raising questions about their effectiveness in cross-project bug localization across multiple programming languages.

Recent studies have highlighted that Large Language Models (LLM) provide detailed and context-rich representations for code retrieval tasks and bug diagnosis [83, 99]. This improved representation can enhance the ability of bug localization tools to accurately link bug reports with the corresponding source code files by integrating context-sensitive semantic understanding [54, 75]. However, such methods still encounter two major challenges. Firstly (**Challenge 1: Limited Context Window**), the typical context window

of LLMs, which ranges from 512 to 2,048 tokens, is insufficient when compared to the average token length of 2,536 for a single source code file in common datasets [363].

Secondly (**Challenge 2: Mapping Accuracy**), such methods struggle to accurately map bug reports to the relevant buggy source code file, as illustrated in Figure 7.1a, where the ideal non-overlapping distribution between buggy and non-buggy source code is not achieved. For this challenge, we highlight the need to prioritize the importance of source code files within projects. Consider the scenario depicted in Figure 7.2, where a font issue involves four files. The model erroneously associates the bug with ‘ThemeManager.java,’ an irrelevant file, and misses ‘FontController.java,’ which is crucial to the issue. Focused learning from such challenging examples can improve the model, aligning the representations of buggy source code files more closely with the bug report.

To overcome the above two challenges, we propose **BLAZE**, an approach that employs:

- **Dynamic Chunking:** Using dynamic programming, **BLAZE** segments source code at the boundaries of source code component declarations (e.g., class, interface, and method) to minimize continuity loss and adhere to context length constraints. We design these segments to be placed at component edges to prevent overlapping, resolving issues of continuity loss and excessive resource consumption caused by repetitive tokens.
- **Hard Example Learning:** **BLAZE** employs a GPT-based bug localization model tailored for identifying bugs across multiple languages and projects. We fine-tune it by prioritizing complex bugs (hard examples). **BLAZE** identifies such cases while tuning out less relevant outliers, which enhances the ability of **BLAZE** to generalize across different projects and languages and identifies critical misclassifications [258, 355]. As demonstrated in Figure 7.1b, we find that the fine-tuning method could minimize the similarity overlap between buggy and non-buggy code files.

BLAZE localizes bugs at the file level, narrowing the search space for faster bug localization and seamless integration with version control systems [272, 370]. We chose file-level localization for **BLAZE** to avoid the complexity (e.g., tracking changes in finer granularity [294, 38], hierarchical dependency of changes [105]) and overhead associated with method or commit-level approaches. This balance enhances both accuracy and usability [262, 111]. Additionally, higher granularity bug localization often leads to more false positives, which negatively impacts developers’ experience [395].

To evaluate the capability of **BLAZE** across languages, we compile **BEETLEBOX**, the most extensive cross-language and cross-project bug localization dataset to date, to the best of

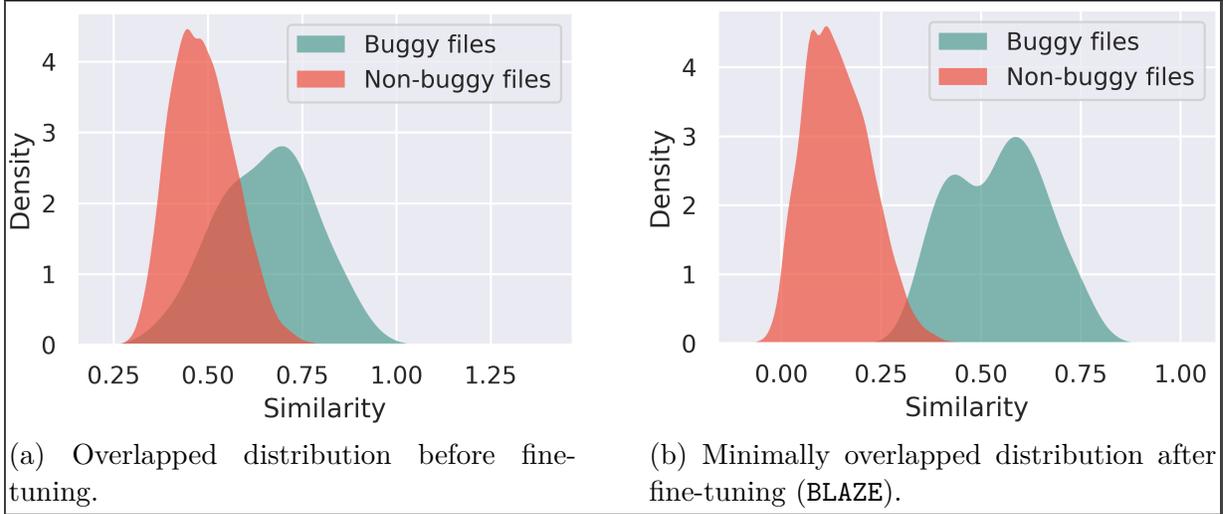


Figure 7.1: Kernel Density Estimate plot showing the distribution of similarities between buggy and non-buggy source code files and bug reports.

our knowledge. **BeetleBox** comprises 26,321 bugs that we mine from 29 real-world and thriving projects across five widely used programming languages: C++, Java, JavaScript, Python, and Go.

We evaluate the performance of **BLAZE** using three bug localization benchmarks, namely SWE-Bench [136], Ye et al. [363], and **BEETLEBOX**, which together encompass 31,745 bugs from 36 real-world software projects. **BLAZE** achieves up to 141% better Top 1 accuracy and 107.69% better MAP than three traditional cross-project bug localization tools [125, 383, 388]. Compared to modern language model-based tools [54, 75, 235], **BLAZE** achieves improvements of up to 63% in Top 1 and 69% in MAP. Moreover, we conduct an extensive ablation study, which confirms that our newly designed pipeline and fine-tuning techniques enhance performance. The results show that different components of our pipeline substantially contribute to improving bug localization performance to different degrees.

Contributions. In summary, this paper makes the following key contributions:

- We introduce a novel bug localization technique (i.e., **BLAZE**). In this technique,
 - We propose a dynamic chunking technique to overcome the limitation of transformer-based approaches (limited context window).

- We fine-tune BLAZE by learning from hard examples. BLAZE focuses on complex and frequently misclassified cases to enhance model performance in cross-language, cross-project settings.
- We create **BeetleBox**, the largest to-date cross-language and cross-project bug localization dataset.
- We conduct an ablation study to understand the importance of each component of the BLAZE pipeline.

7.2 Methodology

In this section, we describe our approach to designing BLAZE, which consists of three phases: **Phase A** introduces a fine-tuned language model to associate textual (bug report) and code elements (source code files). **Phase B** implements a dual-indexing system for source code files using both text and vector data from the fine-tuned model to facilitate a more effective retrieval of relevant files. Finally, in **Phase C**, we leverage the fine-tuned model from Phase A and the resulting indexed databases from Phase B together to facilitate the full process of matching incoming bug reports with the corresponding source code files. Below, we describe each phase in detail.

Phase A. Fine-tuning of Language Model

Figure 7.3 presents the steps we follow to fine-tune the model used in BLAZE. Each step of this process is detailed below.

Extraction of Source Code Elements. Our process begins with obtaining bug reports, the specific repository version where the bug exists, and all the source code files from that version provided in the targeted dataset (more details on how we collect and process the dataset in this study are described in Section 7.3.2). In this initial step, we focus on extracting details such as the names and roles of components (e.g., methods, classes, and interfaces) from the collected source code files. Drawing on insights from prior research [191], which indicates that separate indexing of source code files enhances lexical retriever’s performance in code search tasks, we index source code files separately in the next phase of our pipeline (Phase B).

Moreover, we extract positional information, such as line numbers, which is essential for subsequent analysis stages. To identify all components within the source code, we use

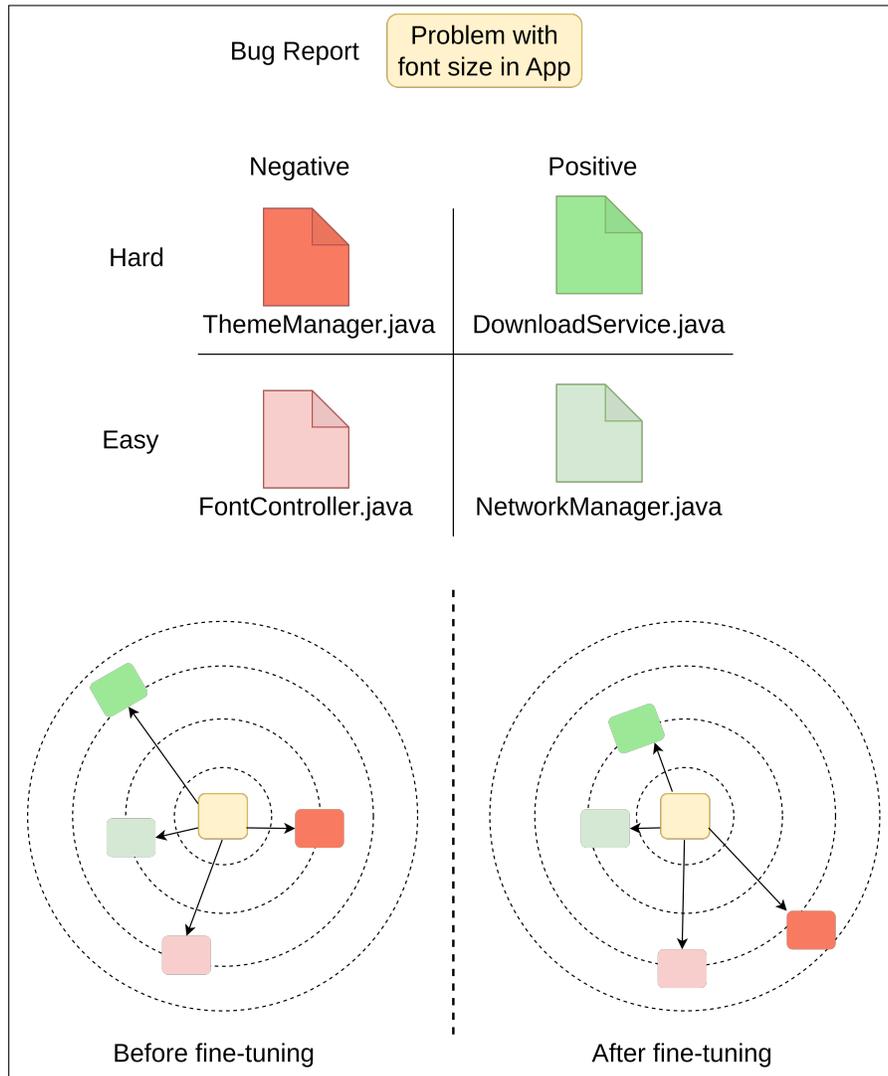


Figure 7.2: Example of mining hard examples.

tree-sitter,¹ a tool that structures code into a graph format. From this tree-sitter graph, we extract all components along with their types, positions (line numbers), and names.

Dynamic chunking of source code file. In this step, we address the challenge of segmenting lengthy source code files into smaller segments that are suited for context-aware embedding models [67, 195, 27]. We employ a dynamic programming approach for

¹<https://github.com/tree-sitter/tree-sitter>

Algorithm 1: Dynamic Chunking.

Input :

cost_map: Map containing costs for each code component type.
total_lines: Total number of lines.
window_size: Maximum chunk size.

Output :

dp: Array containing minimum costs and corresponding breakpoints

```
1 Function dynamic_chunk(cost_map, total_lines, window_size)
2   line_map ← get line number to component type map
3   split_cost_map ← get line number to cost map
4   for i ← 1 to total_lines do
5     minimum_cost ← INFINITY
6     breakpoint ← -1
7     for j ← MAX(i - window_size, 0) to i do
8       cost = dp[j][0] + split_cost_map[i] or DEFAULT_COST
9       if cost < minimum_cost then
10        minimum_cost = cost
11        breakpoint = j
12    dp[i] = (minimum_cost, breakpoint)
13  return dp
```

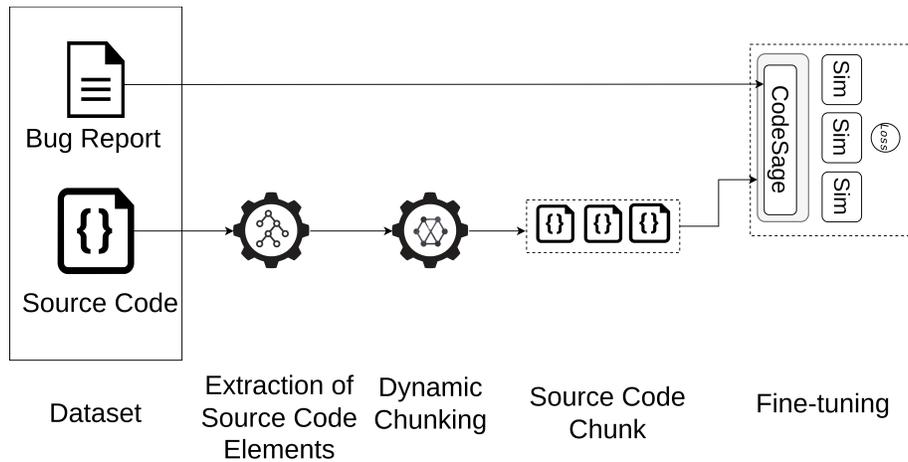


Figure 7.3: Fine-tuning of language model (Phase A).

chunking. This method segments files at natural boundaries (e.g., classes, interfaces, and methods), ensuring continuity and preventing overlaps—issues commonly associated with the traditional static chunking method and the sliding window technique.

Our algorithm, outlined in Algorithm 1, details the pseudo-code for our dynamic chunking approach. In the algorithm, we define ‘DEFAULT_COST’ as the cost of a random line. The approach begins by assigning each line of code a specific cost and categorizing it by component type. It then calculates the least expensive way to segment the code up to each line iteratively. It evaluates all potential breakpoints within the predefined window size (maximum context window supported by the intended model), updates to the minimum cost when a less expensive option is identified, and records the optimal breakpoint. The final output of the algorithm includes a list of the minimal costs associated with each breakpoint, alongside the breakpoints themselves. These are used to divide the source code files into smaller chunks.

Learning from hard examples. We fine-tune a Large Language Model (LLM), called CodeSage [371]. CodeSage is a GPT-based multi-modal embedding model. We adopt CodeSage to align bug reports and source code files within the same embedding space, drawing on its proven effectiveness in zero-shot code search tasks [371]. However, its effectiveness in bug localization is still limited, as shown in Figure 7.1a. A crucial difference between CodeSage and BLAZE lies in their strategies for mining hard examples; BLAZE mines in the embedding space, while CodeSage targets hard examples from docstrings and code. Mining in the embedding space enables BLAZE to detect hard examples across different contexts, enhancing its ability to generalize across a variety of programming lan-

guages and projects. To optimize CodeSage for bug localization, we implement contrastive learning, which helps to distinguish between similar and dissimilar data pairs, i.e., this technique adjusts the model to minimize the distance between embeddings of bug reports and their corresponding buggy files (positive) while maximizing the separation from non-buggy files (negative). This distinction is guided by the Normalized Temperature-scaled Cross-Entropy (NT-Xent) loss [46], defined as:

$$\mathcal{L}_{i,j} = -\log \frac{\exp^{sim(z_i, z_j)/\tau}}{\sum_{k=1}^N 1_{[k \neq i]} \exp^{sim(z_i, z_k)/\tau}} \quad (7.1)$$

where z_i is the embedding of the bug report, z_j is the embedding of the positive (buggy) source code file, and τ is the temperature parameter that scales the similarity. This loss function distinguishes between positive and negative examples by comparing similarity scores.

However, the standard NT-Xent loss may fall short in distinguishing between positive and negative source code files, potentially causing a bug report to align closer with a negative file. To mitigate this, we incorporate in-batch hard example mining into our training loop. This strategy prioritizes the most challenging cases: positive files with low similarity to the bug report and negative files with unexpectedly high similarity. We identify these hard examples by calculating the median similarity of all pairs in each batch. Negative source code files with similarity above this median and positive source code files below the median are considered hard examples. These cases are then given priority in the training process, with their impact on the loss function adjusted as follows:

$$\mathcal{L}_{i,j} = -\log \frac{M_{i,j} * \exp^{sim(z_i, z_j)/\tau}}{\sum_{k=1}^N 1_{[k \neq i]} \exp^{sim(z_i, z_k)/\tau} * M_{i,k}} \quad (7.2)$$

$$M_{m,n} = \begin{cases} \alpha, & \text{if } z_n \text{ negative and } sim(z_m, z_n) > median(sim(z_u, z_v)) \\ \beta, & \text{if } z_n \text{ positive and } sim(z_m, z_n) < median(sim(z_u, z_v)) \\ 1, & \text{otherwise} \end{cases}$$

$$\alpha > 1; \beta > 1; 1 \leq u, v \leq N$$

In an average project, approximately 700 source code files exist, with only 1-2 directly related to a reported bug, as evidenced in Apache project dataset [363]. This rarity of relevant files presents a considerable challenge in bug localization. Our approach of hard example mining and scaling enhances the gradients from challenging cases and helps maintain a balanced learning environment, preventing the model from being overwhelmed by

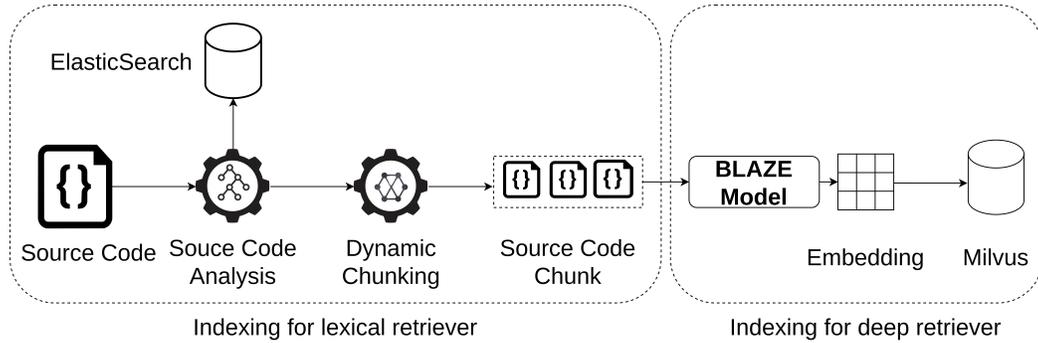


Figure 7.4: Offline indexing phase of BLAZE (Phase B).

outliers. Moreover, the continuous nature of the scaling multiplier preserves the convergence properties of the NT-Xent loss. By employing median-based scaling, our method could improve the model learning from difficult examples, thereby enhancing its convergence. Henceforth, we refer to the fine-tuned language model as the BLAZE model.

Phase B. Offline Indexing

We introduce the offline indexing phase to enhance bug localization by proactively processing and indexing source code files. As mentioned earlier, repositories contain (on average) about 700 source code files, and without this phase, developers could face delays due to real-time data processing.

As depicted in Figure 7.4, upon receiving a bug report, BLAZE employs two distinct indexing strategies: the *deep retriever* stores data in the Milvus vector database [320], while the *lexical retriever* uses the no-SQL database ElasticSearch [95]. Having two types of retrievers can help to offset their weaknesses and take advantage of their strengths. Note that this phase builds upon the earlier phase of extracting names and positions of components (see Phase A). Below, we describe each retriever in detail.

Indexing for lexical retriever. Lexical retrievers are fast and use keyword matching to assess similarity. After extracting component names and positions as detailed in Phase A), each source code file is indexed in ElasticSearch. ElasticSearch leverages the Okapi BM25 algorithm [279] to compute similarity scores, a technique well-documented in previous studies for applications in code retrieval [191, 282]. Additionally, we index filenames and file paths to match logs and stack traces within bug reports.

Indexing for deep retriever. Deep retrievers integrate semantics to calculate the similarity between bug reports and source code files. Starting with the dynamically chunked

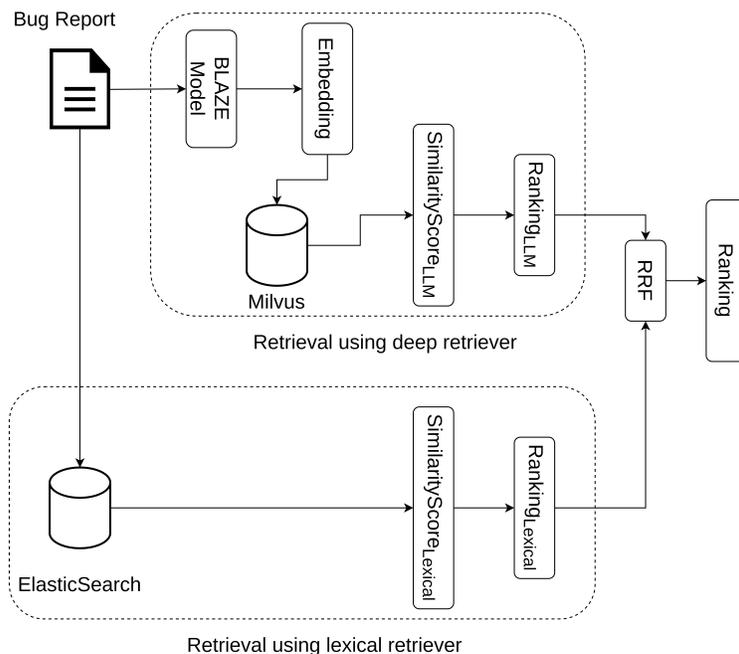


Figure 7.5: Retrieval phase of BLAZE (Phase C).

source code files—as outlined in [Phase A](#)—each chunk is transformed into an embedding using the BLAZE model. These embeddings are then indexed in Milvus, an open-source vector database specifically optimized for storing and retrieving embedding vectors in artificial intelligence applications.

Phase C. Retrieval Pipeline

Figure 7.5 illustrates the retrieval phase of BLAZE. We leverage a combination of lexical and deep retrievers along with the fine-tuned BLAZE model. As shown in the figure, upon the arrival of a bug report, we query the Milvus and ElasticSearch indices to pinpoint the relevant source code files. We then integrate the rankings from both indices using a technique known as Reciprocal Rank Fusion (RRF). Next, we provide a detailed description of these steps.

Retrieval using deep retriever. Deep retrievers have demonstrated superior performance in ranking tasks compared to traditional vocabulary matching approaches [1]. To use a deep retriever, we first calculate the embedding of the bug report using our fine-tuned BLAZE model. Then, we retrieve the embedding of all the source code files from Milvus,

a vector index we built during offline indexing. Milvus delivers the top K most similar chunks to the bug report embedding. We use cosine similarity to assess the similarity between embeddings. Finally, we sort the source code files based on decreasing similarity.

Retrieval using lexical retriever. In the lexical retrieval process, we query the ElasticSearch index with the bug report, which returns similarity scores for each source code file based on lexical matches. We use these scores to rank the source code files, leveraging ElasticSearch text searching capabilities to localize files potentially containing the bug. By directly using the bug report as a query, we ensure that the retrieval is tightly focused on the textual content relevant to the bug.

Reciprocal rank fusion. While deep retrievers demonstrate superior performance in capturing semantic similarity, they sometimes struggle with lengthy documents and may face challenges such as domain shifts when applied across different projects [202, 325]. These shifts can lead to outdated understandings of new domains. Conversely, lexical retrievers are immune to domain shifts but can encounter difficulties with vocabulary mismatches due to their lack of a semantic understanding. To address these limitations and harness the strengths of both retrieval types, we employ Reciprocal Rank Fusion (RRF) [62]. RRF integrates the rankings from both the lexical and deep retrievers by considering the position of each source code file in the rankings rather than merging their numerical scores. This method, supported by prior research [62, 28, 49], has proven to be a reliable and powerful technique for creating robust ensembles that enhance overall retrieval performance.

Given a bug report r , a source code file s , and a set of retrievers M , where $\pi^m(r, s)$ represents the ranking by retriever m for source code file s and bug report r , the RRF score is defined as follows:

$$RRF(r, s, M) = \sum_{m \in M} \frac{1}{k + \pi^m(r, s)} \quad (7.3)$$

Here, k is a hyperparameter. Based on prior studies [62, 49], $k = 60$ has been found to provide the highest mean average precision (MAP). We also experimented with k values ranging $\epsilon[50, 70]$ in the BeetleBox dataset and confirmed that $k = 60$ continues to yield the highest MAP.

7.3 Experimental Design

In this section, we present the Research Questions (RQs) we tackle in our study, the process of our dataset collection, the subject baselines, and the performance metrics employed to evaluate the performance of BLAZE.

7.3.1 Research Questions

Below, we present our RQs by explaining the motivation behind each one.

RQ1. *How effective is BLAZE compared to state-of-the-art cross-project bug localization tools?*

A plethora of bug localization tools have been proposed (e.g., [390, 344, 121, 388]), with cross-project tools (e.g., [125, 388]) being particularly effective. However, these tools still require project-specific additional training. Our proposed approach, BLAZE, is designed to function without such a requirement. This RQ evaluates how well BLAZE performs compared to state-of-the-art cross-project bug localization tools.

RQ2. *How effective is BLAZE compared to state-of-the-art embedding-based bug localization tools?*

Embedding-based techniques, which assess document similarity, are extensively used for bug localization. These techniques differ from traditional bug localizers that provide binary predictions by producing embeddings and employing metrics like cosine distance to evaluate if a source code file contains bugs. Their ability to contextualize associations between bug reports and source code enhances their effectiveness. This RQ explores how well BLAZE performs compared to these embedding-based tools.

7.3.2 Dataset Curation of BeetleBox

We have compiled an extensive multi-language, multi-project bug localization dataset. Previous research [363, 383, 224, 73] has proposed various datasets for bug localization, but these have numerous shortcomings, such as support for only one programming language and a limited range of projects. Additionally, some datasets, such as Bench4BL [173], contain inaccuracies in their ground truth data [145]. To address these challenges, we introduce BEETLEBOX. Below, we detail the process we follow for curating BeetleBox.

- **Selection of repositories.** We use the GitHub API search feature to collect repository names. We search for repositories written in the top five programming languages,

Java, Python, C++, JavaScript, and Go, as ranked in the StackOverflow Survey.² We select the top 20 repositories for each language, sorting them in descending order by their star ratings and most recent updates. This step guarantees that the repositories are actively contributed to and popular within the open-source community.

- **Linking pull-requests and issues.** We gather ‘closed’ issues for each repository and their corresponding fixing Pull Request (PR). We identify these PRs by using predefined GitHub keywords.³
- **Filtration of issues.** An issue may have several PRs, but typically only one is merged. To minimize errors in our ground truth, we filter out issues with the ‘duplicate’ tag and verify using GitHub API that each issue is linked to a single, merged PR. We initially started with 100 repositories. However, in certain projects, we found no issues meeting all the criteria. Furthermore, to avoid potential data leakage [78], we have to filter out 63 repositories. In the end, we are left with 29 repositories across five programming languages.

For each bug report in **BeetleBox**, the dataset, we include the bug status, repository name, repository URL, issue ID, a list of files modified during the fix, the title of the bug report, its body, URL link to the merged PR, URL link to the issue, SHA values of before and after the fix, the date and time the bug was reported, and the date and time of the fixing commit. Overall, **BeetleBox** comprises 26,321 bugs extracted from 29 projects.

Finally, to assess the accuracy of the ground truth in **BeetleBox**, we use the method outlined by Kim et al. [145]. Our manual analysis reveals that **BeetleBox** has an error rate of 0.06%, with a 95% confidence level.

7.3.3 Evaluation Benchmarks and Baselines

Benchmarks. In addition to our **BeetleBox** dataset, we employ SWE-Bench [136], and Ye et al. [363]. Similar to **BeetleBox**, Ye et al. dataset is also recognized for its low error rate in its ground truth. While **BeetleBox** and Ye et al. are solely bug localization datasets, SWE-Bench is a recent dataset that is a benchmarking suite for bug localization as well as automatic patch generation.

²<https://survey.stackoverflow.co/2022/#technology-most-popular-technologies>

³<https://docs.github.com/en/issues/tracking-your-work-with-issues/linking-a-pull-request-to-an-issue>

Table 7.1: Statistics on the used datasets, showing the distribution of bugs across training and testing, categorized by language for each dataset.

Dataset	Purpose	Language	# Bugs
BeetleBox	train	c++	3,868
		go	758
		java	3,369
		javascript	1,974
		python	3,215
	test	c++	4,783
		go	400
		java	2,270
		javascript	3,085
		python	2,599
SWE-Bench [136]	test	python	2,294
Ye et al. [363]	test	java	16,314

In our methodology, we fine-tune the language model using the training split of `BeetleBox`. For evaluation, we use the test splits of `BeetleBox`, `SWE-Bench`, and the dataset from Ye et al. Table 7.1 provides brief statistics of these three datasets.

Studied baselines. We compare `BLAZE` with state-of-the-art cross-project bug localization tools, `TRANP-CNN` [125] and `CooBa` [388], which both use few-shot learning for cross-project adaptation. `TRANP-CNN`, a CNN-based deep learning tool, and `CooBa`, which uses dual encoders, train on a source project and are fine-tuned on a target project with limited samples. We also include `BugLocator` [383] as a baseline in our evaluation. Although not designed for cross-project use, `BugLocator`, like `BLAZE`, is an information retrieval-based tool that does not require additional training for deployment in new projects. The characteristics make it a relevant comparison in our evaluation.

In addition, we evaluate the performance of `BLAZE` compared to embedding-based bug localizers, namely `SemanticCodeBERT` [75] and `FBL-BERT` [54]. Both are BERT-based models, where `SemanticCodeBERT` incorporates semantic flow graph information into its

training, and FBL-BERT incorporates changeset data. Recent advancements in generative AI have also introduced models like *openai-embedding* [235], which have demonstrated superior performance in tasks involving natural language and code. Therefore, OpenAI embedding is also included in our baseline for comparison.

7.3.4 Configuration Setup

We fine-tune the BLAZE model in a distributed setting with eight cores, 128GB of RAM, and fourteen Nvidia V100 32GB GPUs. We conduct inference on the same server using an Nvidia V100 32 GB GPU, which allows for fast and efficient predictions. We use PyTorch v.2.1.2, PyTorch-lightning v2.2.0, and the HuggingFace library v.4.36.2 to fine-tune our model. We also use Elasticsearch v8.7.1 and Milvus v2.3.4 for efficient similarity search. We fine-tune our language model on the `BeetleBox` dataset for 15 epochs, with batches of size 80 and a decaying learning rate starting at 2E-07, as recommended in prior work [67, 54]. More details about the hyperparameters are available in our online Appendix [18].

7.4 Results

In this section, we present our results per each RQ.

RQ1: Comparison to Cross-Project Tools.

Approach. We train our model using `BeetleBox` on 13K bugs from five languages, covering 13 projects. Our evaluation dataset contains 31,745 bugs from five languages covering 29 real-world software projects. Following prior studies [136, 80], `BeetleBox` uses a completely different set of projects for training and testing splits. As previously presented in Table 7.1, the SWE-Bench and Ye et al. datasets are used only for testing.

Result. Table 7.2 presents the performance of BLAZE compared to TRANP-CNN, CooBa, and BugLocator, across the studied datasets. Overall, BLAZE demonstrates its highest performance in the SWE-Bench dataset. Conversely, its performance is reduced in the Ye et al. dataset, as indicated by both Top 1 and MAP scores. Nonetheless, **BLAZE consistently outperforms comparative baselines in this dataset and across all evaluated datasets.**

Table 7.2: Comparative performance of BLAZE and cross-project bug localization tools.

Dataset	Model Name	Top 1	Top 5	Top 10	MAP	MRR
BeetleBox	BLAZE	0.18	0.37	0.45	0.22	0.27
	TRANP-CNN	0.08	0.13	0.19	0.09	0.13
	CooBa	0.06	0.1	0.14	0.08	0.09
	BugLocator	0.06	0.10	0.15	0.08	0.12
SWE-Bench	BLAZE	0.29	0.68	0.79	0.43	0.45
	TRANP-CNN	0.13	0.28	0.42	0.26	0.34
	CooBa	0.12	0.25	0.38	0.21	0.3
	BugLocator	0.1	0.14	0.21	0.17	0.21
Ye et al.	BLAZE	0.16	0.35	0.44	0.21	0.26
	TRANP-CNN	0.09	0.17	0.23	0.14	0.15
	CooBa	0.08	0.13	0.19	0.09	0.13
	BugLocator	0.09	0.11	0.16	0.07	0.12

In the **BeetleBox dataset**, BLAZE surpasses TRANP-CNN, with improvements ranging between 107% to 144.44%. Specifically, BLAZE achieves a 125% increase in Top 1 accuracy, a 144.44% increase in MAP, and a 107.69% in MRR. When compared to CooBa, BLAZE shows substantial gains: 200% in Top 1, 175% in MAP, and 200% in MRR. Against BugLocator, BLAZE shows improvements of 200% in Top 1, 175% in MAP, and 125% in MRR.

In the **SWE-Bench dataset**, BLAZE performs better than TRANP-CNN with increases between 32% and 123% across various metrics. The enhancements include a 123% improvement in Top 1, 65% in MAP, and 32% in MRR. Against CooBa, BLAZE shows a 141% increase in Top 1, 104% in MAP, and 50% in MRR. Compared to BugLocator, BLAZE shows improvements of 190% in Top 1, 52.9% in MAP, and 61.9% in MRR.

In the **Ye et al. dataset**, BLAZE shows an improvement over TRANP-CNN with gains of 77% in Top 1, 50% in MAP, and 73% in MRR. In comparison to CooBa, BLAZE shows increases of 100% in Top 1, 133% in MAP, and 100% in MRR. Against BugLocator, the model posts gains of 77.7% in Top 1, 200% in MAP, and 25% in MRR.

Additionally, across the studied programming languages, BLAZE achieves Top 1 scores of

Table 7.3: Comparative performance of BLAZE and embedding-based bug localization tools.

Dataset	Model Name	Top 1	Top 5	Top 10	MAP	MRR
BeetleBox	BLAZE	0.18	0.37	0.45	0.22	0.27
	OpenAI	0.13	0.26	0.32	0.1	0.19
	SemanticCodeBERT	0.11	0.20	0.31	0.13	0.21
	FBL-BERT	0.12	0.23	0.30	0.16	0.19
SWE-Bench	BLAZE	0.29	0.68	0.79	0.43	0.45
	OpenAI	0.27	0.63	0.72	0.36	0.39
	SemanticCodeBERT	0.23	0.5	0.64	0.31	0.33
	FBL-BERT	0.20	0.57	0.51	0.31	0.30
Ye et al.	BLAZE	0.16	0.35	0.44	0.21	0.26
	OpenAI	0.17	0.36	0.43	0.19	0.25
	SemanticCodeBERT	0.13	0.24	0.30	0.17	0.22
	FBL-BERT	0.10	0.24	0.28	0.14	0.18

0.18 in C++, 0.36 in Go, 0.26 in Java, 0.21 in JavaScript, and 0.3 in Python. These results show an improvement of 50.9% to 135.6% compared to TRANP-CNN, with the highest improvement observed in JavaScript and the lowest in Python. The trend is similar to that of CooBa, with improvements ranging from 82% to 162%, with the highest increase observed in C++ and the lowest in Python. Against BugLocator, the improvement ranges from 87% to 160%, with the highest improvement in JavaScript and the lowest in Python.

RQ2: Comparison to Embedding-Based Tools.

Approach. We examine three embedding-based bug localization tools: SemanticCodeBERT [75], FBL-BERT [54], and OpenAI [235]. For SemanticCodeBERT⁴ and FBL-BERT⁵, we use their publicly available replication package to train the models and produce embeddings. For OpenAI, we obtain embeddings for both bug reports and source code files

⁴<https://github.com/IBM/semanticflowgraph>

⁵<https://anonymous.4open.science/r/fbl-bert-700C/README.md>

using the OpenAI API.⁶ We employ cosine similarity to rank the source code files based on their similarity scores, facilitating the identification of the buggy files.

Result. Table 7.3 illustrates the performance of BLAZE relative to the studied embedding-based tools. From the table, **we observe trends similar to those observed in Table 7.2.** We also observe that the performance across all models is strongest in the SWE-Bench dataset, which comprises Python projects, and weakest in the Ye et al. dataset, featuring Java projects.

In the **BeetleBox dataset.**, BLAZE demonstrates superior performance to OpenAI, with improvements ranging from 38% in Top 1 accuracy to 120% in MAP and 42.11% in MRR. Similarly, when compared to SemanticCodeBERT, BLAZE achieves substantial gains, with increases of 63% in Top 1, 69% in MAP, and 28% in MRR. It also shows notable enhancements against FBL-BERT, including a 50% rise in Top 1 and improvements in MAP and MRR by 37% and 42%, respectively.

The performance trend continues in the **SWE-Bench dataset**, where BLAZE outperforms OpenAI with gains from 7% in Top 1 to 19% in MAP and 15% in MRR. It also outperforms SemanticCodeBERT with a 26% increase in Top 1 and even greater enhancements in MAP and MRR. Against FBL-BERT, BLAZE shows substantial improvements of 45% in Top 1, 38% in MAP, and 50% in MRR.

In the **Ye et al. dataset**, BLAZE slightly lags behind OpenAI in Top 1 by a margin of 5%, but it still achieves better performance in MAP and MRR by 10% and 4%, respectively. It demonstrates enhancements over SemanticCodeBERT, improving by 23% in both Top 1 and MAP, and 18% in MRR. Similarly, against FBL-BERT, BLAZE achieves substantial performance increases of 60% in Top 1, 50% in MAP, and 44% in MRR.

When analyzing the performance of BLAZE across the studied programming languages, we find that BLAZE consistently outperforms the studied models in Top-1 scores across most programming languages. The exception is Go, where OpenAI slightly outperforms BLAZE in Go projects. We find that Go projects contain only 758 bugs, the fewest in our dataset. Additionally, the unique concurrency model and error-handling patterns of Go may pose challenges for BLAZE and contribute to lower performance in this language. Moreover, BLAZE surpasses OpenAI with improvements ranging from 14.24% to 16.86% in all languages except Go, where it lags by 0.66%. Compared to OpenAI, BLAZE achieves the highest improvement in Python and the lowest improvement in C++. Compared to SemanticCodeBERT, BLAZE shows consistent improvement between 15% and 41%, with the highest being observed in JavaScript and the lowest in Go. The trend continues when

⁶<https://openai.com/index/openai-api/>

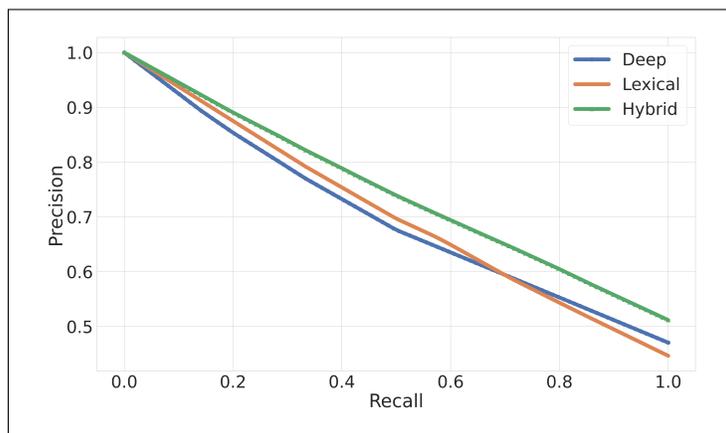


Figure 7.6: Precision recall curves of the retrievers.

compared with FBL-BERT, where improvements range from 34.1% to 67.5%, peaking in Java and being lowest in JavaScript.

7.5 Ablation Study

This section presents an ablation study to assess the influence of individual components in the BLAZE pipeline on its overall performance. Key strategies contributing to the effectiveness of BLAZE include (1) a combined pipeline of lexical and deep retrievers, (2) learning from hard examples, and (3) dynamic chunking. For all the experiments in this ablation study, we use `BeetleBox` as it contains bugs from five different languages. Next, we detail the results of the ablation study to evaluate the impact of each component.

7.5.1 Impact of hybrid retrieval technique

In BLAZE, we use a hybrid technique by combining outputs from two types of retrievers: lexical and deep. The lexical retriever is typically useful at finding exact matches and managing simple searches, while the deep retriever is better suited for understanding complex queries through their semantics. We assess their combined performance with a precision-recall graph, as shown in Figure 7.6. The figure demonstrates how accurately BLAZE identifies buggy source code files at various decision thresholds, with higher precision and recall indicating better performance.

Table 7.4: Performance comparison between Lexical, Deep, and Hybrid Retriever.

Retriever Type	Top 1	Top 5	Top 10	MAP	MRR
Lexical	0.07	0.12	0.19	0.11	0.13
Deep	0.12	0.25	0.31	0.14	0.18
Hybrid (BLAZE)	0.18	0.37	0.45	0.22	0.27

Figure 7.6 shows that the lexical retriever initially has high precision but low recall, indicating it identifies relevant files well but may miss others. As it retrieves more files, precision drops. Conversely, the deep retriever’s precision increases as it processes more files, particularly at higher recall levels. By combining both retrievers in the BLAZE pipeline (as denoted ‘Hybrid’ in Figure 7.6), we achieve a balance that enhances overall results.

Furthermore, we analyze the performance of each retriever and their combined effect in the BLAZE pipeline on the BeetleBox dataset. From Table 7.4, we observe that the deep retriever outperforms the lexical retriever, showing improvements of 71% in Top 1, 27% in MAP, and 38% in MRR. Combining the retrievers further enhances performance, showing improvements of 45% to 57% over the deep retriever alone. This result demonstrates that combining both retriever types mitigates their individual weaknesses and enhances the overall performance of BLAZE.

7.5.2 Impact of fine-tuning

In BLAZE, we fine-tune a GPT-based model [371] to better target hard-to-localize bugs by adjusting how closely buggy (positive files) and non-buggy (negative files) source code relate to bug reports (see Phase A). We assess the impact of this fine-tuning by comparing the performance of the model at different stages: epoch 0 (CodeSage), epoch 5, and epoch 15 (BLAZE). We use t-SNE [312], a method that simplifies complex data into two or three dimensions for easier visualization, to generate and display embeddings for a specific bug report (i.e., *Apache Dubbo-1216*) and its corresponding source code.

Figure 7.7 shows that the ability of the CodeSage model to distinguish between buggy and non-buggy files in the analyzed bug report (i.e., *Apache Dubbo-1216*) evolves over time. At the start (epoch 0), the model struggles to differentiate, with overlapping positive and negative chunks. By epoch 5, there is a clear improvement: positive chunks start clustering

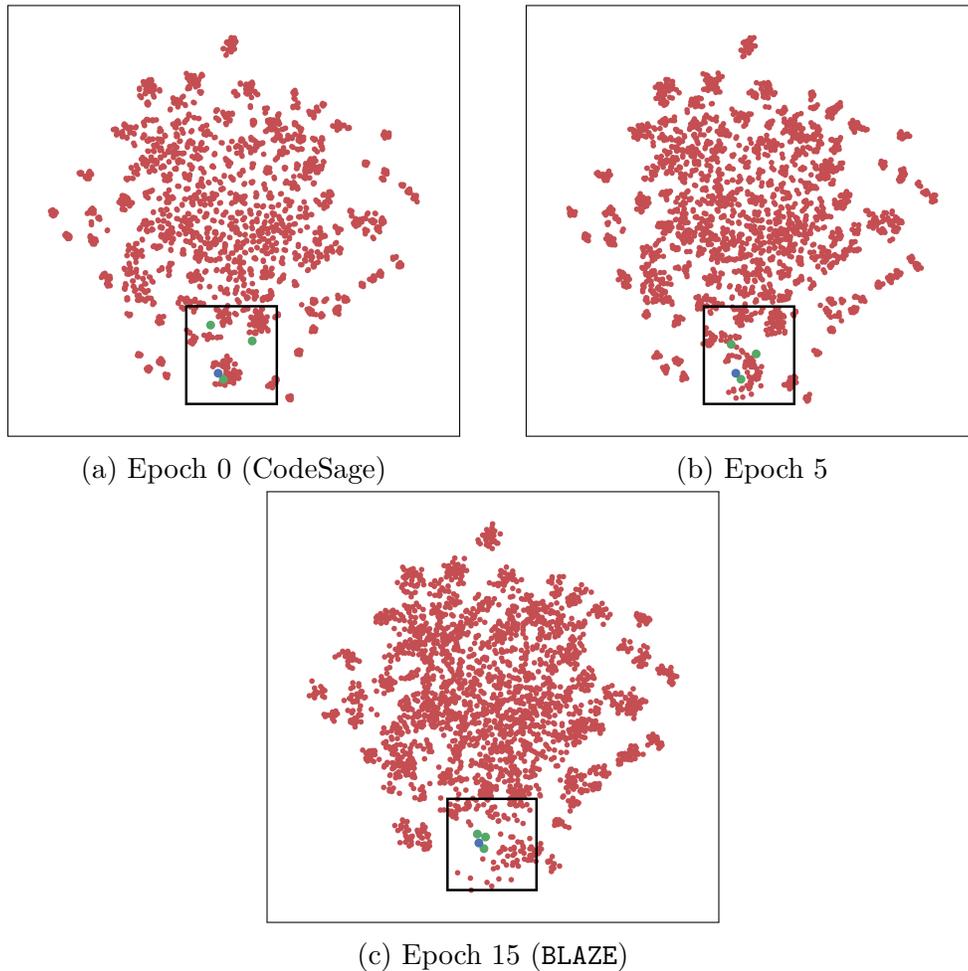


Figure 7.7: Scatter plots showing the chunks of the buggy and non-buggy files associated with the Apache Dubbo-1216 bug report. ● denotes bug report, ● denotes chunks from buggy files, and ● denotes chunks from non-buggy files.

more closely, and negative chunks spread out, although some overlap remains. By epoch 15, the model has made substantial progress; positive chunks are well-grouped, and no negative chunks are closer than the positives, indicating successful fine-tuning.

Also, we evaluate the CodeSage model on the `BeetleBox` dataset. Table 7.5 shows the results of the CodeSage and BLAZE model with different types of retrievers. For instance, `CodeSageDeep` reflects the performance with a deep retriever, while `CodeSageHybrid` indicates the results using a hybrid retriever combining deep and lexical retriever. Using only the

Table 7.5: Performance of CodeSage vs. BLAZE.

Retriever	Type	Top 1	Top 5	Top 10	MAP	MRR
CodeSage	Deep	0.11	0.21	0.28	0.12	0.16
CodeSage	Hybrid	0.17	0.32	0.42	0.19	0.24
BLAZE	Deep	0.12	0.25	0.31	0.14	0.18
BLAZE	Hybrid	0.18	0.37	0.45	0.22	0.27

deep retriever, BLAZE achieves a performance improvement of 6.9%–12.5% over CodeSage, including 9.1% for Top1, 16.7% for MAP, and 12.5% for MRR. The hybrid technique used by the BLAZE pipeline leads to even more enhancements, ranging from 5.8% to 15.7%, with gains of 5.8% in Top1, 15.8% in MAP, and 12.5% in MRR. These results support that fine-tuning the model on challenging (hard) examples improves its effectiveness.

7.5.3 Impact of dynamic chunking

In BLAZE, we use dynamic chunking to manage the issue of limited context size while reducing continuity loss. Previous research [271, 130] has used methods like sliding windows and static chunking. The sliding window method overlaps chunks to maintain continuity but complicates similarity calculations due to these overlaps. Static chunking, on the other hand, avoids overlaps but can lead to a loss of continuity. Our approach uses dynamic programming to find breakpoints that reduce continuity loss without shrinking the window size. We trained and evaluated BLAZE using all three chunking methods to see the effects of each.

Table 7.6 presents a performance comparison of various chunking techniques. For instance, BLAZE_{Static + Deep} represents the performance of static chunking paired with the deep retriever. The table shows that BLAZE_{Dynamic + Deep} improves performance by 5.5–16% over static chunking and by 3.23–8.33% over the sliding window method. Within the hybrid technique of BLAZE pipeline, dynamic chunking shows a performance increase of 7–18% compared to static chunking and 2–7.41% compared to sliding window.

Table 7.6: Static vs. Sliding window vs. Dynamic chunking.

Model	Top 1	Top 5	Top 10	MAP	MRR
BLAZE _{Static + Deep}	0.11	0.22	0.26	0.13	0.17
BLAZE _{Sliding + Deep}	0.11	0.24	0.3	0.13	0.17
BLAZE _{Dynamic + Deep}	0.12	0.25	0.31	0.14	0.18
BLAZE _{Static + Hybrid}	0.15	0.33	0.37	0.18	0.25
BLAZE _{Sliding + Hybrid}	0.17	0.36	0.42	0.21	0.25
BLAZE _{Dynamic + Hybrid}	0.18	0.37	0.45	0.22	0.27

7.6 Threats to Validity

Below, we discuss the threats to the validity of our study.

Internal validity. One threat to internal validity is the hyperparameters we used to train the models. To mitigate this, we use the same parameters used by several prior studies that use large language models [67, 54, 143, 156].

External validity. A threat to the external validity of our study lies in the generalizability of the results. We assess cross-project capabilities using separate projects for training and testing. Although the choice of test projects may affect performance, this approach aligns with previous studies that evaluate zero-shot and cross-project capabilities [80, 136]. Additionally, our evaluation of 31,745 real-world bugs ensures that our findings hold, even with different project selections, due to the large scale of our dataset. To address potential selection bias in the `BeetleBox` dataset, we collected project repositories based on star ratings and recent updates, ensuring that the repositories are actively contributed to and popular within the open-source community. This approach aims to include high-quality, widely used projects across the selected top five programming languages. Additionally, due to the absence of a replication package, we replicated TRANP-CNN and CooBA based on their descriptions in the original study, which may lead to slight performance deviations. Nevertheless, after experimenting with various hyperparameters, we selected a set that achieves comparable performance to that reported in the original study.

Construct validity. Our selected evaluation metrics can threaten the construct validity

of the study. We used Top N, MAP, and MRR as evaluation metrics. These are well-known evaluation measures adopted in prior studies which are widely used when evaluating the performance of bug localization tools that rely on information retrieval techniques [54, 187, 363, 383].

7.7 Conclusion

In this study, we introduce **BLAZE**, a bug localization tool that operates without project-specific training, facilitating cross-project and cross-language bug localization. **BLAZE** employs a dynamic chunking and hard example learning methodology to model lengthy source code files. To support cross-project and cross-language bug localizers, we curate **BeetleBox**—one of the largest (if not *the* largest) cross-project and cross-language bug localization datasets. We evaluate **BLAZE** with 31,745 bug reports from 34 real-world projects, showing up to a 100% improvement in the Top 1 metric against cross-project localizers and up to a 60% improvement over modern large language model approaches. Our results suggest that the use of dynamic chunking and hard example learning is key in enhancing the performance of bug localization tools.

7.8 Chapter Summary

In this chapter, we tackle two key challenges in embedding-based bug localization techniques: (1) limited context window and (2) mapping accuracy. We present an approach that enhances bug localization performance without the need for project-specific training. The improved accuracy directly addresses developers’ concerns about the low accuracy of existing bug localization tools. Additionally, the elimination of retraining requirements resolves another major concern—potential **IP** leakage—making the adoption of bug localization tools more feasible for developers.

Chapter 8

Conclusion and Future Work

Bug localization tools automate the labor-intensive process of identifying bugs within a software codebase. In addition to assisting with bug localization, these tools can serve as valuable training aids for new developers.

Although these tools theoretically promise to accelerate the software development process, their practical adoption rates often fall short of expectations. In this thesis, we empirically examine developers' expectations from bug localization tools, identifying key challenges through user surveys and interviews. These insights inform the development of solutions that enhance the tools' maintainability and scalability.

One of the primary focuses of this thesis is understanding and solving the issues that affect the adoption of bug localization tools. Our empirical analysis provides a deeper understanding of developers' concerns, with accuracy emerging as a key practical challenge. To address this, we developed BLAZE and RLocator, tools specifically designed to improve accuracy, a factor critical for real-world adoption. We formulated and evaluated the following thesis statement:

Thesis Statement

Practical challenges in bug localization tools, such as low accuracy, hinder their widespread adoption by developers. Addressing these challenges through targeted solutions can not only improve the precision and accuracy of these tools but also enhance their appeal and usability, driving greater adoption.

We conducted a mixed-method study, combining a survey of 95 respondents and interviews with six developers to empirically investigate their expectations. The findings

provided valuable insights into how developers prioritize factors such as accuracy and concerns over IP leakage. Additionally, we explored alternative techniques, including [RL](#) and [LLM](#), to address two major challenges: reliance on labeled data and the limited generalizability of current bug localization tools. These methods not only improved tool accuracy but also mitigated issues like data leakage caused by the need for retraining. Therefore, our studies validate the thesis statement.

8.1 Contributions and Findings

- **Barriers in Adopting of Bug Localization Tools:** Despite developers' willingness to adopt bug localization (BL) tools, significant barriers persist due to concerns about accuracy, performance, and intellectual property (IP) leakage. These issues highlight the need for tools that are better aligned with developers' expectations and offer enhanced security and reliability (Chapter 4).
- **Benefits of Bug Localization Tools:** Bug localization tools are particularly valuable for junior developers, providing an effective training resource. These tools help less experienced developers improve their skills and understanding, accelerating their learning process (Chapter 4).
- **Impact of Project-Specific Training on Model Performance:** Training bug localization models with project-specific data significantly enhances performance. Models trained on familiar datasets show up to a 76% improvement compared to those trained on more general datasets, highlighting the importance of tailoring models to specific project environments (Chapter 5).
- **Performance Variability in Longer Sequence Transformers:** Longer sequence transformers, such as LongCodeBERT, demonstrate improved performance in bug localization tasks. However, these gains are inconsistent across different projects, indicating that resource allocation and project-specific factors need to be carefully considered when using such models (Chapter 5).
- **Reinforcement Learning to Reduce Dependency on Labeled Data:** Modeling bug localization as a reinforcement learning (RL) problem offers a flexible solution to reduce reliance on labeled data. This approach allows for more adaptive training, making the tools more versatile in varied development environments (Chapter 6).
- **Optimizing Performance Through Reinforcement Learning:** Directly optimizing performance metrics using reinforcement learning can significantly enhance

the effectiveness of bug localization tools, with improvements of up to 38% in key performance indicators (Chapter 6).

- **Contribution of a Large Cross-Language Dataset:** We have contributed the largest cross-language and cross-project bug localization dataset to date. This dataset includes over 26,000 bugs from 29 projects across five programming languages, providing a valuable resource for future research in the field (Chapter 7).
- **Improvements from Hard Example Learning:** The use of a hard example learning technique greatly improves performance, particularly in challenging scenarios. With up to 120% improvement in Top 1 accuracy, this approach demonstrates its effectiveness in tackling difficult bug localization tasks (Chapter 7).
- **Dynamic Chunking to Solve the Issue of Limited Context Window:** We propose Dynamic Chunking, a novel method to address the limitations of large language models (LLMs) related to context window size. This technique significantly enhances the ability to process long source code files while reducing continuity loss, making it a valuable tool for handling complex bug localization tasks (Chapter 7).

8.2 Prospects for Future Research

This thesis makes a contribution to understanding developers' expectations and the practical challenges involved in making bug localization tools more scalable. However, many questions remain unanswered, leaving room for further research. Below, we highlight several promising directions for future work.

8.2.1 Customization and Personalization of Bug Localization Tools

Given that BL tools can significantly aid junior developers, a promising direction is to explore how customizable and personalized versions of these tools could further enhance learning and productivity for specific user profiles. This research could explore how adjusting BL tools based on the experience level and coding style of developers impacts both efficiency and learning outcomes.

8.2.2 Enhancing Bug Localization Tools Through Project-Specific Insights

The substantial performance improvements observed when training models with project-specific data suggest the need for deeper investigation. Identifying the key characteristics that distinguish projects using the same programming language but operating in different domains (e.g., mobile apps vs. web servers) could aid in designing more generalized bug localization tools.

8.2.3 Expanding the Application of Reinforcement Learning in Bug Localization

Reinforcement learning (RL) has demonstrated the potential to minimize reliance on labeled data. Future research should explore the application of advanced RL techniques, including the use of continuous action spaces and the identification of optimal reward functions, to develop more flexible and adaptable bug localization solutions.

8.2.4 Explainable Reinforcement Learning for Bug Localization

An important area for future exploration is the development of explainable reinforcement learning (XRL) within bug localization tools. By generating transparent explanations for why the RL agent identifies a particular section of code as the likely source of a bug, developers can better trust and understand the tool’s recommendations. This approach could involve visualizing decision pathways, highlighting features the agent considered or generating natural language explanations. Such improvements would enhance the interpretability of RL-based tools, fostering greater developer confidence and accelerating adoption.

8.2.5 Investigating Bug Propagation Across Projects

A promising area for future research is the study of bug propagation across different projects that share similar codebases, libraries, or dependencies. By developing models that can trace how bugs in one project may propagate to others through shared components, bug localization tools could preemptively identify and address bugs in related projects. This approach would be particularly useful in large ecosystems where multiple projects rely on common frameworks or third-party libraries, helping to prevent the recurrence of the same bug in multiple places.

8.2.6 Investigating Multimodal Approaches for Bug Localization

Another promising direction for future work is to integrate multimodal approaches into bug localization tools. A more holistic understanding of the bug's context could be achieved by combining source code analysis with additional modalities, such as project documentation, diagrams, and developer communication logs. This multimodal approach could improve the precision and relevance of bug localization, especially in complex and collaborative projects.

References

- [1] Zahra Abbasiantaeb and Saeedeh Momtazi. Text-based question answering from information retrieval and deep neural network perspectives: A survey. *WIREs Data Mining and Knowledge Discovery*, 11(6), May 2021.
- [2] Rui Abreu, Peter Zoetewij, and Arjan J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. IEEE, September 2007.
- [3] Arash Afkanpour, Shabir Adeel, Hansenclever Bassani, Arkady Epshteyn, Hongbo Fan, Isaac Jones, Mahan Malihi, Adrian Nauth, Raj Sinha, Sanjana Woonna, Shiva Zamani, Elli Kanal, Mikhail Fomitchev, and Donny Cheung. BERT for long documents: A case study of automated ICD coding. In *Proceedings of the 13th International Workshop on Health Text Mining and Information Analysis (LOUHI)*, pages 100–107. Association for Computational Linguistics, December 2022.
- [4] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C. Shepherd. Software documentation: the practitioners’ perspective. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE ’20*. ACM, June 2020.
- [5] H. Agrawal, J.R. Horgan, S. London, and W.E. Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE’95*. IEEE Comput. Soc. Press.
- [6] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online, June 2021. Association for Computational Linguistics.

- [7] Zafarali Ahmed, Nicolas Le Roux, Mohammad Norouzi, and Dale Schuurmans. Understanding the impact of entropy on policy optimization. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 151–160. PMLR, 09–15 Jun 2019.
- [8] Shayan Akbar and Avinash Kak. SCOR: Source code retrieval with semantics and order. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, May 2019.
- [9] Shayan A. Akbar and Avinash C. Kak. A large-scale comparative evaluation of IR-based tools for bug localization. In *Proceedings of the 17th International Conference on Mining Software Repositories*. ACM, June 2020.
- [10] Oscar Alejo, Juan M. Fernandez-Luna, Juan F. Huete, and Ramiro Perez-Vazquez. Direct optimization of evaluation measures in learning to rank using particle swarm. In *2010 Workshops on Database and Expert Systems Applications*. IEEE, August 2010.
- [11] Mamdouh Alenezi and Khaled Almustafa. Empirical analysis of the complexity evolution in open-source software systems. *International Journal of Hybrid Information Technology*, 8(2):257–266, February 2015.
- [12] Wajdi Aljedaani and Yasir Javed. *Bug Reports Evolution in Open Source Systems*, page 63–73. Springer International Publishing, 2018.
- [13] Rafi Almhana and Marouane Kessentini. Considering dependencies between bug reports to improve bugs triage. *Automated Software Engineering*, 28(1), January 2021.
- [14] Rafi Almhana, Marouane Kessentini, and Wiem Mkaouer. Method-level bug localization using hybrid multi-objective search. *Information and Software Technology*, 131:106474, 2021.
- [15] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019.
- [16] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, January 2019.

- [17] Anonymous. Aligning programming language and natural language: Exploring design choices in multi-modal transformer-based embedding for bug localization. <https://zenodo.org/doi/10.5281/zenodo.6760333>, Jun 2023. [Accessed 09-06-2023].
- [18] Anonymous. Blaze: Bug localization in zero-shot environment. <https://zenodo.org/record/7897697>, Jun 2023. [Accessed 09-06-2023].
- [19] Anonymous. Rlocator: Reinforcement learning for bug localization. <https://zenodo.org/record/7591879>, Jun 2023. [Accessed 09-06-2023].
- [20] John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange - eclipse '05*. ACM Press, 2005.
- [21] David Azcona, Piyush Arora, Ihan Hsiao, and Alan F. Smeaton. user2code2vec: Embeddings for profiling students based on distributional representations of source code. *Proceedings of the 9th International Conference on Learning Analytics & Knowledge*, 2019.
- [22] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*, page 177–188, New York, NY, USA, 2016. Association for Computing Machinery.
- [23] Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel Briand. Reinforcement learning for test case prioritization. *IEEE Transactions on Software Engineering*, 48(8):2836–2856, August 2022.
- [24] Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *Proceedings of the 36th International Conference on Software Engineering, ICSE '14*. ACM, May 2014.
- [25] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning, 2016.
- [26] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.
- [27] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020.

- [28] Michael Bendersky, Honglei Zhuang, Ji Ma, Shuguang Han, Keith Hall, and Ryan McDonald. Rrf102: Meeting the trec-covid challenge with a 100+ runs ensemble, 2020.
- [29] Berkay Berabi, Alexey Gronskiy, Veselin Raychev, Gishor Sivanrupan, Victor Chibotaru, and Martin Vechev. Deepcode ai fix: Fixing security vulnerabilities with large language models, 2024.
- [30] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering - SIGSOFT '08/FSE-16*. ACM Press, 2008.
- [31] Ghazala Bilquise, Samar Ibrahim, and Sa'Ed M. Salhieh. Investigating student acceptance of an academic advising chatbot in higher education institutions. *Education and Information Technologies*, 29(5):6357–6382, August 2023.
- [32] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, August 2017.
- [33] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.
- [34] 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, 2017.
- [35] Tolga Bolukbasi, Kai-Wei Chang, James Y. Zou, Venkatesh Saligrama, and Adam Kalai. Man is to computer programmer as woman is to homemaker? debiasing word embeddings. *CoRR*, abs/1607.06520, 2016.
- [36] Irma van den Brandt, Floris Fok, Bas Mulders, Joaquin Vanschoren, and Veronika Cheplygina. Cats, not cat scans: a study of dataset similarity in transfer learning for 2d medical image classification, 2021.
- [37] Lutz Büch and Artur Andrzejak. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–104, 2019.

- [38] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *11th IEEE International Software Metrics Symposium (METRICS'05)*. IEEE, 2005.
- [39] 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)*. IEEE, May 2021.
- [40] Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Jacob Ginesin, Edward Berman, George Chakhnashvili, Anton Lozhkov, Carolyn Jane Anderson, and Arjun Guha. Can it edit? evaluating the ability of large language models to follow code editing instructions, 2023.
- [41] Partha Chakraborty, Rifat Shahriyar, Anindya Iqbal, and Amiangshu Bosu. Understanding the software development practices of blockchain projects: a survey. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '18*. ACM, October 2018.
- [42] Cecilia Ka Yuk Chan and Wenjie Hu. Students' voices on generative ai: perceptions, benefits, and challenges in higher education. *International Journal of Educational Technology in Higher Education*, 20(1), July 2023.
- [43] Ming-Wei Chang, Lev-Arie Ratinov, Dan Roth, and Vivek Srikumar. Importance of semantic representation: Dataless classification. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 830–835. AAAI Press, 2008.
- [44] Debarshi Chatterji, Jeffrey C. Carver, Beverly Massengil, Jason Oslin, and Nicholas A. Kraft. Measuring the efficacy of code clone information in a bug localization task: An empirical study. In *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, September 2011.
- [45] An Ran Chen, Tse-Hsun Chen, and Shaowei Wang. Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs. *IEEE Transactions on Software Engineering*, 48(8):2905–2919, August 2022.
- [46] Limin Chen, Zhiwen Tang, and Grace Hui Yang. Balancing reinforcement learning training experiences in interactive information retrieval. In *Proceedings of the 43rd*

International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, July 2020.

- [47] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.
- [48] Minmin Chen, Alex Beutel, Paul Covington, Sagar Jain, Francois Belletti, and Ed Chi. Top-k off-policy correction for a reinforce recommender system, 2018.
- [49] Tao Chen, Mingyang Zhang, Jing Lu, Michael Bendersky, and Marc Najork. Out-of-domain semantics to the rescue! zero-shot hybrid retrieval models. In *Lecture Notes in Computer Science*, pages 95–110. Springer International Publishing, 2022.
- [50] Tianqi Chen and Carlos Guestrin. XGBoost. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, August 2016.
- [51] Zimin Chen and Martin Monperrus. The remarkable role of similarity in redundancy-based program repair. *CoRR*, abs/1811.05703, 2018.
- [52] Zimin Chen and Martin Monperrus. A literature study of embeddings on source code. *CoRR*, abs/1904.03061, 2019.
- [53] 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)*. IEEE, December 2020.
- [54] Agnieszka Ciborowska and Kostadin Damevski. Fast changeset-based bug localization with bert. In *Proceedings of the 44th International Conference on Software Engineering, ICSE '22*. ACM, May 2022.

- [55] Agnieszka Ciborowska and Kostadin Damevski. Fast changeset-based bug localization with BERT. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, May 2022.
- [56] Agnieszka Ciborowska and Kostadin Damevski. Too few bug reports? exploring data augmentation for improved changeset-based bug localization, 2023.
- [57] Jürgen Cito, Isil Dillig, Vijayaraghavan Murali, and Satish Chandra. Counterfactual explanations for models of code. *CoRR*, abs/2111.05711, 2021.
- [58] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.
- [59] CodeScene. Next generation code analysis — CodeScene — codescene.com. <https://codescene.com/>. [Accessed 11-07-2024].
- [60] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International Conference on Software Maintenance*. IEEE, September 2013.
- [61] CoPilot. GitHub Copilot · Your AI pair programmer — github.com. <https://github.com/features/copilot>. [Accessed 11-07-2024].
- [62] Gordon V. Cormack, Charles L A Clarke, and Stefan Buettcher. Reciprocal rank fusion outperforms condorcet and individual rank learning methods. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*. ACM, jul 2009.
- [63] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight bug localization with AMPLE. In *Proceedings of the Sixth sixth international symposium on Automated analysis-driven debugging - AADEBUG'05*. ACM Press, 2005.
- [64] Lysandre Debut. Benchmarking transformers: Pytorch and tensorflow, Sep 2020.
- [65] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018*, page 423–433, New York, NY, USA, 2018. Association for Computing Machinery.

- [66] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [67] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.
- [68] Jacob Devlin, Jonathan Uesato, Rishabh Singh, and Pushmeet Kohli. Semantic code repair using neuro-symbolic transformation networks, 2017.
- [69] Georgios Digkas, Mircea Lungu, Paris Avgeriou, Alexander Chatzigeorgiou, and Apostolos Ampatzoglou. How do developers fix issues and pay back technical debt in the apache ecosystem? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, March 2018.
- [70] Ming Ding, Chang Zhou, Hongxia Yang, and Jie Tang. Cogltx: Applying bert to long texts. In *NeurIPS*, 2020.
- [71] Ming Ding, Chang Zhou, Hongxia Yang, and Jie Tang. Cogltx: Applying bert to long texts. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 12792–12804. Curran Associates, Inc., 2020.
- [72] Zishuo Ding, Heng Li, Weiyi Shang, and Tse-Hsun Peter Chen. Can pre-trained code embeddings improve model performance? revisiting the use of code embeddings in software engineering tasks. *Empirical Software Engineering*, 27(3), March 2022.
- [73] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, November 2011.
- [74] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *J. Softw. (Malden)*, 25(1):53–95, January 2013.
- [75] Yali Du and Zhongxing Yu. Pre-training code representation with semantic flow graph for effective bug localization. In *Proceedings of the 31th ACM Joint Meeting*

on *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, 2023.

- [76] Vasiliki Efstathiou and Diomidis Spinellis. Semantic source code models using identifier embeddings. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, May 2019.
- [77] Ahmed El-Deeb. What the software industry is measuring? *ACM SIGSOFT Software Engineering Notes*, 48(2):10–11, April 2023.
- [78] Aparna Elangovan, Jiayuan He, and Karin Verspoor. Memorization vs. generalization : Quantifying data leakage in NLP performance evaluation. In Paola Merlo, Jorg Tiedemann, and Reut Tsarfaty, editors, *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 1325–1335, Online, April 2021. Association for Computational Linguistics.
- [79] Achiam et al. Gpt-4 technical report, 2023.
- [80] Guodong Fan, Shizhan Chen, Cuiyun Gao, Jianmao Xiao, Tao Zhang, and Zhiyong Feng. Rapid: Zero-shot domain adaptation for code search with pre-trained models. *ACM Transactions on Software Engineering and Methodology*, 33(5):1–35, June 2024.
- [81] Jiajun Fan, Changnan Xiao, and Yue Huang. GDI: rethinking what makes reinforcement learning different from supervised learning. *CoRR*, abs/2106.06232, 2021.
- [82] Fan Fang, John Wu, Yanyan Li, Xin Ye, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. On the classification of bug reports to improve bug localization. *Soft Computing*, 25(11):7307–7323, March 2021.
- [83] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, 2020.
- [84] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings*. IEEE, 2003.
- [85] Juan Manuel Florez, Oscar Chaparro, Christoph Treude, and Andrian Marcus. Combining query reduction and expansion for text-retrieval-based bug localization. In

2021 *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, March 2021.

- [86] David Freedman, Robert Pisani, and Roger Purves. Statistics (international student edition). *Pisani, R. Purves, 4th edn. WW Norton & Company, New York, 2007.*
- [87] Scott Fujimoto, David Meger, and Doina Precup. Off-policy deep reinforcement learning without exploration. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2052–2062. PMLR, 09–15 Jun 2019.
- [88] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2022.
- [89] Frédéric Garcia and Emmanuel Rachelson. Markov decision processes. In *Markov Decision Processes in Artificial Intelligence*, pages 1–38. John Wiley & Sons, Inc., March 2013.
- [90] Ali Ghanbari, Deepak-George Thomas, Muhammad Arbab Arshad, and Hridesh Rajan. Mutation-based fault localization of deep neural networks. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, September 2023.
- [91] Ahmad Nauman Ghazi, Kai Petersen, Sri Sai Vijay Raj Reddy, and Harini Nekkanti. Survey research in software engineering: Problems and mitigation strategies. *IEEE Access*, 7:24703–24718, 2019.
- [92] Michael Glass, Alfio Gliozzo, Rishav Chakravarti, Anthony Ferritto, Lin Pan, G P Shrivatsa Bhargav, Dinesh Garg, and Avi Sil. Span selection pre-training for question answering. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2773–2782, Online, July 2020. Association for Computational Linguistics.
- [93] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.
- [94] Artyom V. Gorchakov, Liliya A. Demidova, and Peter N. Sovietov. A rule-based algorithm and its specializations for measuring the complexity of software in educational digital environments. *Computers*, 13(3):75, March 2024.

- [95] Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2015.
- [96] 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*. ACM, November 2016.
- [97] Yu Gu, Robert Tinn, Hao Cheng, Michael Lucas, Naoto Usuyama, Xiaodong Liu, Tristan Naumann, Jianfeng Gao, and Hoifung Poon. Domain-specific language model pretraining for biomedical natural language processing. *ACM Transactions on Computing for Healthcare*, 3(1):1–23, October 2021.
- [98] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation, 2022.
- [99] 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, 2021.
- [100] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1861–1870. PMLR, 10–15 Jul 2018.
- [101] C. Hait and G. Tasse. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. DIANE Publishing Company, 2002.
- [102] Jiaxuan Han, Cheng Huang, Siqi Sun, Zhonglin Liu, and Jiayong Liu. bjXnet: an improved bug localization model based on code property graph and attention mechanism. *Automated Software Engineering*, 30(1), March 2023.
- [103] Jacob A. Harer, Louis Y. Kim, Rebecca L. Russell, Onur Ozdemir, Leonard R. Kosta, Akshay Rangamani, Lei H. Hamilton, Gabriel I. Centeno, Jonathan R. Key, Paul M. Ellingwood, Marc W. McConley, Jeffrey M. Opper, Sang Peter Chin, and Tomo Lazovich. Automated software vulnerability detection with machine learning. *CoRR*, abs/1803.04497, 2018.
- [104] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

- [105] A.E. Hassan and R.C. Holt. Predicting change propagation in software systems. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004.
- [106] Foyzul Hassan, Na Meng, and Xiaoyin Wang. Uniloc: Unified fault localization of continuous integration failures. *ACM Transactions on Software Engineering and Methodology*, 32(6):1–31, September 2023.
- [107] Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015.
- [108] Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *ArXiv*, abs/1507.06527, 2015.
- [109] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web.* International World Wide Web Conferences Steering Committee, April 2017.
- [110] Steffen Herbold, Alexander Trautsch, and Fabian Trautsch. On the feasibility of automated prediction of bug and non-bug issues. *Empirical Software Engineering*, 25(6):5333–5369, September 2020.
- [111] 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)*. IEEE, May 2013.
- [112] Thomas Hirsch. A fault localization and debugging support framework driven by bug tracking data. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, October 2020.
- [113] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, November 1997.
- [114] Sebastian Hofstätter, Hamed Zamani, Bhaskar Mitra, Nick Craswell, and Allan Hanbury. Local self-attention over long text for efficient document retrieval. *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2020.
- [115] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*. ACM Press, 2007.

- [116] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. A deep dive into large language models for automated bug localization and repair, 2024.
- [117] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 328–339, Melbourne, Australia, July 2018. Association for Computational Linguistics.
- [118] Yujing Hu, Qing Da, Anxiang Zeng, Yang Yu, and Yinghui Xu. Reinforcement learning to rank in e-commerce search engine: Formalization, analysis, and application. *CoRR*, abs/1803.00710, 2018.
- [119] Ken Huang, Fan Zhang, Yale Li, Sean Wright, Vasam Kidambi, and Vishwas Manral. *Security and Privacy Concerns in ChatGPT*, page 297–328. Springer Nature Switzerland, 2023.
- [120] Xuxiang Huang, Chen Xiang, Hua Li, and Peng He. SBugLocator: Bug localization based on deep matching and information retrieval. *Mathematical Problems in Engineering*, 2022:1–14, August 2022.
- [121] Xuan Huo, Ming Li, and Zhi-Hua Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *IJCAI*, 2016.
- [122] Xuan Huo, Ming Li, and Zhi-Hua Zhou. Control flow graph embedding based on multi-instance decomposition for bug localization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):4223–4230, Apr. 2020.
- [123] Xuan Huo, Ming Li, Zhi-Hua Zhou, et al. Learning unified features from natural and programming languages for locating buggy source code. In *IJCAI*, volume 16, pages 1606–1612, 2016.
- [124] 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, July 2019.
- [125] 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, July 2021.

- [126] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2020.
- [127] IEEE. Iso/iec/ieee international standard - systems and software engineering–vocabulary. *ISO/IEC/IEEE 24765:2017(E)*, pages 1–541, 2017.
- [128] Md Rakibul Islam and Minhaz F. Zibran. What changes in where?: an empirical study of bug-fixing change patterns. *ACM SIGAPP Applied Computing Review*, 20(4):18–34, January 2021.
- [129] Robert Logan IV, Ivana Balazevic, Eric Wallace, Fabio Petroni, Sameer Singh, and Sebastian Riedel. Cutting down on prompts and parameters: Simple few-shot learning with language models. In *Findings of the Association for Computational Linguistics: ACL 2022*. Association for Computational Linguistics, 2022.
- [130] Maor Ivgi, Uri Shaham, and Jonathan Berant. Efficient Long-Text Understanding with Short-Text Models. *Transactions of the Association for Computational Linguistics*, 11:284–299, 03 2023.
- [131] Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. Unsupervised dense information retrieval with contrastive learning, 2021.
- [132] Peter Izsak, Moshe Berchansky, and Omer Levy. How to train BERT with an academic budget. *CoRR*, abs/2104.07705, 2021.
- [133] Sigma Jahan, Mehil B. Shah, and Mohammad Masudur Rahman. Towards understanding the challenges of bug localization in deep learning systems, 2024.
- [134] Sooyoung Jang and Hyung-Il Kim. Entropy-aware model initialization for effective exploration in deep reinforcement learning. *Sensors*, 22(15):5845, August 2022.
- [135] Bo Jiang, Pengfei Liu, and Jie Xu. A deep learning approach to locate buggy files. In *2020 IEEE 11th International Conference on Dependable Systems, Services and Technologies (DESSERT)*. IEEE, May 2020.
- [136] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

- [137] Jeff Johnson, Matthijs Douze, and Herve Jegou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, July 2021.
- [138] James M. Joyce. *Kullback-Leibler Divergence*, page 720–722. Springer Berlin Heidelberg, 2011.
- [139] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. Challenges and applications of large language models, 2023.
- [140] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code, 2020.
- [141] Paul Kassianik, Erik Nijkamp, Bo Pang, Yingbo Zhou, and Caiming Xiong. Bigissue: A realistic bug localization benchmark, 2022.
- [142] Yogita Khatri and Sandeep Kumar Singh. Cross project defect prediction: a comprehensive survey with its swot analysis. *Innovations in Systems and Software Engineering*, 18(2):263–281, January 2021.
- [143] Omar Khattab and Matei Zaharia. ColBERT. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, July 2020.
- [144] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering*, 39(11):1597–1610, 2013.
- [145] Misoo Kim and Eunseok Lee. Are datasets for information retrieval-based bug localization techniques trustworthy? *Empir. Softw. Eng.*, 26(3), May 2021.
- [146] Misoo Kim and Eunseok Lee. Are datasets for information retrieval-based bug localization techniques trustworthy?: Impact analysis of bug types on irbl. *Empirical Software Engineering*, 26(3), March 2021.
- [147] 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)*. IEEE, May 2007.
- [148] 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)*. IEEE, December 2020.

- [149] Yunho Kim, Seokhyeon Mun, Shin Yoo, and Moonzoo Kim. Precise learn-to-rank fault localization using dynamic and static features of target programs. *ACM Transactions on Software Engineering and Methodology*, 28(4):1–34, October 2019.
- [150] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [151] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2020.
- [152] Barbara A. Kitchenham and Shari L. Pfleeger. *Personal Opinion Surveys*, page 63–92. Springer London, 2008.
- [153] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, August 2013.
- [154] Denis Kocetkov, Raymond Li, Loubna Ben allal, Jia LI, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro Von Werra, and Harm de Vries. The stack: 3 TB of permissively licensed source code. *Transactions on Machine Learning Research*, 2023.
- [155] Pavneet Singh Kochhar, Yuan Tian, and David Lo. Potential biases in bug localization. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ACM, September 2014.
- [156] Pavneet Singh Kochhar, Yuan Tian, and David Lo. Potential biases in bug localization: do they matter? In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14*. ACM, September 2014.
- [157] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, July 2016.
- [158] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA '16*. ACM, July 2016.
- [159] Tim Koomen and Martin Pol. *Test Process Improvement: A Practical Step-by-Step Guide to Structured Testing*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999.

- [160] Herb Krasner. The cost of poor software quality in the us: A 2020 report. 2021.
- [161] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, March 1951.
- [162] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, December 2015.
- [163] 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)*. IEEE, nov 2015.
- [164] 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)*, pages 476–481. IEEE, 2015.
- [165] 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)*. IEEE, May 2017.
- [166] 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)*, pages 218–229. IEEE, 2017.
- [167] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. *CoRR*, abs/1909.11942, 2019.
- [168] Thomas D. LaToza and Brad A. Myers. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*. ACM Press, 2010.
- [169] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models. In *Proceedings of the 28th international conference on Software engineering*. ACM, May 2006.

- [170] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, ICSE06. ACM, May 2006.
- [171] Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. In *The Twelfth International Conference on Learning Representations*, 2024.
- [172] Tien-Duy B. Le, Richard J. Oentaryo, and David Lo. Information retrieval and spectrum based bug localization: better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, August 2015.
- [173] 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*, ISSTA 2018, page 61–72, New York, NY, USA, 2018. Association for Computing Machinery.
- [174] 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*, ISSTA '18. ACM, July 2018.
- [175] Yoonho Lee, Annie S. Chen, Fahim Tajwar, Ananya Kumar, Huaxiu Yao, Percy Liang, and Chelsea Finn. Surgical fine-tuning improves adaptation to distribution shifts, 2022.
- [176] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead. Does bug prediction support human developers? findings from a google case study. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, May 2013.
- [177] David D. Lewis. Naive (bayes) at forty: The independence assumption in information retrieval. In *Machine Learning: ECML-98*, pages 4–15. Springer Berlin Heidelberg, 1998.
- [178] Wei Li, Qingan Li, Yunlong Ming, Weijiao Dai, Shi Ying, and Mengting Yuan. An empirical study of the effectiveness of ir-based bug localization for large-scale industrial projects. *Empirical Software Engineering*, 27(2), January 2022.

- [179] Yansong Li, Zhixing Tan, and Yang Liu. Privacy-preserving prompt tuning for large language model services, 2023.
- [180] Yi Li, Shaohua Wang, and Tien Nguyen. Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, May 2021.
- [181] 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, October 2019.
- [182] 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, October 2019.
- [183] Yizhi Li, Zhenghao Liu, Chenyan Xiong, and Zhiyuan Liu. More robust dense retrieval with contrastive dual learning. In *Proceedings of the 2021 ACM SIGIR International Conference on Theory of Information Retrieval*. ACM, July 2021.
- [184] Zheng Li, Yonghao Wu, and Yong Liu. An empirical study of bug isolation on the effectiveness of multiple fault localization. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, July 2019.
- [185] Zhiqiang Li, Hongyu Zhang, Xiao-Yuan Jing, Juanying Xie, Min Guo, and Jie Ren. DSSDPP: Data selection and sampling based domain programming predictor for cross-project defect prediction. *IEEE Transactions on Software Engineering*, 49(4):1941–1963, April 2023.
- [186] Davis Liang, Peng Xu, Siamak Shakeri, Cicero Nogueira dos Santos, Ramesh Nallapati, Zhiheng Huang, and Bing Xiang. Embedding-based zero-shot retrieval through query generation, 2020.
- [187] Hongliang Liang, Dengji Hang, and Xiangyu Li. Modeling function-level interactions for file-level bug localization. *Empirical Software Engineering*, 27(7), October 2022.
- [188] 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.

- [189] Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten, and Thomas B. Schön. *Machine Learning: A First Course for Engineers and Scientists*. Cambridge University Press, March 2022.
- [190] Trond Linjordet and Krisztian Balog. Impact of training dataset size on neural answer selection models. In *Lecture Notes in Computer Science*, pages 828–835. Springer International Publishing, 2019.
- [191] Chao Liu, Xin Xia, David Lo, Zhiwe Liu, Ahmed E. Hassan, and Shanping Li. CodeMatcher: Searching code based on sequential semantics of important query words. *ACM Transactions on Software Engineering and Methodology*, 31(1):1–37, September 2021.
- [192] Chao Liu, Xin Xia, David Lo, Zhiwe Liu, Ahmed E. Hassan, and Shanping Li. CodeMatcher: Searching code based on sequential semantics of important query words. *ACM Transactions on Software Engineering and Methodology*, 31(1):1–37, January 2022.
- [193] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. SOBER. *ACM SIGSOFT Software Engineering Notes*, 30(5):286–295, September 2005.
- [194] 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.
- [195] 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.
- [196] Tie-Yan Liu. Learning to rank for information retrieval. *Foundations and Trends® in Information Retrieval*, 3(3):225–331, 2007.
- [197] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.
- [198] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

- [199] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, August 2021.
- [200] Mingming Lu, Yan Liu, Haifeng Li, Dingwu Tan, Xiaoxian He, Wenjie Bi, and Wendbo Li. Hyperbolic function embedding: Learning hierarchical representation for functions of source code in hyperbolic space. *Symmetry*, 11(2), 2019.
- [201] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.
- [202] Yi Luan, Jacob Eisenstein, Kristina Toutanova, and Michael Collins. Sparse, dense, and attentional representations for text retrieval. *Transactions of the Association for Computational Linguistics*, 9:329–345, 2021.
- [203] Lucia, David Lo, and Xin Xia. Fusion fault localizers. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14*. ACM, September 2014.
- [204] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017.
- [205] Yuanhua Lv and ChengXiang Zhai. Lower-bounding term frequency normalization. In *Proceedings of the 20th ACM international conference on Information and knowledge management - CIKM '11*. ACM Press, 2011.
- [206] Yuanhua Lv and ChengXiang Zhai. When documents are very long, BM25 fails! In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information - SIGIR '11*. ACM Press, 2011.
- [207] Dimitris Mamakas, Petros Tsotsi, Ion Androutopoulos, and Ilias Chalkidis. Processing long legal documents with pre-trained transformers: Modding LegalBERT and longformer. In *Proceedings of the Natural Legal Language Processing Workshop*

2022, pages 130–142, Abu Dhabi, United Arab Emirates (Hybrid), December 2022. Association for Computational Linguistics.

- [208] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, March 1947.
- [209] Nina Marwede, Matthias Rohr, André van Hoorn, and Wilhelm Hasselbring. Automatic failure diagnosis support in distributed large-scale software systems based on timing behavior anomaly correlation. In *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, March 2009.
- [210] Nestor Maslej, Loredana Fattorini, Raymond Perrault, Vanessa Parli, Anka Reuel, Erik Brynjolfsson, John Etchemendy, Katrina Ligett, Terah Lyons, James Manyika, Juan Carlos Niebles, Yoav Shoham, Russell Wald, and Jack Clark. Artificial intelligence index report 2024, 2024.
- [211] Lucas Maystre, Daniel Russo, and Yu Zhao. Optimizing audio recommendations for the long-term: A reinforcement learning perspective, 2023.
- [212] Tim Menzies, William Nichols, Forrest Shull, and Lucas Layman. Are delayed issues harder to resolve? revisiting cost-to-fix of defects throughout the lifecycle. *Empirical Software Engineering*, 22(4):1903–1935, November 2016.
- [213] Leo A. Meyerovich and Ariel S. Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications, SPLASH '13*. ACM, October 2013.
- [214] Tommi Mikkonen, Jukka K. Nurminen, Mikko Raatikainen, Ilenia Fronza, Niko Mäkitalo, and Tomi Männistö. Is machine learning software just software: A maintainability view. In *Software Quality: Future Perspectives on Software Engineering Quality*, pages 94–105. Springer International Publishing, 2021.
- [215] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.
- [216] Chris Mills, Gabriele Bavota, Sonia Haiduc, Rocco Oliveto, Andrian Marcus, and Andrea De Lucia. Predicting query quality for applications of text retrieval to software

- engineering tasks. *ACM Transactions on Software Engineering and Methodology*, 26(1):1–45, January 2017.
- [217] Nima Miryeganeh, Sepehr Hashtroudi, and Hadi Hemmati. GloBug: Using global data in fault localization. *Journal of Systems and Software*, 177:110961, July 2021.
- [218] Mkhonto Mkhonto and Tranos Zuva. *Technology Acceptance: A Critical Review of Technology Adoption Theories and Models*, page 414–428. Springer Nature Switzerland, 2024.
- [219] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.
- [220] Amir H. Moin and Mohammad Khansari. Bug localization using revision log analysis and open bug repository text categorization. In *IFIP Advances in Information and Communication Technology*, pages 188–199. Springer Berlin Heidelberg, 2010.
- [221] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, March 2014.
- [222] Laura Moreno, Wathsala Bandara, Sonia Haiduc, and Andrian Marcus. On the relationship between the vocabulary of bug reports and source code. In *2013 IEEE International Conference on Software Maintenance*. IEEE, September 2013.
- [223] Laura Moreno, Gabriele Bavota, Sonia Haiduc, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, and Andrian Marcus. Query-based configuration of text retrieval solutions for software engineering tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, August 2015.
- [224] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. On the use of stack traces to improve text retrieval-based bug localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 151–160, 2014.
- [225] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. On the use of stack traces to improve text retrieval-based bug localization. In *2014 IEEE*

- International Conference on Software Maintenance and Evolution*, pages 151–160. IEEE, 2014.
- [226] Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. Convolutional neural network over tree structures for programming language processing. Unpublished, 2014.
- [227] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.
- [228] Christof Müller and Iryna Gurevych. Using wikipedia and wiktionary in domain-specific information retrieval. In *Lecture Notes in Computer Science*, pages 219–226. Springer Berlin Heidelberg, 2009.
- [229] Vijayaraghavan Murali, Lee Gross, Rebecca Qian, and Satish Chandra. Industry-scale IR-based bug localization: A perspective from facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, may 2021.
- [230] Vijayaraghavan Murali, Lee Gross, Rebecca Qian, and Satish Chandra. Industry-scale ir-based bug localization: A perspective from facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, May 2021.
- [231] Vijayaraghavan Murali, Lee Gross, Rebecca Qian, and Satish Chandra. Industry-scale IR-based bug localization: A perspective from facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, May 2021.
- [232] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005*. IEEE, 2005.
- [233] Koichi Nagatsuka, Clifford Broni-Bediako, and Masayasu Atsumi. Pre-training a BERT with curriculum learning by increasing block-size of input text. In *Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP 2021)*, pages 989–996, Held Online, September 2021. INCOMA Ltd.
- [234] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, May 2013.

- [235] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, Johannes Heidecke, Pranav Shyam, Boris Power, Tyna Eloundou Nekoul, Girish Sastry, Gretchen Krueger, David Schnurr, Felipe Petroski Such, Kenny Hsu, Madeleine Thompson, Tabarak Khan, Toki Sherbakov, Joanne Jang, Peter Welinder, and Lilian Weng. Text and code embeddings by contrastive pre-training, 2022.
- [236] Thanh Thi Nguyen and Vijay Janapa Reddi. Deep reinforcement learning for cyber security. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–17, 2021.
- [237] Thanh Van Nguyen, Anh Tuan Nguyen, Hung Dang Phan, Trong Duc Nguyen, and Tien N. Nguyen. Combining word2vec with revised vector space model for better code retrieval. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, May 2017.
- [238] Chao Ni, Wei Wang, Kaiwen Yang, Xin Xia, Kui Liu, and David Lo. The best of both worlds: integrating semantic features with expert features for defect prediction and localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, November 2022.
- [239] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguang Huang, and Bin Luo. Spt-code: Sequence-to-sequence pre-training for learning source code representations. *CoRR*, abs/2201.01549, 2022.
- [240] Feifei Niu, Christoph Mayr-Dorn, Wesley K. G. Assunção, LiGuo Huang, Jidong Ge, Bin Luo, and Alexander Egyed. The ablots approach for bug localization: is it replicable and generalizable? In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, May 2023.
- [241] Santiago Ontañón, Joshua Ainslie, Vaclav Cvicek, and Zachary Fisher. Making transformers solve compositional tasks. *CoRR*, abs/2108.04378, 2021.
- [242] OpenAI. <https://openai.com/blog/openai-codex>. [Accessed 11-07-2024].
- [243] Mark Palatucci, Dean Pomerleau, Geoffrey E. Hinton, and Tom M. Mitchell. Zero-shot learning with semantic output codes. In *Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems 2009. Proceedings of a meeting held 7-10 December 2009, Vancouver, British Columbia, Canada*, pages 1410–1418. Curran Associates, Inc., 2009.

- [244] Liang Pang, Yanyan Lan, Jiafeng Guo, Jun Xu, Jingfang Xu, and Xueqi Cheng. DeepRank. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, November 2017.
- [245] Mike Papadakis and Yves Le Traon. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628, September 2013.
- [246] Raghavendra Pappagari, Piotr Zelasko, Jesus Villalba, Yishay Carmiel, and Najim Dehak. Hierarchical transformers for long document classification. In *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. IEEE, December 2019.
- [247] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11*. ACM, July 2011.
- [248] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2014.
- [249] 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, November 2018.
- [250] Charith Peris, Christophe Dupuy, Jimit Majmudar, Rahil Parikh, Sami Smaili, Richard Zemel, and Rahul Gupta. Privacy in the time of language models. In *Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining, WSDM '23*. ACM, February 2023.
- [251] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *CoRR*, abs/1802.05365, 2018.
- [252] Matthew E. Peters, Sebastian Ruder, and Noah A. Smith. To tune or not to tune? adapting pretrained representations to diverse tasks. In *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*, pages 7–14, Florence, Italy, August 2019. Association for Computational Linguistics.

- [253] Sravya Polisetty, Andriy Miransky, and Ayşe Başar. On usefulness of the deep-learning-based bug localization models to practitioners. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, New York, NY, USA, September 2019. ACM.
- [254] Sravya Polisetty, Andriy Miransky, and Ayşe Başar. On usefulness of the deep-learning-based bug localization models to practitioners. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 16–25, 2019.
- [255] Michael Pradel, Vijayaraghavan Murali, Rebecca Qian, Mateusz Machalica, Erik Meijer, and Satish Chandra. Scaffle: bug localization on millions of files. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, jul 2020.
- [256] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.
- [257] Binhang Qi, Hailong Sun, Wei Yuan, Hongyu Zhang, and Xiangxin Meng. Dreamloc: A deep relevance matching-based framework for bug localization. *IEEE Transactions on Reliability*, 71(1):235–249, March 2022.
- [258] Zile Qiao, Wei Ye, Dingyao Yu, Tong Mo, Weiping Li, and Shikun Zhang. Improving knowledge graph completion with generative hard negative mining. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023*, pages 5866–5878, Toronto, Canada, July 2023. Association for Computational Linguistics.
- [259] Alex John Quijano, Sam Nguyen, and Juanita Ordonez. Grid search hyperparameter benchmarking of bert, albert, and longformer on duorc. *CoRR*, abs/2101.06326, 2021.
- [260] Ella Rabinovich, Matan Vetzler, Samuel Ackerman, and Ateret Anaby Tavor. Reliable and interpretable drift detection in streams of short texts. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*. Association for Computational Linguistics, 2023.
- [261] Alec Radford, Ilya Sutskever, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. In *arxiv*, 2018.
- [262] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, May 2013.

- [263] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. BugCache for inspections. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*. ACM Press, 2011.
- [264] Mohammad Masudur Rahman, Foutse Khomh, Shamima Yeasmin, and Chanchal K. Roy. The forgotten role of search queries in ir-based bug localization: an empirical study. *Empirical Software Engineering*, 26(6), August 2021.
- [265] Mohammad Masudur Rahman, Foutse Khomh, Shamima Yeasmin, and Chanchal K. Roy. The forgotten role of search queries in IR-based bug localization: an empirical study. *Empirical Software Engineering*, 26(6), August 2021.
- [266] Mohammad Masudur Rahman and Chanchal K. Roy. Improving IR-based bug localization with context-aware query reformulation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, October 2018.
- [267] Shivani Rao and Avinash Kak. Retrieval from software libraries for bug localization. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, May 2011.
- [268] Shivani Rao and Avinash Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, ICSE11. ACM, May 2011.
- [269] Shivani Rao and Avinash Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 43–52, 2011.
- [270] Ayushi Rastogi and Georgios Gousios. How does software change?, 2021.
- [271] Nir Ratner, Yoav Levine, Yonatan Belinkov, Ori Ram, Inbal Magar, Omri Abend, Ehud Karpas, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. Parallel context windows for large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6383–6402, Toronto, Canada, July 2023. Association for Computational Linguistics.
- [272] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. A large-scale study of programming languages and code quality in github. *Communications of the ACM*, 60(10):91–100, September 2017.

- [273] Abdul Razzaq, Jim Buckley, James Vincent Patten, Muslim Chochlov, and Ashish Rajendra Sai. BoostNSift: A query boosting and code sifting technique for method level bug localization. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, September 2021.
- [274] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. Quickly generating diverse valid test inputs with reinforcement learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, June 2020.
- [275] M. Renieres and S.P. Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings*. IEEE Comput. Soc.
- [276] Manos Renieres and Steven P Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 30–39. IEEE, 2003.
- [277] Hamed Rezapour, Sadegh Jamali, and Alireza Bahmanyar. Review on artificial intelligence-based fault location methods in power distribution networks. *Energies*, 16(12):4636, June 2023.
- [278] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. ”why should I trust you?”: Explaining the predictions of any classifier. *CoRR*, abs/1602.04938, 2016.
- [279] Stephen Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford. Okapi at trec-3. In *Overview of the Third Text REtrieval Conference (TREC-3)*, pages 109–126. Gaithersburg, MD: NIST, January 1995.
- [280] Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. Okapi at trec-3. In *TREC*, volume 500-225 of *NIST Special Publication*, pages 109–126. National Institute of Standards and Technology (NIST), 1994.
- [281] Sebastian Ruder and Barbara Plank. Learning to select data for transfer learning with bayesian optimization. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.
- [282] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM, June 2018.

- [283] 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)*. IEEE, November 2013.
- [284] Abhishek Sainani, Preethu Rose Anish, Vivek Joshi, and Smita Ghaisas. Extracting and classifying requirements from software engineering contracts. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*. IEEE, August 2020.
- [285] Tommaso Dal Sasso, Andrea Mocci, and Michele Lanza. What makes a satisficing bug report? In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, August 2016.
- [286] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, May 2010.
- [287] Maria N.K. Schwarz and August Flammer. Text structure and title—effects on comprehension and recall. *Journal of Verbal Learning and Verbal Behavior*, 20(1):61–66, February 1981.
- [288] Agnia Sergeyuk, Yaroslav Golubev, Timofey Bryksin, and Iftekhar Ahmed. Using ai-based coding assistants in practice: State of affairs, perceptions, and ways forward, 2024.
- [289] Meera Sharma, Madhu Kumari, and V. B. Singh. Multi-attribute dependent bug severity and fix time prediction modeling. *International Journal of System Assurance Engineering and Management*, 10(5):1328–1352, September 2019.
- [290] Jeffrey Shen. Pretraining of transformers on question answering without external data, 2018. Accessed on 01.18.2021.
- [291] Zhendong Shi, Jacky Keung, and Qinbao Song. An empirical study of BM25 and BM25f based feature location techniques. In *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices*. ACM, November 2014.
- [292] B. Sisman and A. C. Kak. Incorporating version histories in information retrieval based bug localization. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, June 2012.

- [293] Bunyamin Sisman and Avinash C. Kak. Assisting code search with automatic query reformulation for bug localization. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 309–318, 2013.
- [294] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, July 2005.
- [295] Kihyuk Sohn. Improved deep metric learning with multi-class n-pair loss objective. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.
- [296] Mozhan Soltani, Felienne Hermans, and Thomas Bäck. The significance of bug report elements. *Empirical Software Engineering*, 25(6):5255–5294, September 2020.
- [297] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, July 2017.
- [298] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp, 2019.
- [299] Visual Studio. Visual Studio IntelliCode — Visual Studio - Visual Studio — visualstudio.microsoft.com. <https://visualstudio.microsoft.com/services/intellicode/>. [Accessed 11-07-2024].
- [300] 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*. ACM, aug 2021.
- [301] Chul Sung, Tejas Dhamecha, Swarnadeep Saha, Tengfei Ma, Vinay Reddy, and Rishi Arora. Pre-training bert on domain resources for short answer grading. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6073–6077, 2019.
- [302] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

- [303] Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211, August 1999.
- [304] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empir. Softw. Eng.*, 19(6):1665–1705, December 2014.
- [305] Zhiwen Tang and Grace Hui Yang. A reinforcement learning approach for dynamic search. In *Text REtrieval Conference (TREC)*. ACM, July 2017.
- [306] G. Tassej. The economic impacts of inadequate infrastructure for software testing, 2002.
- [307] TestSigma. AI-powered, low-code test automation platform — testsigma.com. <https://testsigma.com/ai-driven-test-automation>. [Accessed 11-07-2024].
- [308] Burak Turhan, Tim Menzies, Ayşe B Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.
- [309] Burak Turhan, Tim Menzies, Ayşe B. Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, January 2009.
- [310] Pradeep K. Tyagi. The effects of appeals, anonymity, and feedback on mail survey response patterns from salespeople. *Journal of the Academy of Marketing Science*, 17(3):235–241, June 1989.
- [311] William Uther, Dunja Mladenić, Massimiliano Ciaramita, Bettina Berendt, Aleksander Kołcz, Marko Grobelnik, Dunja Mladenić, Michael Witbrock, John Risch, Shawn Bohn, Steve Poteet, Anne Kao, Lesley Quach, Jason Wu, Eamonn Keogh, Risto Miikkulainen, Pierre Flener, Ute Schmid, Fei Zheng, Geoffrey I. Webb, and Siegfried Nijssen. TF-IDF. In *Encyclopedia of Machine Learning*, pages 986–987. Springer US, 2011.
- [312] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.
- [313] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. Fault localization using function call frequencies. *Journal of Systems and Software*, 193:111429, November 2022.

- [314] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.
- [315] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, September 2018.
- [316] Zhiyuan Wan, Xin Xia, Ahmed E. Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering*, 46(11):1241–1266, November 2020.
- [317] Bei Wang, Ling Xu, Meng Yan, Chao Liu, and Ling Liu. Multi-dimension convolutional neural network for bug localization. *IEEE Transactions on Services Computing*, pages 1–1, 2020.
- [318] Chaoyue Wang, Chang Xu, Xin Yao, and Dacheng Tao. Evolutionary generative adversarial networks. *IEEE Transactions on Evolutionary Computation*, 23(6):921–934, December 2019.
- [319] Di Wang, Matthias Galster, and Miguel Morales-Trujillo. A systematic mapping study of bug reproduction and localization. *Information and Software Technology*, 165:107338, January 2024.
- [320] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD/PODS '21*. ACM, June 2021.
- [321] Jiapeng Wang and Yihong Dong. Measurement of text similarity: A survey. *Information*, 11(9):421, August 2020.
- [322] Qianqian Wang, Chris Parnin, and Alessandro Orso. Evaluating the usefulness of IR-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, July 2015.

- [323] 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*. ACM Press, 2014.
- [324] 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*. ACM, June 2014.
- [325] Shuai Wang, Shengyao Zhuang, and Guido Zuccon. BERT-based dense retrievers require interpolation with BM25 for effective passage retrieval. In *Proceedings of the 2021 ACM SIGIR International Conference on Theory of Information Retrieval*. ACM, July 2021.
- [326] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, May 2016.
- [327] Yaojing Wang, Yuan Yao, Hanghang Tong, Xuan Huo, Ming Li, Feng Xu, and Jian Lu. Enhancing supervised bug localization with metadata and stack-trace. *Knowledge and Information Systems*, 62(6):2461–2484, February 2020.
- [328] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.
- [329] Anthony I. Wasserman. *Tool integration in software engineering environments*, page 137–149. Springer Berlin Heidelberg, 1990.
- [330] Zeng Wei, Jun Xu, Yanyan Lan, Jiafeng Guo, and Xueqi Cheng. Reinforcement learning to rank with markov decision process. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, August 2017.
- [331] Karl Weiss, Taghi M. Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big Data*, 3(1), May 2016.
- [332] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering*, 47(11):2348–2368, November 2021.

- [333] 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*. ACM, August 2016.
- [334] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, February 2019.
- [335] Tony White and Bernard Pagurek. Distributed fault location in networks using learning mobile agents. In *Approaches to Intelligence Agents*, pages 182–196. Springer Berlin Heidelberg, 1999.
- [336] Ratnadira Widyasari, Gede Artha Azriadi Prana, Stefanus Agus Haryono, Shaowei Wang, and David Lo. Real world projects, real faults: evaluating spectrum based fault localization techniques on python projects. *Empirical Software Engineering*, 27(6), August 2022.
- [337] Wikipedia. Kullback–Leibler divergence - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence, Jun 2022. [Accessed 09-06-2022].
- [338] E. Winter, D. Bowes, S. Counsell, T. Hall, S. Haraldsson, V. Nowack, and J. Woodward. How do developers really feel about bug fixing? directions for automatic program repair. *IEEE Transactions on Software Engineering*, 49(04):1823–1841, apr 2023.
- [339] 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*. IEEE, September 2014.
- [340] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. CrashLocator: locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. ACM Press, 2014.
- [341] 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)*. IEEE, July 2019.

- [342] Xin Xia, David Lo, Xingen Wang, Chenyi Zhang, and Xinyu Wang. Cross-language bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension, ICSE '14*. ACM, June 2014.
- [343] Shundan Xiao, Jim Witschey, and Emerson Murphy-Hill. Social influences on secure development tool adoption: why security tools spread. In *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing, CSCW'14*. ACM, February 2014.
- [344] 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)*. IEEE, December 2018.
- [345] 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.
- [346] 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, July 2018.
- [347] 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, January 2019.
- [348] 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)*. IEEE, December 2017.
- [349] 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)*, pages 338–347. IEEE, 2017.
- [350] Yan Xiao, Jacky Keung, Qing Mi, and Kwabena E. Bennin. Bug localization with semantic and structural features using convolutional neural network and cascade forest. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*. ACM, June 2018.
- [351] Huan Xie, Yan Lei, Meng Yan, Yue Yu, Xin Xia, and Xiaoguang Mao. A universal data augmentation approach for fault localization. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, May 2022.

- [352] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology*, 22(4):1–40, October 2013.
- [353] Jun Xu, Long Xia, Yanyan Lan, Jiafeng Guo, and Xueqi Cheng. Directly optimize diversity evaluation measures. *ACM Transactions on Intelligent Systems and Technology*, 8(3):1–26, January 2017.
- [354] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, oct 2017.
- [355] Hong Xuan, Abby Stylianou, Xiaotong Liu, and Robert Pless. *Hard Negative Examples are Hard, but Useful*, page 126–142. Springer International Publishing, 2020.
- [356] Aidan Z. H. Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. Large language models for test-free fault localization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24*. ACM, February 2024.
- [357] Geunseok Yang, Kyeongsic Min, and Byungjeong Lee. *Applying Deep Learning Algorithm to Automatic Bug Localization and Repair*, page 1634–1641. Association for Computing Machinery, New York, NY, USA, 2020.
- [358] Haoran Yang, Yu Nong, Tao Zhang, Xiapu Luo, and Haipeng Cai. Learning to detect and localize multilingual bugs. 2024.
- [359] Zhou Yang, Jieke Shi, Shaowei Wang, and David Lo. Incbl: Incremental bug localization. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, November 2021.
- [360] Xingcheng Yao, Yanan Zheng, Xiaocong Yang, and Zhilin Yang. NLP from scratch without large-scale pretraining: A simple and efficient framework. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 25438–25451. PMLR, 17–23 Jul 2022.
- [361] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing*, 4(2):100211, June 2024.

- [362] Xin Ye. The dataset of six open source Java projects. 8 2014.
- [363] 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*. ACM, November 2014.
- [364] 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*. ACM, may 2016.
- [365] Klaus Changsun Youm, June Ahn, Jeongho Kim, and Eunseok Lee. Bug localization based on code change histories and bug reports. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, December 2015.
- [366] Chao Yu, Jiming Liu, Shamim Nemati, and Guosheng Yin. Reinforcement learning in healthcare: A survey. *ACM Computing Surveys*, 55(1):1–36, November 2021.
- [367] 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)*. IEEE, December 2020.
- [368] Yisong Yue, Thomas Finley, Filip Radlinski, and Thorsten Joachims. A support vector method for optimizing average precision. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, July 2007.
- [369] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences. *CoRR*, abs/2007.14062, 2020.
- [370] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.
- [371] Dejiao Zhang*, Wasi Ahmad*, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. Codesage: Code representation learning at scale. In *The Twelfth International Conference on Learning Representations*, 2024.
- [372] Hanwang Zhang, Yang Yang, Huanbo Luan, Shuicheng Yang, and Tat-Seng Chua. Start from scratch. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, November 2014.

- [373] Hongyu Zhang. An investigation of the relationships between lines of code and defects. In *2009 IEEE International Conference on Software Maintenance*. IEEE, September 2009.
- [374] Jinglei Zhang, Rui Xie, Wei Ye, Yuhan Zhang, and Shikun Zhang. Exploiting code knowledge graph for bug localization via bi-directional attention. In *Proceedings of the 28th International Conference on Program Comprehension*. ACM, July 2020.
- [375] Jinglei Zhang, Rui Xie, Wei Ye, Yuhan Zhang, and Shikun Zhang. Exploiting code knowledge graph for bug localization via bi-directional attention. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 219–229, 2020.
- [376] 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, June 2019.
- [377] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*. ACM, May 2006.
- [378] Xunhui Zhang, Yue Yu, Georgios Gousios, and Ayushi Rastogi. Pull request decisions explained: An empirical overview. *IEEE Transactions on Software Engineering*, 49(2):849–871, 2023.
- [379] Yang Zhang, Chenwei Zhang, and Xiaozhong Liu. Dynamic scholarly collaborator recommendation via competitive multi-agent reinforcement learning. In *Proceedings of the Eleventh ACM Conference on Recommender Systems*. ACM, August 2017.
- [380] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, Xin Xia, and David Lo. Context-aware neural fault localization. *IEEE Transactions on Software Engineering*, 49(7):3939–3954, July 2023.
- [381] Feng Zhao, Xinning Li, Yating Gao, Ying Li, Zhiquan Feng, and Caiming Zhang. Multi-layer features ablation of BERT model and its application in stock trend prediction. *Expert Systems with Applications*, 207:117958, November 2022.
- [382] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, May 2015.

- [383] 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)*. IEEE, June 2012.
- [384] 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)*, pages 14–24. IEEE, 2012.
- [385] Jianghong Zhou and Eugene Agichtein. RLIRank: Learning to rank with reinforcement learning for dynamic search. In *Proceedings of The Web Conference 2020*. ACM, April 2020.
- [386] Runjie Zhu, Xinhui Tu, and Jimmy Xiangji Huang. Deep learning on information retrieval and its applications. In *Deep Learning for Data Analytics*, pages 125–153. Elsevier, 2020.
- [387] Wei Zhu, Xiaoling Wang, Yuan Ni, and Guotong Xie. AutoTrans: Automating transformer design via reinforced architecture search. In *Natural Language Processing and Chinese Computing*, pages 169–182. Springer International Publishing, 2021.
- [388] Ziye Zhu, Yun Li, Hanghang Tong, and Yu Wang. CooBa: Cross-project bug localization via adversarial transfer learning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, July 2020.
- [389] Ziye Zhu, Yun Li, Yu Wang, Yaojing Wang, and Hanghang Tong. A deep multimodal model for bug localization. *Data Mining and Knowledge Discovery*, April 2021.
- [390] Ziye Zhu, Hanghang Tong, Yu Wang, and Yun Li. BL-GAN: Semi-supervised bug localization via generative adversarial network. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–14, 2022.
- [391] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, page 531–540, New York, NY, USA, 2008. Association for Computing Machinery.
- [392] 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*, ESEC/FSE09. ACM, August 2009.
- [393] 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*, pages 91–100, 2009.
- [394] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, September 2010.
- [395] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE’07: ICSE Workshops 2007)*. IEEE, May 2007.
- [396] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 47(2):332–347, February 2021.
- [397] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

APPENDICES

Appendix A

Themes emerged from qualitative study

A.1 Themes identified from developers interview

After conducting the interview, we transcribed the responses and employed an open coding methodology to extract codes and identify overarching themes. Through this analytical process, we derived the following key themes:

- The current debugging process is a tool-assisted manual effort.
- Developers expect better accuracy from a bug localization (BL) tool compared to human developers.
- Concerns exist among developers about over-reliance on specific tools.
- Developers worry about low-quality solutions provided by the tool.
- Suggestions for similar bugs are seen as a productivity booster.
- Suggesting the best person to fix an issue may not be helpful due to team dynamics and shifts.
- Developers expect the tool to integrate seamlessly into an automated pipeline.
- Some believe the tool might not add value if developers already understand the problem.

- Past experiences with bug localization tools have not been favorable.
- Developers recognize that bug localization speed improves with experience.
- The tool may not perform well in poorly maintained codebases.
- There is a preference for tools that do not transmit data to external sources.
- Developers expect the tool to operate faster than manual debugging by humans.
- Commit-time bug localization is considered more critical than issue-based localization.
- The tool should focus on automating failing test generation rather than just bug localization.
- Line-level functionality is a preferred feature for the tool.
- The effectiveness of top-k suggestions is seen as dependent on code quality.
- Developers value transparency in the tool's workings.
- The tool is expected to be more beneficial for identifying hard-to-localize bugs.
- Junior developers are likely to benefit the most from the tool's guidance.
- Software developers overall would gain significant advantages from using the tool.
- The tool could serve as a training aid for onboarding new developers.
- Developers express concerns about losing essential soft skills due to reliance on such tools.