UNDERSTANDING THE CHALLENGES OF BUG REPORT MANAGEMENT: A COMPREHENSIVE STUDY WITH DUPLICATE BUG REPORT DETECTION AND BUG LOCALIZATION

by

Sigma Jahan

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science

at

Dalhousie University
Halifax, Nova Scotia
July 2023

I dedicate this thesis with all my heart to my beloved father, whose unwavering support and endless encouragement have made this journey possible.

Contents

List of Tables

List of Figures

List of Abbreviations Used

AST Abstract Syntax Tree

AUC Area Under Curve

BM25 Best Match 25

CNN Convolutional Neural Network

DNN Deep Neural Network

DL Deep Learning

DLSW Deep Learning Software Systems

EB Expected Behavior

FP False Positive

FN False Negative

GloVe Global Vectors

GPU Graphics Processing Unit

IDE Integrated Development Environment

IR Information Retrieval

IRBL Information Retrieval-based Bug Localization

LDA Latent Dirichlet Allocation

LSTM Long Short Term Memory

ML Machine Learning

MAP Mean Average Precision

MRR Mean Reciprocal Rank

NLP Natural Language Processing

NN Neural Network

NDLSW Non-Deep Learning Software Systems

NDL Non-Deep Learning

OB Observed Behavior

OS Operating System

RNN Recurrent Neural Network

ROC Receiver Operating Characterstics

rVSM Revised Vector Space Model

S2R Steps to Reproduce

TP True Positive

TN True Negative

t-SNE t-Distributed Stochastic Neighbor Embedding

VCH Version Control History

VSM Vector Space Model

VMP Vocabulary Mismatch Problem

Abstract

Software bugs cost the global economy trillions of dollars annually and claim ∼50% of the developers' time. In a recent survey with the major tech giants (e.g., Google, Meta, Microsoft), software practitioners identify several challenging parts of bug report management, such as duplicate bug report detection and bug localization. Over the last 20 years, Information Retrieval (IR)-based techniques have been frequently used to automate these tasks due to their computational efficiency and lightweight nature. However, to date, IR-based solutions have not been mainstream in bug report management despite their potential, which warrants an in-depth investigation. We thus conduct a large-scale empirical study to better understand the challenges and potential of IR-based techniques in two bug report management tasks, namely in duplicate bug report detection and bug localization. First, traditional techniques for detecting duplicate bug reports primarily target textually similar duplicates, often overlooking textually dissimilar duplicates commonly found in bug tracking systems. We thus collect 92,854 bug reports, construct two datasets containing textually similar and textually dissimilar duplicate bug reports, and determine the performance of three existing techniques. By answering three research questions, our findings underscore the limitations of existing techniques in detecting textually dissimilar duplicate bug reports and suggest that these reports often miss crucial information (e.g., steps to reproduce). Second, pinpointing the location of a bug within the software code is another challenging task where IR-based techniques have been used. However, only a little attempt has been made to detect the bugs in deep learning systems. Thus, we collect 2,365 bugs from deep-learning applications and 2,913 bugs from traditional systems and determine the performance of three existing IR-based techniques in localizing software bugs. We found that IR-based methods show poor performance in localizing bugs from deep-learning applications. We also found that deep learning bugs are connected to artifacts other than source code (e.g., GPU, training data, external dependencies), which might explain the poor performance of IR-based techniques for these bugs.

Acknowledgements

First and foremost, I am thankful to the Almighty for granting me the physical and mental capacities that empower me to pursue my endeavors with purpose and vigor.

I would like to thank my father, Md. Aminul Hoque. His constant encouragement, unconditional love, and confidence in my abilities have been instrumental in shaping my aspirations and goals, including embarking on this transformative journey of Ph.D.

Then, I would like to thank my supervisor, Dr. Masud Rahman, for giving me the life-changing opportunity to pursue a Ph.D. under his guidance. My gratitude also goes to all the members of our research lab (RAISE lab) for sharing their journey with me and making my graduate life more enjoyable.

I am grateful to Dalhousie University and the Faculty of Computer Science for their kind and generous financial support through scholarships, awards, and bursaries, which have enabled me to focus on my research work. My gratitude also goes to all the individuals that I have encountered at Dalhousie University, as they have been there for me through thick and thin, providing unwavering support which goes beyond academia. In particular, I want to express my sincere appreciation to Dr. Michael McAllister, Megan Baker, and Robert Hawkey for their constant encouragement and guidance.

Lastly, I would like to express my sincere and heartfelt gratitude to all my friends and family. Each and every one of them has been an invaluable presence throughout my journey, continuously supporting me, encouraging my personal growth, and enriching my life in countless ways.

Chapter 1

Introduction

1.1 Motivation

Software bugs are human-made errors in the code that prevent it from working correctly [1]. The number of bugs in a large software system could range from hundreds to thousands [2]. Due to software bugs and failures, the global economy loses billions of dollars every year [3]. Developers also spend ∼50% of their programming time dealing with software bugs and failures [4].

During software maintenance, developers perform several tasks involving software bugs that are commonly known as *bug report management* [3]. In a recent survey with tech giants (e.g., Google, Meta, Microsoft, Amazon, and Twitter), software practitioners identify several challenging parts of bug report management. In particular, ∼80% of 327 participants suggest the difficulties and importance of two tasks – duplicate bug report detection and bug localization. Given the significant interest of practitioners, an in-depth investigation into these tasks is warranted. In this work, we thus investigate two tasks from bug report management, namely duplicate bug report detection and bug localization.

First, software bugs are submitted to a bug-tracking system as *bug reports*. Hundreds of bug reports are submitted every day in large software systems (e.g., Mozilla, Eclipse, Firefox) [3]. Duplicate bug reports occur when multiple persons submit multiple bug reports for the same bug. Due to the asynchronous nature of the bug report submission, traditional bug tracking systems (e.g., Bugzilla) can not prevent duplicate bug reports [5]. As a result, on average, 35.8%–41.6% of bug reports remain duplicates in the bug tracking systems, which pose a major overhead during software maintenance [3, 6]. For example, the submission of duplicate bug reports often leads to bug non-reproducibility [7] and delayed bug resolution [8]. The process of finding the duplicates of a given bug report is called *duplicate bug report detection* (or bug de-duplication) [9]. Second, once duplicate bugs are resolved, and non-duplicate

ones are triaged, the developers first need to identify the location of a bug within a software system, which is known as *bug localization* [10]. According to a recent survey [3], bug localization has been reported to be the most challenging part of bug report management. Given the importance and challenges of these tasks from bug report management, the research community proposed many tools and techniques to support them.

To detect duplicate bug reports, researchers employ various methodologies, including Natural Language Processing (NLP) [11, 12, 13], Information Retrieval (IR) [14, 15, 16, 17], and Machine Learning (ML) [18, 19, 20, 21, 22]. However, they are far from perfect due to the complexity and ambiguity of natural language texts in a bug report. NLP based techniques might be limited in detecting duplicate reports when there is a textual mismatch between the reports [23]. IR-based approaches suffer from the *vocabulary mismatch problem* [24, 25], a typical phenomenon that stems from two textual documents describing the same concept using different vocabularies. On the other hand, ML-based approaches suffer from data imbalance problems and the lack of generalizability and explainability [18, 23, 26].

Similarly, to localize software bugs, researchers employ various methodologies, including IR, data mining, and ML [27]. In particular, IR methods have received significant attention due to their low computational cost and minimal external dependencies [28]. However, to date, most of these methods focus on detecting bugs in traditional software systems (e.g., Eclipse IDE). Unlike bugs in non-deep learning software systems (hereby traditional), deep learning-related bugs could be hidden in the source code, training data, trained models, or even deployment scripts [29, 30, 31]. Besides, as reported earlier [32], the use of popular Deep Learning (DL) libraries (e.g., PyTorch, Caffe, and TensorFlow) could lead to complex bugs (e.g., incorrect model parameter bugs). Due to simplistic assumptions (e.g., the textual similarity between bug reports and code), IR-based techniques might not be sufficient for detecting complex bugs such as deep learning bugs.

Given the importance of the above two tasks in bug report management and the limitations of existing tools or techniques, an in-depth investigation is warranted to better understand their challenges and opportunities.

1.2 Problem Statement

To automatically detect duplicate bug reports and to localize the bugs, researchers employ various methodologies, including Natural Language Processing (NLP) [11, 12, 13], Information Retrieval (IR) [14, 15, 16, 17, 28], Machine Learning (ML), and Deep Learning (DL) [18, 19, 20, 21, 22, 33, 34]. However, despite their strengths, these techniques have not been widely adopted in practice to date. In this research, we focus on two different aspects concerning software bugs and attempt to understand the strengths and weaknesses of the existing tools and techniques supporting bug report management.

Existing techniques might not be able to handle the textual dissimilarity in duplicate bug reports. Duplicate bug reports can be divided into two different categories: bug reports describing the same issue with similar texts (e.g., Table 2.1) and bug reports describing two similar issues using different texts (e.g., Table 2.2) [11]. The second category refers to duplicate bug reports that share the same underlying root cause but are written in different styles. There could be instances where multiple bugs have different Observed Behavior (OB) and Steps to Reproduce (S2R), but they share the same underlying cause (e.g., Table 2.2). We call them *textually dissimilar* duplicate bug reports. According to our investigation, 19% – 23% of the duplicate bug reports could be textually dissimilar.

Most of the existing NLP and IR-based techniques focus on detecting duplicate bug reports that use similar texts [23, 35, 36]. Unlike NLP and IR-based techniques, ML-based techniques can capture the non-linear, complex relationships between two items [37, 38, 39], and thus have the potential to tackle the challenge of textual dissimilarity in duplicate bug report detection. However, they also suffer from poor outlier handling, class imbalance problem, and a lack of explainability [40]. Thus, automated detection of duplicate bug reports still remains a highly challenging problem that warrants further investigation [41].

Existing techniques might not be able to handle multifaceted dependencies of software bugs. Deep learning bugs have multifaceted dependencies such as external frameworks (e.g., third-party libraries), external environments (e.g., OS, GPU), and non-code artifacts (e.g., training data). Existing bug localization

techniques, including IR-based ones, might not be equipped well to tackle these multifaceted dependencies. For example, deep learning-related bugs could be hidden in the source code, training data, trained models, or even deployment scripts [29, 30, 31]. Retrieval-based approaches might not be effective for these bugs since they can find bugs in the source code only. As suggested by a recent work [32], the use of popular deep learning libraries (e.g., PyTorch, Caffe, and TensorFlow) could also lead to complex bugs in deep learning applications. Besides, software bugs can be triggered by external entities (e.g., third-party libraries), which are called *extrinsic bugs* [42]. According to our investigation, deep learning software systems contain more extrinsic bugs than traditional software systems.

Over the years, many approaches have been designed to localize bugs in traditional software systems using IR [43, 44, 45, 46], dynamic program analysis [47, 48], and DL [33, 34, 49]. With the increasing prevalence of deep learning systems, recently Wardat et al. [34] propose to localize bugs in deep neural networks combining dynamic and statistical analysis. However, their sole focus on model and training bugs, low accuracy, and a strong reliance on the Keras library pose major challenges towards industry-wide adoption. Kim et al. [50] use basic IR-based techniques, such as rVSM and BM25, to localize bugs in deep learning applications and report poor performance without any comprehensive analysis or explanation. In other words, to date, the potential of existing solutions (e.g., IR-based approaches) for localizing bugs in deep learning applications has neither been rigorously investigated nor well understood.

In short, given the above gaps in the literature – textual dissimilarity issue in duplicate bug reports and multifaceted dependency issue of software bugs — a comprehensive investigation of duplicate bug report detection and bug localization is highly warranted. Furthermore, the knowledge gained from the investigation can help us to develop more effective strategies for effective bug report management, leading to cost-effective software maintenance.

1.3 Contribution

We conduct two separate but complementary empirical studies to better understand the challenges and opportunities of two important tasks from bug report management: *duplicate bug report detection* and *bug localization*. In this section, we divide our research contribution into two parts as follows.

In the first study, we attempt to better understand the impacts of textual dissimilarity on the detection of duplicate bug reports. First, we collect a total of 92,854 bug reports from three large-scale software systems (Eclipse, Firefox, and Mobile). We divided our dataset into textually similar and textually dissimilar duplicate bug reports using N-grams and textual similarity analysis. First, we extract the N-grams from both bug reports of duplicate reports. Then we used cosine similarity on the N-gram representations of texts to measure the textual similarity between duplicate pairs and apply quartile-based analysis to identify the two types of duplicate bug reports. We found that the existing techniques (BM25 [14], LDA+GloVe [51], and Siamese CNN [52]) perform significantly poorly in detecting textually dissimilar duplicate bug reports.

To better understand the differences between textually similar and textually dissimilar duplicate bug reports, we conduct three different analyses: descriptive analysis, embedding analysis, and manual analysis. We found that textually dissimilar duplicate reports indeed have a low textual similarity between each pair. Our embedding analysis using t-SNE [53] shows that textually dissimilar duplicate bug reports tend to position themselves at lower coordinates within the embedding space (Fig. 3.5), indicating a lower pairwise similarity between them in the embedding space. Finally, our manual analysis suggests that textually dissimilar duplicate bug reports often *miss important components* (e.g., steps to reproduce) or they have components (e.g., observed behaviors) that are written differently, which could lead to their overall textual differences.

Given the challenges of textual dissimilarity between duplicate bug reports above, we attempt to overcome them using domain-specific embedding [54]. Embedding models can capture semantic relations between two documents despite their textual differences [55, 56, 57]. We retrain a popular DL model – Siamese CNN [52] – for detecting duplicate bug reports where embeddings were learned from 92K bug reports

with the Skip-gram algorithm. We found that domain-specific embedding shows *mixed results* by improving the detection of textually dissimilar duplicate bug reports but worsening the detection of textually similar duplicate bug reports.

Our research findings above offer valuable insights into the challenges and complexities of detecting duplicate bug reports, especially when considering the quality and content of bug reports. They underscore the importance of providing detailed and comprehensive bug reports. Understanding that missing components like steps to reproduce and observed behaviors can hinder duplicate bug report detection will inform the stakeholders of the pain points, which could lead to improved bug reporting.

In the second study, we attempt to better understand the challenges of localizing bugs in deep-learning software systems. Given the extensive research in localizing bugs with IR methods and their inherent explainability, we use IR-based techniques in our experiments. First, we used two benchmark datasets – BugGL [58] for bugs in traditional software systems (a.k.a non-deep learning software systems) and Denchmark [59] for bugs in deep learning software systems. We found that the performance of existing IR-based techniques (BugLocator [43], BLUiR [44], and BLIA [60]) is significantly lower in localizing bugs from deep learning software systems than that of non-deep learning software systems.

To better understand the nature of bugs in deep-learning software systems, we conduct a manual analysis where we analyze the prevalence of different bug types in deep-learning software systems. We found that 64.80% of the bugs from deep learning software systems are related to deep learning algorithms (e.g., tensor bug), whereas 35.20% of the bugs are not related to deep learning (e.g., memory leak bug). We found that the bugs connected to deep-learning algorithms (e.g., training data bugs, GPU bugs) are more difficult to localize than other bugs in deep-learning software systems. Our analysis also highlights the specific strengths and weaknesses of existing IR-based techniques depending on the bug type, which could be useful for improving bug localization in deep-learning software systems.

To further understand the poor performance of IR-based techniques, we also analyze the implication of extrinsic bugs (triggered by external entities) in deep learning software systems [42]. We found that deep learning software systems have four times more extrinsic bugs than traditional software systems. We also found that extrinsic

bugs, which are related to Graphics Processing Unit (GPU), are the most challenging ones to locate using IR-based techniques. Our analysis also indicates a significant correlation between bugs in deep learning software systems and extrinsic factors, which could be valuable insight for designing effective bug localization solutions for deep learning software systems.

Bug reports play a vital role in bug localization, serving as queries in IR-based localization [61]. In order to gain a deeper understanding of their role in detecting bugs from deep learning software systems, we conducted a manual analysis focusing on the quality of these reports. Our analysis showed that bug reports from deep learning applications contain a higher percentage of code snippets (83.11%) compared to that of traditional software (33.24%). However, they do not help much in bug localization, as code snippets alone might not be sufficient. Complex bugs (e.g., gradient instability during training) warrant a deeper understanding of the model architecture and training processes, which the code snippets may not always capture.

Our research on bug localization has important implications for various stakeholders, including software developers. Understanding that bugs can have an 'extrinsic' nature—arising from external factors like data, hardware, and environments—can help developers extend their scope of investigation. For bugs with external dependencies, they can focus on understanding and addressing the external factors contributing to the issue. Our findings suggest that IR-based techniques may not be suitable for bugs involving multifaceted dependencies or extrinsic factors (e.g., data bugs, GPU bugs). On the other hand, several deep learning bugs, such as model and tensor bugs, could be detected by carefully adapting IR-based methods. Being aware of the different bug categories, developers can also make more informed decisions during bug localization, which might ultimately improve the overall debugging process.

Our research offers strong empirical evidence and actionable insights regarding the textual dissimilarity issue of duplicate bug reports and multifaceted dependencies of deep learning software bugs, advancing bug report management, which makes our contribution *novel*.

1.4 Related Publications

The first part of this RAD report has been accepted and published at a reputable international conference. We provide the list of publications here. In each of these papers, I am the primary author, and all the studies are conducted by me under the supervision of Dr. Masud Rahman.

- Sigma Jahan and M. Masudur Rahman. *Towards Understanding the Impacts of Textual Dissimilarity on Duplicate Bug Report Detection.* In Proceeding of The 30th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2023), pp. 25, Macao, China, March 2023.

Apart from the aforementioned paper, one more paper is currently ready to submit to a journal, as listed below.

- Sigma Jahan, Mehil Shah, and M. Masudur Rahman. *Towards Automated Bug Localization in Deep Learning Software Systems.* Journal of Empirical Software Engineering (EMSE 2023) - to be submitted.

1.5 Outline of the report

The report contains five chapters in total. We conduct two separate but complementary studies to better understand the challenges of bug report management. This section outlines the chapters as follows:

- Chapter 1 discusses the motivation, problem statements, and research contributions and provides an overview of the report structure.
- Chapter 2 provides a comprehensive background, covering topics such as textually dissimilar bugs, extrinsic bugs, and the techniques which will be required to follow the rest of the report.
- Chapter 3 discusses our first study on duplicate bug report detection, *Towards Understanding the Impacts of Textual Dissimilarity on Duplicate Bug Report Detection.*

- Chapter 4 discusses our second study on bug localization, *Towards Automated Bug Localization in Deep Learning Software Systems.*
- Chapter 5 concludes the report with a list of directions for future works.

Chapter 2

Background

In this chapter, we introduce the necessary terminologies and concepts to follow the remainder of the report. We introduce duplicate bug reports, textually similar and textually dissimilar duplicate bug reports, Best Match 25 (BM25), Latent Dirichlet Allocation (LDA), Global Vectors (GloVe), Convolutional Neural Network (CNN), Siamese CNN, extrinsic and intrinsic bugs, the taxonomy of deep-learning bugs, Information Retrieval (IR), and Vector Space Model (VSM).

2.1 Software bug

A software bug is a human-made error in a computer program that causes it to behave unexpectedly or produce incorrect results [5]. Once a bug is reported, it goes through various stages, including duplicate bug detection, triaging, bug localization, fixing, quality assurance, and finally, closure. Many of these tasks are supported by tools and techniques and are commonly known as bug report management [3].

2.2 Bug report

A bug report is a formal document that is prepared and submitted by a software tester, user, or any individual who encounters a software bug [43]. Ideally, a bug report includes at least three types of information – Observed Behavior (OB) (e.g., symptoms of a bug), Steps to Reproduce (S2R), and Expected Behavior (EB) [5]. Bug reports serve as a means of communication between the submitter of a bug and the development team, providing crucial information to facilitate the bug resolution.

Table 2.1: Example of textually similar duplicate bug reports from Firefox

Textually Similar Duplicate Bug Reports
Bug ID: 1337204
Title
Mozilla Firefox 51.0.1 uses a lot of memory with Kaspersky, and seven other addons enabled
Description
EB: Mozilla Firefox must use less RAM Memory, and I believe there is a bug in memory, and resources management of this version, and it needs debugging and releases a new version. S2R: I tried to clean up my RAM memory with some system tools, but this issue returned, and I am obligated to close and open my browser again and again without result. OB: Mozilla Firefox version 51.0.1 uses too much RAM memory and sometimes lags and freezes. I am using many tabs to work with my Mozilla Firefox version. 51.0.1 for the Windows 64-Bits version. After a few hours, firefoxes overload my RAM memory for unexpected reasons. I need to have many tabs to work, but I have 8GB RAM, and Mozilla eats at least 3-4GB of my RAM.
Bug ID: 1346556
Title
High memory usage from orphan nodes on forum websites with addon
Description
EB: Mozilla must take care of this issue and fix this bug, such the users are obligated to downgrade to previous versions. This solution is temporary, such as the security levels of Firefox are reducing and making the users remain under security risks. S2R: Hello, I have updated my Mozilla Firefox to version 50.1.0 direct to version 52 due to version 51 appearing with high RAM Memory usage. OB: After this update, Firefox appeared again with a high RAM memory leakage of about 5-6GB of RAM after a few hours. It seems the previous bug from version 51 is not fixed, but it transferred to version 52. This incident made to downgrade my version 52 back to version 50.1.0. My OS is Windows ten Pro x64 Bits.
Keyword Overlaps: 20+

EB=Expected Behaviour, **S2R**=Steps to Reproduce, **OB**=Observed Behaviour

2.3 Duplicate bug report

Duplicated bug reports occur when multiple persons submit multiple bug reports for the same bug. Due to the asynchronous nature of bug report submission, traditional bug tracking systems (e.g., Bugzilla) can not prevent duplicate bug reports. Thus, on

Table 2.2: Example of textually dissimilar duplicate bug reports from Mobile system

Textually Dissimilar Duplicate Bug Reports
Bug ID: 1618582
Title
Dynamic toolbar should not hide when the page is not scrollable.
Description
EB: I think this just boils down to Fenix should not hide the toolbar when a page is not scrollable S2R: Load the test case in Fenix Nightly and scroll down so that the dynamic toolbar disappears; observe that the grey overlay doesn't cover the bottom part of the Screen OB: Elements with height: 100% don't include the dynamic toolbar.
Bug ID: 1618579
Title
Text selection caret is misplaced with the dynamic toolbar.
Description
EB: Load a minimal example in Fenix Nightly, ensure the URL bar is not visible, select the text, and observe text selection carets are shifted up from where they usually should be. S2R: Long tap to select any word in the test case. Swipe up to make the dynamic URL bar disappear. OB: The text in the test case is in a 'position: fixed; bottom: 20px' element, so this looks similar to bug 1611032. It needs to update its position.
Keyword Overlaps: 3

EB=Expected Behaviour, **S2R**=Steps to Reproduce, **OB**=Observed Behaviour

average, 35.8%–41.6% of bug reports are duplicates in the bug tracking systems [3]. Duplicate bug reports can be divided into two different categories: textually similar duplicate bug reports (Table 2.1) and textually dissimilar duplicate bug reports (Table 2.2) [11].

2.3.1 Textually similar duplicate bug reports

Textually similar duplicate bug reports are such bug reports that describe the same issue with similar texts. From Table 2.1, we can see a pair of textually similar duplicate bug reports that have an overlap of 20 keywords. In both bug reports, the title discusses topics related to memory usage in Mozilla Firefox. The description texts also discuss issues with high memory consumption and suggest the presence of a bug in memory and resource management. Although the wording and specific

details may differ, the overall content and concerns expressed in both bug reports are textually similar, indicating a duplication of the reported problem.

2.3.2 Textually dissimilar duplicate bug reports

Textually dissimilar duplicate bug reports are such bug reports that address the same underlying root cause but express it using different writing styles. There could be instances where two bugs have different observable behaviors (OB) and steps to reproduce (S2R), but they share the same underlying cause. Table 2.2 shows a pair of bug reports that are considered duplicates by the developers, despite being textually dissimilar. Although these bug reports share only three keywords, they both address issues related to the dynamic toolbar in the Mobile system. The first bug report indicates that the dynamic toolbar in a mobile application disappears when the page cannot be scrolled. On the other hand, the second bug report suggests an issue in text selection (e.g., incorrect positioning) when the dynamic toolbar is present. Although the symptoms of these bugs are different, and the bug reports are textually dissimilar as a result, the root cause was the same – an error within the dynamic toolbar. Thus, developers have marked this pair as a duplicate.

2.4 Categories of Bug

2.4.1 Extrinsic bug

A bug caused by the factors external to a software system, such as changes to the operating environment, requirements, or third-party libraries, is known as *extrinsic bug*. Rodriguez-Perez et al. suggest three heuristics based on bug reports to identify extrinsic bugs as follows [42].

(a) Environment: An extrinsic bug is caused by a modification to the environment in which the software system operates. The environment could be an operating system, a physical machine, or even a cloud infrastructure.

(b) Requirement: An extrinsic bug is triggered by a change outside of the project's version control system. During software development, if a user requirement gets changed after implementation, the development team might implement the new requirement without discarding the old feature. The old, unexpected feature will then

be considered as an extrinsic bug.

(c) Third-party library: The bug found in the project's third-party library is considered an extrinsic bug. For example, if a software project uses a third-party library for processing images for a mobile application, and the app crashes when processing certain image formats due to a bug in that third-party library, that bug will be then considered an extrinsic bug.

2.4.2 Intrinsic bug

Any external factors do not cause an intrinsic bug, rather it is caused by a bug-introducing change (BIC) in the version control system [42]. For example, if a messaging application fails to deliver messages due to a logical error in a recent code change, that would be an intrinsic bug.

2.4.3 Taxonomy of bugs in deep learning software systems

Software bugs in deep learning applications can be divided into two categories – DL bug and NDL bug [62].

Deep Learning (DL) bug refers to a software error that is connected to the deep learning module embedded in the software system, causing inaccurate or unexpected output. According to the existing literature [62], DL bugs can be divided into five main categories: Model, Training, Tensor & Input, API, and GPU.

- **Model bug** is connected to the structure and properties of a deep learning model (Table 4.5). An example of the model bug is an incorrect model initialization caused by an input image size mismatch, resulting in inaccurate output in a computer vision application.

- **Training bug** occurs during the training phase of a deep learning application (Table 4.6). For instance, during the training of a deep learning model for object detection, if the loss function is incorrectly defined, the model will learn to detect objects with very poor accuracy, leading to incorrect output from the system.

- **Tensor & Input bug** occurs due to wrong tensor input or tensor calculation

Table 2.3: Example of deep learning-related and non deep-learning-related extrinsic bugs

DL+Extrinsic Bug **Bug ID: 1860 (Project: PyTorch+Fairseq)**
Title: wmt19 model cannot run on GPU except #0. **Description:** I am running the tutorial. I successfully loaded the model from the hub and tried to run it on the second GPU (id=1). , Which raised an exception that data and models are stored on different GPUs. With GPU (id=0) works fine. Code sample: import torch en2de = torch.hub.load('pytorch/fairseq', 'transformer.wmt19.en-de' checkpoint_file='model1.pt:model2.pt:model3.pt:model4.pt' tokenizer='moses',bpe='fastbpe').to(torch.device('cuda:1')) result = en2de.translate(['hello']) Environment: fairseq Version==0.9.0, PyTorch Version ==1.4.0, OS: Ubuntu 18.04,vPython version: 3.6, CUDA/cuDNN version: 10.2, GPU models and configuration: RTX 2080 x 2
NDL+Extrinsic Bug **Bug ID: 1426 (Project: PyTorch+Ignite)**
Title: GitHub CI on Windows is broken. **Description:** Normally, we should skip distributed tests on Windows with SKIP_DISTRIB_TESTS=1 CI_PYTHON_VERSION="3.7" sh tests/run_cpu_tests.sh, but a distributed test was executed: tests/ignite/contrib/engines/test_common.py ::test_distrib_cpu ERROR [2%] Related to beta support of distributed on Windows in Pytorch 1.7

issues (Table 4.7). For instance, if the tensor input shape is declared incorrectly, it will lead to output errors.

- **API bug** occurs due to incorrect use of an API in the deep learning software

system (Table 4.8). For example, an API bug might occur if a developer mistakenly calls the wrong API function from the deep learning framework, causing inaccurate results in the output.

- **GPU bug** is connected to the graphics processing unit used in the system (Table 4.9). For example, if the model's memory requirements exceed the available GPU memory, or the GPU is not compatible with the DL framework, then they could lead to errors during model training.

Non-Deep Learning (NDL) bug refers to a software error that is not related to the deep learning module but still leads to unexpected behaviors in deep learning applications. An example of NDL bugs could be a logical error in the code that leads to a deadlock, making the program being stuck in an infinite loop.

As shown in Table 2.3, Bug 1860 is a deep learning-related extrinsic bug triggered by the change in the environment. When the WMT19 model runs on multiple GPUs, the execution fails since the same GPU cannot store both the model and data. It is clearly related to the deep learning module. On the other hand, this bug is not related to the Fairseq library (a.k.a., deep learning application), rather it is related to external factors (e.g., GPU), which indicates its extrinsic nature.

In Table 2.3, Bug 1426 is another extrinsic bug connected to the Windows OS environment. The bug triggers when the tests from the CI pipeline are distributed over multiple Windows machines. It is clearly not related to deep learning (a.k.a., Non-DL bug), but the triggering factors are outside of the version control system, which indicates an extrinsic nature.

2.5 Information Retrieval (IR)

IR is a popular technique to capture relevant information from a vast collection of documents [63]. Once a query (i.e., information needed) is submitted, the IR executes the query to deliver relevant information from the document collection. In this report, several of these following IR concepts will be used frequently:

- Indexing: Indexing involves preprocessing all documents from a collection and constructing a data structure that links terms to documents.

- Search Space: Search space is the set of documents that can be searched and retrieved by executing a user's query.

- Query: A query is a set of keywords that specifies the information needed by a user.

- Ranking: Ranking involves sorting documents in descending order of relevance against a user query.

2.5.1 BM25

Best Match 25 (BM25) is a widely adopted ranking function in IR where the relevant documents are identified against a query using probabilistic retrieval models [64]. In particular, it assumes that certain documents are more relevant to the query based on the presence of query terms in each document [64]. BM25 incorporates various factors such as term frequency, document length, and inverse document frequency to compute the relevance score of a document [64]. BM25 score can be calculated as follows:

$$\text{BM25}(D,Q) = \sum_{i=1}^{n} \left(\frac{\text{IDF}(q_i) \cdot f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{\text{DL}(D)}{\text{avgDL}})} \right) \tag{2.1}$$

Here, D is a document in the collection, Q is the query, q_i is a query term, n is the total number of query terms, $f(q_i, D)$ is the frequency of query term in the document, $DL(D)$ is the document length, $avgDL$ is the average document length in the collection, $IDF(q_i)$ is the inverse document frequency of query term, k_1 and b are tuning parameters that control the impact of term frequency and document length normalization. In our experiment, we used the default parameter values (*k1=1.5*, *b=0.75*) for retrieving the duplicate bug reports according to the literature [14].

2.5.2 Vector Space Model

The vector space model is a popular concept in information retrieval. Using VSM, both documents and queries can be transformed into numerical vectors in a high-dimensional space. Then cosine similarity can be used to measure the similarity

between these vectors, allowing for text-based retrieval [65]. The cosine similarity score can be calculated as follows:

$$\text{Cosine Similarity} = \frac{A \cdot B}{\|A\| \cdot \|B\|}$$

Here, A and B represent the vectors being compared, . denotes the dot product of the vectors, $\|A\|$ and $\|B\|$ represent the magnitudes of the vectors. The Cosine similarity of the two documents ranges from 0 to 1. A cosine similarity of 1 indicates that the two vectors (a.k.a., two documents) being compared are similar, whereas 0 suggests that the vectors are completely dissimilar.

2.6 Topic Modeling

Topic modeling is a frequently used technique to automatically identify and extract latent topics from a collection of documents. It assigns a set of topics to each document and a set of words to each topic, capturing the underlying structure and patterns in the text data [66].

2.6.1 Latent Dirichlet Allocation (LDA)

LDA is a probabilistic model used for topic modeling, which assumes that each document is a mixture of topics, and each topic is a probability distribution over words [67]. LDA aims to estimate the topic-word and document-topic distributions by iteratively inferring the hidden structure that best explains the observed data [67]. In our first work on duplicate bug report detection, we apply LDA to a corpus containing master bug reports (a.k.a original bug reports), which generates document-topic probability distributions across all master reports (Fig. 2.1). We examine the topic distributions in each master bug report, detect the strongest topic, and then add the document to the corresponding cluster. We then employ machine learning techniques to identify duplicate bug reports by selecting the top N clusters with the most similar topic distributions.

To determine the optimal number of topics for our project, we utilized the Coherence score, as suggested [68]. We found that the model achieved the highest coherence score for ten topics. Fig. 2.1 illustrates ten chosen topics on the left and highlights

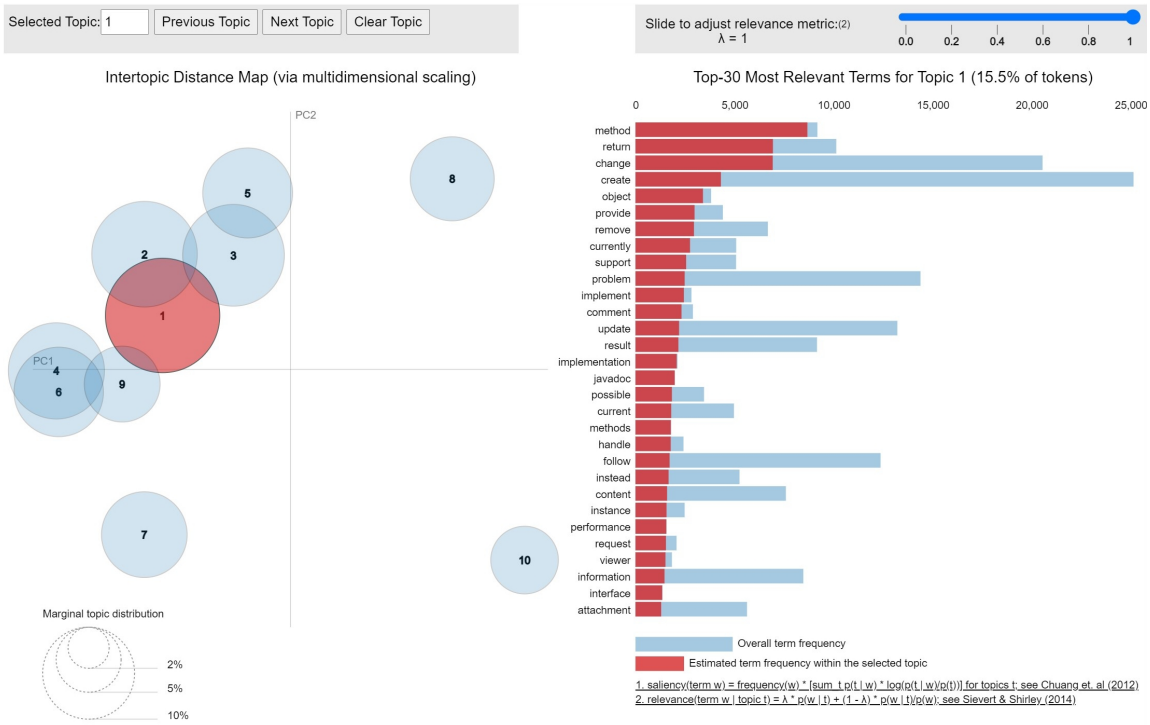


Figure 2.1: Example of topic modeling using LDA from our experiment on duplicate bug report detection

the top 30 most relevant terms for topic one, containing the highest number of tokens (15.5%) from the Eclipse corpus. The generative process of LDA can be represented by the following equation:

$$P(w, z, \theta, \phi|\alpha, \beta) = \prod_{d=1}^{D} P(\theta_d|\alpha) \prod_{k=1}^{K} P(\phi_k|\beta) \prod_{n=1}^{N} P(z_{d,n}|\theta_d) P(w_{d,n}|z_{d,n}, \phi_k) \qquad (2.2)$$

Here, $P(w, z, \theta, \phi|\alpha, \beta)$ is the joint probability of words (w), topic assignments (z), document-topic distributions θ, and topic-word distributions ϕ given the hyperparameters α and β. α determines the diversity of topics within documents. A higher value of α encourages documents to be composed of a mixture of more topics, resulting in broader coverage of topics within each document. β controls the diversity of words within topics. A higher value of β encourages topics to be composed of a wider range of words, allowing for more diversity in the vocabulary associated with each topic. D is the total number of documents in the corpus. K is the number of topics. N is the number of words in each document. d is the index variable representing a specific

document. k is the index variable representing a specific topic. n is the index variable representing a specific word within a document.

2.7 Embedding

In machine learning, embedding is a mechanism to represent high-dimensional data (e.g., images, text, or categorical variables) within a lower-dimensional space [69]. The process of learning an embedding involves mapping the original data to a vector space where the embedding values reliably capture different aspects or characteristics of the data.

2.7.1 Word Embedding

Word embedding refers to a numerical representation of words or phrases in a continuous vector space. By representing words into dense vectors in a continuous vector space, word embeddings are able to capture the semantic and syntactic links among words [70]. Word embeddings are typically learned through unsupervised ML techniques such as Word2Vec, GloVe, or fastText [70]. These pre-trained word embeddings come in a variety of dimensions, including 100, 200, and 300. Models can benefit from leveraging the pre-trained word embeddings since they already have relevant linguistic knowledge, which can be transferred to improve various NLP tasks [70].

2.7.2 GloVe: A Pre-trained Word Embedding

Pre-trained word embeddings are pre-computed vector representations of words learned from extensive text corpora. GloVe (Global Vectors for Word Representation) is a pre-trained word embedding technique developed by Stanford University that leverages word co-occurrence statistics to learn word embeddings [71]. It can capture semantic and syntactic information and serve as valuable resources for natural language processing tasks [70]. It utilizes matrix factorization techniques to capture both local and global relationships among words [71].

$$J = \sum_{i=1}^{V} \sum_{j=1}^{V} f(X_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij})^2 \tag{2.3}$$

Here, J represents the loss function to be minimized. V denotes the vocabulary size. $f(X_{ij})$ is a weighting function that assigns varying importance to co-occurrence pairs based on observed counts, capturing meaningful relationships between words. w_i and $\tilde{w}_j$ are the word vectors for words i and j respectively. b_i and $\tilde{b}_j$ are the corresponding bias terms. X_{ij} is the co-occurrence count or weight between words.

2.8 Neural Network

A neural network is a computational model in computer science that consists of interconnected nodes, known as artificial neurons, organized in layers [72]. It utilizes weighted connections, activation functions, and training algorithms to process input data. It propagates information through the network and adjusts the weights to optimize performance in various tasks such as pattern recognition, classification, or regression [72].

2.8.1 Convolutional Neural Network

CNN is specifically designed for processing grid-like input data (e.g., images, sequences) [73]. It incorporates convolutional layers that convolve across the input data and apply customizable filters or kernels to extract local features [73]. CNNs typically include pooling layers (e.g., max pooling) to downsample the features and reduce their spatial dimensions.

2.8.2 Siamese Convolutional Neural Network

Siamese CNN is a variant of the Siamese network that uses CNN layers. With two identical networks that learn shared representations for comparing pairs of inputs, Siamese CNN aims to bring similar inputs closer and dissimilar inputs farther apart [74]. It is commonly used for tasks involving similarity or distance-based comparisons, such as image similarity and text similarity. In our first work on duplicate bug report detection, we use Siamese CNN, where the network learned shared representations to compare pairs of bug reports and determine their similarity, helping in detecting duplicate reports.

In this chapter, we provided an overview of key terminologies and background concepts to help readers in understanding the subsequent sections of the report. We examined the concept of duplicate bug reports, including different types that can arise. Additionally, we explored diverse categories of bugs found in deep-learning software systems. Furthermore, we discussed essential techniques such as information retrieval, topic modeling, word embedding, and neural networks, which have been used in our empirical work.

Chapter 3

Duplicate Bug Report Detection

Existing works in the duplicate bug report detection [16, 52, 75, 76] primarily focus on textually similar duplicates without considering the intricacies of textually dissimilar duplicates. As a result, these techniques might not be effective in identifying textually dissimilar duplicate bug reports. However, to the best of our knowledge, there exist no studies that investigate the challenges in detecting textually dissimilar duplicate bug reports. In this chapter, we present our first study that investigates the impact of textual dissimilarity on duplicate bug report detection and offers meaningful insights.

The rest of this chapter is organized as follows. Section 3.1 provides an introduction to the research work. Section 3.2 presents our experimental design, datasets, and performance metrics. Section 3.3 discusses our experimental results and discusses our key findings. Section 3.4 discusses the threats to the validity of our work, and Section 3.5 discusses the related work.

3.1 Introduction

Software bugs are human-made errors in the code that prevent software from working correctly. During software maintenance, software bugs are submitted to a bug-tracking system as *bug reports* [4]. Hundreds of bugs are reported daily in large software systems (e.g., Mozilla, Eclipse). Duplicated bug reports occur when multiple people submit multiple bug reports for the same bug. Due to the asynchronous nature of bug report submission, traditional bug tracking systems (e.g., Bugzilla) can not prevent duplicate bug reports. Thus, on average, 35.8%–41.6% of bug reports are duplicates in the bug tracking systems [3]. These duplicate bug reports pose a major overhead during software maintenance since they often cost valuable development time and resources [6].

Manually examining hundreds of bug reports for duplicates is neither feasible nor practical. One of the major challenges in detecting duplicate bug reports is their

unstructured and ambiguous nature. Bug reports are written in natural language texts and thus may contain different words describing the same issue. The probability of two persons using the same text to explain the same issue is very low (e.g., 10%–15%) [25]. Given all these inherent challenges, automated detection of duplicate bug reports has become an active research topic since the last decade, which is also known as *bug deduplication* [9].

To automate the process of detecting duplicate bug reports, researchers employ various methodologies, including NLP [11, 12, 13], IR [14, 15, 16, 17], and ML [18, 19, 20, 21, 22]. However, they are far from perfect due to the complexity and ambiguity of natural language texts. NLP based techniques might be limited in detecting duplicate reports when there is a textual mismatch between the reports [23]. IR-based approaches suffer from the *vocabulary mismatch problem* [24, 25], a typical phenomenon that stems from two textual documents describing the same concept with different vocabularies. On the other hand, ML-based approaches suffer from data imbalance problems and a lack of generalizability and explainability [18, 23, 26].

Duplicate bug reports can be divided into two different categories: those that describe the same issue with similar texts and those that describe two similar issues using different texts (e.g., Fig. 2.2) [11]. The second category refers to duplicate bug reports that have the same underlying root cause but completely different writing styles. There could be instances where two bugs have different observable behaviors (OB) and steps to reproduce (S2R), but they share the same underlying cause. We call these types of duplicate bug reports *textually dissimilar* duplicate bug reports in this paper. According to our investigation, 19%–23% of the duplicate bug reports could be textually dissimilar.

Most of the existing NLP and IR-based techniques focus on detecting duplicate bug reports that use similar texts. Unlike NLP and IR-based techniques, ML-based techniques can capture the non-linear relationships between two items [37, 38, 39], and thus have the potential to tackle the challenge of textually dissimilar duplicate bug reports. However, they also suffer from poor outlier handling, class imbalance problem, and a lack of explainability [40]. Thus, automated detection of duplicate bug reports still remains a highly challenging problem that warrants further investigation [41].

In this paper, we conduct a large-scale empirical study to better understand the impacts of textual dissimilarity on the detection of duplicate bug reports. First, we collect a total of 92,854 bug reports from three large-scale software systems (Eclipse, Firefox, and Mobile) and *empirically* show how the existing techniques (BM25 [14], LDA+GloVe [51], and Siamese CNN [52]) perform poorly in detecting textually dissimilar duplicate bug reports. Second, we compare textually similar and textually dissimilar duplicate bug reports using a *combination* of quantitative and qualitative analyses. We found that the textually dissimilar duplicate bug reports differ from textually similar duplicate bug reports in terms of their underlying semantics and structures. For instance, textually dissimilar duplicate bug reports often have *missing components* (e.g., steps to reproduce) or components that are written differently (e.g., observed behaviors), which could lead to their overall textual differences. Finally, being inspired by the previous successes of domain-specific embedding [55, 57], we apply *domain-specific embedding* to counteract the impact of textual dissimilarity in duplicate bug report detection. We thus answer three important research questions in our study as follows.

(a) **RQ_1: Does the performance of existing techniques differ significantly in duplicate bug report detection between textually similar and textually dissimilar duplicate bug reports?**

We conducted experiments on our dataset using three existing techniques in duplicate bug report detection that employ IR, Topic Modeling, and ML, respectively. We found that the performance of existing techniques is lower (e.g., for recall rate@100, 10.02%–18.45% for BM25, 2.00%–6.49% for LDA+GloVe) in detecting *textually dissimilar* duplicate bug reports than that of *textually similar* duplicate bug reports. Our statistical tests (e.g., Wilcoxon-test [77], Cliff's delta) also report that their performance is significantly low for textually dissimilar duplicate bug reports. Although our findings reinforce a common belief about existing techniques, they also substantiate it with *solid empirical evidence* of the performance gap between the two categories of duplicate bug reports.

(b) **RQ_2: How do textually similar and textually dissimilar duplicate bug reports differ in their semantics and structures?**

To investigate the differences between textually similar and textually dissimilar duplicate bug reports, we use three different analyses: descriptive analysis, embedding analysis, and manual analysis. We found negative skewness in the similarity scores of textually dissimilar duplicate bug reports, which indicates a low textual similarity between each pair. We also visualize their embedding matrices using t-SNE [53], a non-linear dimensionality reduction technique. The visualization shows that textually dissimilar duplicate bug reports have a lower dimensional space than that of textually similar duplicate bug reports, which indicates a lower pairwise distance within the embedding space (Fig. 3.5). Finally, our manual analysis suggests that textually dissimilar duplicate bug reports often *miss important components* (e.g., steps to reproduce) or they have components (e.g., observed behaviors) that are written differently, which could lead to their overall textual differences.

(c) **RQ_3: Does domain-specific embedding help improve the detection of textually dissimilar duplicate bug reports?**

Our experiments in RQ_1 use a pre-trained, generic embedding model, GloVE [54], which might not have effectively overcome the challenges of textual dissimilarity in duplicate bug report detection [78, 79, 80]. We thus retrain our selected DL-based technique (e.g., Siamese CNN [52]) with a domain-specific embedding model [55, 56, 57] and repeat our experiments. In particular, we analyze 92,854 bug reports from Eclipse, Firefox, and Mobile systems to capture domain-specific embedding, which is then used to retrain the DL-based technique. We have also used oversampling to deal with imbalanced data problems during model training [81]. We found that domain-specific embedding shows *mixed results* by improving the detection of textually dissimilar duplicate bug reports but worsening the detection of textually similar duplicate bug reports.

3.2 Study Methodology

Fig. 3.1 shows the schematic diagram of our conducted study in this paper. We first collect bug reports from three different subject systems and construct two sets of

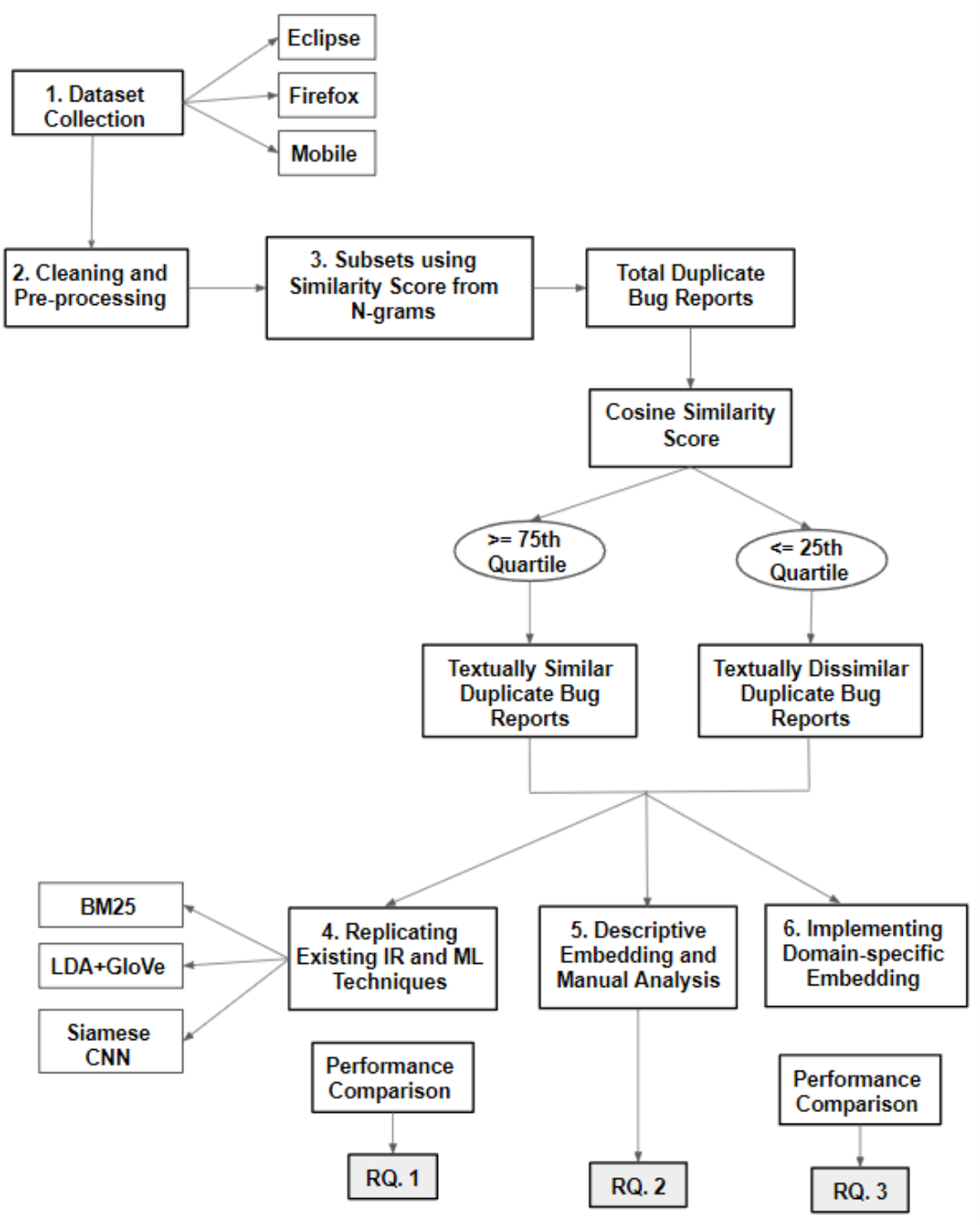


Figure 3.1: Schematic diagram of our conducted study on bug duplication

datasets for textually similar and textually dissimilar duplicate bug reports. We contrast the performance of three existing techniques [14, 51, 52] for duplicate bug report detection between textually similar and textually dissimilar duplicate bug reports.

Second, we perform three different analyses: manual analysis, descriptive analysis, and embedding analysis to understand the nature of textually dissimilar duplicate bug reports. Finally, we investigate the influence of domain-specific embedding with Siamese CNN on textually dissimilar duplicate bug reports. This section discusses the major steps of our study design as follows.

3.2.1 Construction of dataset

Dataset Collection. We collect duplicate bug reports from three large-scale, open-source software systems – Eclipse, Firefox, and Mobile – using a popular bug tracking system, namely *Bugzilla*. Existing studies [16, 18, 20, 23] have frequently used these systems, which makes them suitable for our research. Besides, these systems have stemmed from diverse application domains. Eclipse is a popular open-source Integrated Development Environment (IDE) written in Java. Firefox is a popular open-source web browser. While the above two systems are desktop-based applications, the remaining systems are mobile-based (e.g., Firefox for iOS, Focus for iOS, and GeckoView for Android). For the sake of brevity, we combine these three small systems and call them *Mobile* in the rest of the paper. Almost all the existing studies on duplicate bug report detection used the dataset dated before 2017 [23], which might not be an ideal representation of recent software bugs and issues [82]. In order to maintain a contemporary approach, we choose the bug reports from the last five years. We thus collected a total of 92,854 bug reports from three systems that were submitted within the last five years (01-01-2017 to 01-01-2022).

For many duplicate bug reports, the master reports were created before the last five years (Eclipse: 766, Firefox: 3514, Mobile: 214). In order to make a complete dataset with all the duplicate bug reports along with their master bug reports, we also retrieved them separately. Table 3.1 summarizes our study dataset. We see that about 6.60%, 20.53%, and 10.56% of the submitted bug reports were duplicates in Eclipse, Firefox, and Mobile systems, respectively.

Data cleaning and preprocessing. We capture four key fields from each bug

Table 3.1: Study dataset for duplicate bug report detection

Dataset (2017 – 2022)	**Eclipse**	**Firefox**	**Mobile**
Whole Dataset	49,244	38,290	5,320
Total Duplicate	3,248	7,859	562
Duplicate Ratio	6.60 %	20.53%	10.56%
Experimental Dataset (BM25, LDA + GloVe)			
Textually Similar Duplicate	679	1,414	122
Textually Dissimilar Duplicate	662	1,455	131
Experimental Dataset (Siamese CNN)			
Training Set	39,395	30,632	4,256
Testing Set	9,848	7,658	1,064
Textually Similar Duplicate	504	610	117
Textually Dissimilar Duplicate	497	734	89

report for our study: (a) bug id, which is unique for each bug report, (b) duplicate bug id, which points to the duplicate bug report, (c) title and description of the bug report, and (d) resolution, which indicates the duplicate status of a bug report.

We collect *title* and *description* from each bug report since they capture pertinent information for detecting duplicate bug reports. We clean and preprocess the *title* and *description* from each bug report using several steps as follows.

We apply standard natural language preprocessing to the title and description texts. First, we remove stopwords using a standard set of stopwords, which have little to no significance in capturing semantics. Then we perform token splitting along with the removal of punctuation marks, non-alphanumeric characters, numbers, HTML meta tags, and URLs. We also replace any non-alphanumeric characters with spaces and transform the text into lowercase [9]. Lastly, to transform each term into its base form, we use lemmatization using the NLTK library in Python. As performed by an earlier work [9], we discard any description with fewer than 50 characters since they do not contain enough information to be meaningful. On the other hand, bug reports might have long description text containing source code, lengthy stack traces, and log files, which could be noisy [9]. Hence, several previous studies [9, 52] selected bug reports containing at most 350 to 500 tokens. We experimented with 350, 500, and 1000 tokens. Using 500 tokens, the model delivers the best performance. Thus,

Table 3.2: Construction of textually similar and dissimilar duplicate pairs using n-gram based similarity scores

Unigram				**Bigram**			**Trigram**			**Duplicate BRs**	
Dataset	Median	Lower Quartile	Upper Quartile	Median	Lower Quartile	Upper Quartile	Median	Lower Quartile	Upper Quartile	Textually Similar	Textually Dissimilar
Eclipse	0.0502	0.0361	0.0675	0.0261	0.0189	0.0368	0.0212	0.0156*	0.0297*	679	662
Firefox	0.0633	0.0483	0.0776	0.0298	0.0233	0.0355	0.0232	0.0179*	0.0274*	1414	1455
Mobile	0.0679	0.0488	0.0944	0.0451	0.0316	0.0639	0.0403	0.0283*	0.0565*	122	131

we chose 500 as our token limit for the bug reports.

Construction of triplets. After we clean and preprocess the dataset, we construct triplets (b, b+, b-) where the existing techniques are supposed to detect b+. Here, b means the query bug report (query bug can be both duplicate and non-duplicate), b+ means duplicate bug report, and b- means non-duplicate bug reports. We created these triplets inspired by an existing work [52]. In the existing work, the duplicates (b,b+) were determined based on bug-tracking systems, whereas non-duplicates (b,b-) were randomly selected from the dataset. We also follow the same process in our ground truth construction. Then (b,b+) pairs were used as the ground truth for evaluating the existing techniques in duplicate bug report detection.

Dataset preprocessing for ML-based approach. To design an ML-based model (e.g., Siamese CNN) for duplicate bug report detection, pairwise bug reports containing texts and ground truth are required. Hence, we extract (b, b+) and (b, b-) pairs from the triplets above to generate our positive and negative samples, respectively. As was done by the original work [52], we split the whole dataset into an 80:20 ratio with random shuffling for training and testing. The test dataset was then further split into test and validation sets using a 50:50 ratio. We use the training set for model training and the validation set for assessing the model performance, fine-tuning, hyperparameter optimization, and mitigating overfitting [37]. On the other hand, the test set evaluates our model's performance on new and unseen data. Table 3.1 (bottom section) shows our ML models' training and testing datasets from all three systems.

Constructing subsets of study datasets based on textual similarity. We

divide our dataset into textually similar and textually dissimilar duplicate bug reports, which are essential for answering our research questions.

First, we store all duplicate bug reports as pairs in a separate dataset. Then, we collect N-grams (n = 1, 2, and 3) to compute the textual similarity [83] of each pair of duplicate bug reports. We used character-level n-grams to detect textually similar and textually dissimilar duplicate bug reports. This approach offers two primary advantages [13] as follows.

(a) Character-level n-grams have language independence, which enhances cross-language applicability.

(b) Character-level n-grams can capture sub-word features for noisy text analysis, such as bug reports.

Character-level n-grams are more robust to spelling variations, typos, and textual noise. They identify resemblances through shared character sequences, even in misspelled words. Furthermore, character-level n-grams can effectively capture distinctive writing styles and intricacies, making them particularly useful for our case [84]. We found that texts from bug reports are often noisy, and duplicate bug reports often occur due to distinct writing styles with different nuances. Thus, we used character-level n-grams to detect the textually similar and textually dissimilar duplicate bug reports.

We use the cosine similarity metric [11] to calculate the textual similarity between two bug reports. After getting the similarity score between the duplicate pairs, we analyze their descriptive statistics to determine their median similarity score, lower quartile (25th percentile) value, and upper quartile (75th percentile) value. For duplicate pairs that have a similarity score less than the lower quartile value, we denote them as *textually dissimilar* duplicate bug reports.

On the other hand, for duplicate pairs that have a similarity score more than the upper quartile value, we denote them as *textually similar* duplicate bug reports. We repeat all these steps using Unigram, Bigram, and Trigram to collect the common set of textually similar and textually dissimilar duplicate bug reports across three trials for our experiment. Table 3.2 shows similarity scores used for various N-grams to construct our textually similar and dissimilar duplicate bug reports. Finally, we got two subsets for textually similar and textually dissimilar duplicate bug reports,

respectively (Eclipse: 679 & 662, Firefox: 1414 & 1455, Mobile: 122 & 131).

3.2.2 Replication of existing techniques for experiments

To answer our first research question, we needed to replicate existing techniques on duplicate bug report detection. We thus select suitable representatives from the frequently used methodologies in duplicate bug report detection. In particular, we choose baseline methods from three frequently used methodologies – IR, Topic Modeling, and ML. We select BM25 [14] from IR, LDA+GloVe [51] from Topic-Modeling, and Siamese CNN [52] from ML for our experiment. Most of the recent models are based on these three primary approaches with incremental improvements [23, 36, 85]. We chose these baseline methods to determine the impact of textual dissimilarity on duplicate bug report detection without the effect of compounding factors (e.g., severity, priority, components, products, multimedia attachments).

BM25: IR relies on keyword overlaps between any two documents. We select a representative of IR, namely BM25, with default parameters (*k1=1.5*, *b=0.75*) for retrieving the duplicate bug reports. BM25 is a ranking function used in IR to score and rank documents based on their relevance to a user's query, considering factors like term frequency and document length [64]. It calculates a relevance score using a probabilistic model [64]. Yang et al. [14] first use BM25 for duplicate bug report detection.

LDA+GloVE: While BM25 is an established technique, it suffers from Vocabulary Mismatch Problem (VMP) [24, 25]. Several studies [35, 51, 86] adopt topic modeling in duplicate bug report detection to overcome this challenge. Latent Dirichlet Allocation (LDA) is a topic modeling approach that has the potential to overcome the vocabulary mismatch problem [87]. We call this approach LDA+GloVE in our research. As done by the original work [51], we use LDA for topic-based clustering (topic number=10), GloVe for the pre-trained word embedding (embedding dim = 100), and a unified text similarity measure (Cosine similarity and Euclidean similarity) for ranking the topmost, similar bug reports against a given bug report.

Siamese CNN: Unlike the above two techniques, ML-based approaches might be able to find non-linear relationships between dependent and independent variables [37, 38, 39]. The Siamese-CNN network leverages CNNs' ability to learn hierarchical

features and local structures from the regular texts [52]. This is crucial for identifying duplicate bug reports, as CNN can learn distinctive patterns in the character or word sequences that characterize the duplicates. On the other hand, the model also leverages the Siamese network's ability to learn shared representations to compare pairs of bug reports and determine their similarity, helping to detect the duplicate reports effectively. We thus implement the Siamese CNN for duplicate bug report detection by adapting earlier research [52]. We train our Siamese CNN model using K-fold cross-validation (e.g., *K=10*) and batch gradient descent, where original authors' batch sizes (e.g., 512 for Eclipse and Firefox, 256 for Mobile), learning rate (e.g., 0.001) and epoch number (e.g., 12) were chosen during the training phase to avoid model overfitting. Since the Mobile system has a small number of bug reports, a smaller batch size was chosen.

We used the authors' replication package for the LDA+GloVe model [51]. On the other hand, the replication packages of BM25 [14], and Siamese CNN [52] were not publicly available, and thus those techniques were carefully re-implemented by us based on the corresponding papers.

3.2.3 Performance evaluation

As we detected duplicate bug reports using IR, Topic Modeling, and ML techniques in our experiments, they were evaluated using appropriate performance metrics from these domains. In the case of BM25 and LDA+GloVe, we find all the duplicate bug reports for a given query bug report. Then we used Recall-rate@K [12, 23, 51, 88], one of the most popular performance metrics, to evaluate the IR and Topic Modeling approaches. On the other hand, for the ML-based model (Siamese CNN), we used traditional metrics such as F1 score, AUC, recall, and precision [23]. We used different evaluation metrics based on the original works [14, 51, 52]. It should be noted that our main goal was to contrast the performance of existing techniques between two sets of duplicate bug reports rather than to compare the techniques.

3.2.3.1 Recall-rate@K

Recall-rate@K determines the percentage of bug reports for each of which the duplicate bug report is found within the top K positions [19].

$$RecallRate@K = \frac{N_{detected}}{N_{total}} \tag{3.1}$$

$N_{detected}$ is the number of bug reports for which the duplicate reports have been correctly detected, and N_{total} is the number of total bug reports. We used nine different values of K (K = 1, 5, 10, 20, 25, 30, 50, 75, 100) to calculate the results of our IR-based and Topic Modeling techniques – BM25 and LDA+GloVe models.

3.2.3.2 Precision

Precision determines the percentage of bug reports for which duplicate bug reports are correctly detected. We calculate the precision of a technique as follows:

$$Precision(C) = \frac{TP}{TP + FP} \tag{3.2}$$

Here, True Positive (TP) is the number of items correctly classified as an instance of class C, and False Positive (FP) is the number of items wrongly identified as an instance of class C.

3.2.3.3 Recall

Recall determines the percentage of all duplicate bug reports that are correctly detected by a technique. We calculate the metrics as follows:

$$Recall(C) = \frac{TP}{TP + FN} \tag{3.3}$$

Here, True Positive (TP) is the number of items correctly classified as an instance of class C, and False Negative (FN) is the number of instances of class C that the model could not identify.

3.2.3.4 F1-measure

While both precision and recall focus on a specific aspect of a technique's effectiveness, F1-measure is a more comprehensive and effective method for evaluation. When aiming for high recall, where the model tries to identify all instances of a certain class, it may also classify some instances with low confidence, resulting in more mistakes and lower precision. To strike a balance between these conflicting requirements, the

F1 measure is utilized. We take the harmonic mean of precision and recall to compute the F1 measure as follows:

$$F1(C) = \frac{2 \cdot Precision(C) \cdot Recall(C)}{Precision(C) + Recall(C)} \tag{3.4}$$

here, $Precision(C)$ is precision and $Recall(C)$ is recall.

3.2.3.5 Area Under Curve

The Receiver Operating Characteristics (ROC) curve is a probability curve that separates between true positive and false positive rates [89]. Area Under the Curve (AUC) calculates the fraction of the area that falls under the ROC [89]. The AUC score ranges between 0 and 1, with one indicating that the model can perfectly classify observations into classes. The positive and negative samples are frequently imbalanced in actual data, such as ours. This imbalance significantly impacts precision and recall, whereas AUC is robust against the data imbalance.

After conducting the experiments, we evaluated our BM25 and LDA+GloVe models using Recall-rate@K, as used by existing work [14, 51]. On the other hand, we have evaluated the Siamese CNN model, as was done by the original work [52], using the remaining performance metrics.

3.3 Study Finding

3.3.1 Answering RQ_1: Does the performance of existing techniques differ significantly in duplicate bug report detection between textually similar and textually dissimilar duplicate bug reports?

We first evaluate the existing techniques against our whole dataset. Tables 3.3 and 3.10 summarize their performances.

From Table 3.3, we see that the BM25 approach, on average, performs higher with the Eclipse system than with Firefox and Mobile systems. On the other hand, the performance of the LDA+GloVe model is approximately 38.36% lower than that of BM25 for Recall-rate@100. LDA is limited in modeling topic correlations [90], which could be crucial to duplicate bug report detection. Table 3.10 (top section) shows that the performance of our ML based technique in detecting duplicate bug reports

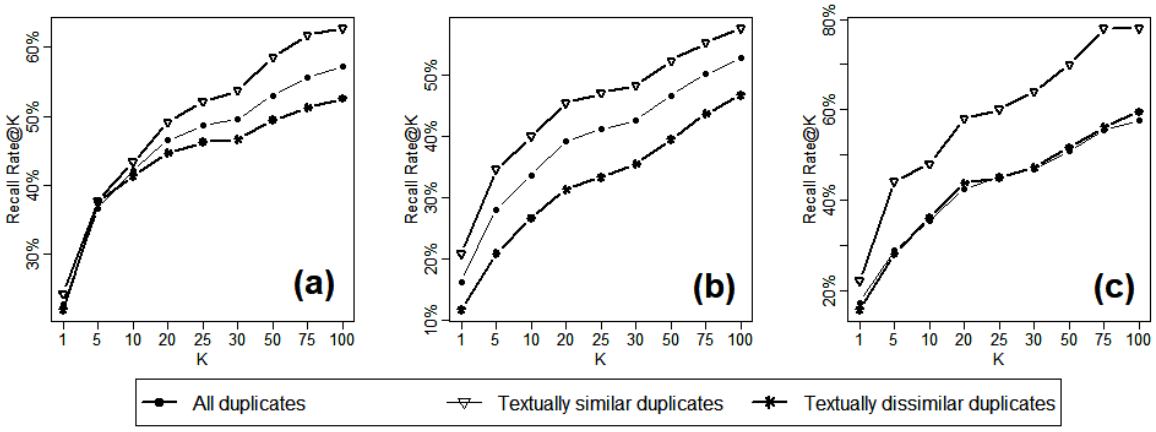


Figure 3.2: Performance of BM25 with all textually similar and dissimilar duplicate bug reports from (a) Eclipse, (b) Firefox, and (c) Mobile systems

Table 3.3: Experimental results of IR and LDA-based techniques (Recall–rate@k)% for duplicate bug report detection on whole dataset

Dataset	Method	k=1	k=5	k=10	k=100
Eclipse	BM25	22.64	36.60	42.03	57.31
	LDA+GloVe	0	5.5	10.5	16.0
Firefox	BM25	16.11	27.98	33.64	52.68
	LDA+GloVe	0	8.5	10.5	16.5
Mobile	BM25	17.25	28.95	35.38	57.60
	LDA+GloVe	0	2.5	7.0	20.0

ranges from 61.41% – 84.80% in terms of AUC. Precision, Recall, and F1-measure are slightly higher for Eclipse than for Firefox. Firefox has a better AUC score than Eclipse, as the AUC metric is robust to imbalanced data [81]. Precision, Recall, F1-measure, and AUC scores are slightly higher for the Mobile dataset than for the other two systems. Overall, the ML-based approach delivers the highest performance in duplicate bug report detection.

While the above analysis focuses on the whole dataset, we also determine the performance gap of existing techniques between *textually similar* and *textually dissimilar* duplicate bug reports. Table 3.4 shows that BM25 delivers a higher Recall-rate@K for textually similar duplicate bug reports than for dissimilar ones across all three systems - Eclipse, Firefox, and Mobile. For instance, with K=100, the difference between these two sets ranges from 10.20% to 18.45%. Fig. 3.2 further shows the performance difference between these two sets of bug reports for various K values.

Table 3.4: Experimental results of IR and LDA-based techniques (Recall–rate@k)% for duplicate bug report detection for textually similar and textually dissimilar duplicate bug reports

Dataset	Method	k=1	k=5	k=10	k=100
Textually Similar Duplicates					
Eclipse	BM25	24.15	37.63	43.26	62.78
	LDA+GloVe	0.00	10.00	13.50	20.50
Firefox	BM25	20.72	34.49	39.92	57.58
	LDA+GloVe	0.00	6.00	8.50	14.49
Mobile	BM25	22.00	44.00	48.00	78.00
	LDA+GloVe	0.00	4.00	7.50	20.50
Textually Dissimilar Duplicates					
Eclipse	BM25	21.83	37.50	41.27	52.58
	LDA+GloVe	0.00	6.50	11.50	18.50
Firefox	BM25	11.64	20.82	26.56	46.72
	LDA+GloVe	0.00	2.00	5.00	8.00
Mobile	BM25	15.73	28.09	35.96	59.55
	LDA+GloVe	0.00	2.50	4.50	15.00

We see that the difference is noticeably higher for Firefox and Mobile systems.

On the other hand, for the LDA+GloVe model, the performance differences between textually similar and textually dissimilar duplicate bug reports are 2.00% for Eclipse, 6.49% for Firefox, and 5.5% for the Mobile system with K=100 (Table 3.4). The performance gap is higher for Firefox and Mobile systems in the same manner as BM25. One possible reason behind this could be the higher duplicate ratios in Firefox and Mobile systems (see Table 3.1).

Table 3.10 shows the performance of our ML model - Siamese CNN - for both sets of duplicate bug reports across three systems. Here, the performance difference between textually similar and textually dissimilar duplicates is less apparent than that of the traditional methods above. ML models, especially deep learning models, can capture more contextual information beyond the syntax [91], which might explain the phenomenon. From Table 3.10, we see that the performance difference is smaller for Eclipse than for the other two datasets. For instance, the AUC differences between textually similar and textually dissimilar duplicate bug reports are 9.57% for Eclipse, 9.38% for Firefox, and 2.97% for the Mobile dataset. In the case of the F1-measure, the performance difference for the Eclipse dataset is 10.92%, whereas, for the Firefox

Table 3.5: Experimental results of Siamese CNN technique (%) for duplicate bug report detection

Metric	Test Dataset	Textually Similar	Textually Dissimilar
Eclipse			
AUC	61.41	56.31	46.74
Recall	93.00	55.00	45.00
Precision	92.00	63.00	51.00
F1	92.49	58.73	47.81
Firefox			
AUC	64.70	66.44	57.06
Recall	82.00	52.00	48.00
Precision	78.00	75.00	59.00
F1	79.95	61.41	52.93
Mobile			
AUC	84.80	63.96	60.99
Recall	94.00	72.00	58.00
Precision	93.00	80.00	76.00
F1	93.49	75.79	65.79

Table 3.6: Experimental results of Siamese CNN technique with Oversampling (%) for duplicate bug report detection

Metric	Textually Similar	Textually Dissimilar
Eclipse		
AUC	56.31	46.98
Recall	55.00	47.00
Precision	63.00	53.00
F1	58.73	49.82
Firefox		
AUC	66.51	57.35
Recall	52.00	49.00
Precision	75.00	60.00
F1	61.41	53.94
Mobile		
AUC	63.96	60.99
Recall	72.00	58.00
Precision	80.00	76.00
F1	75.79	65.79

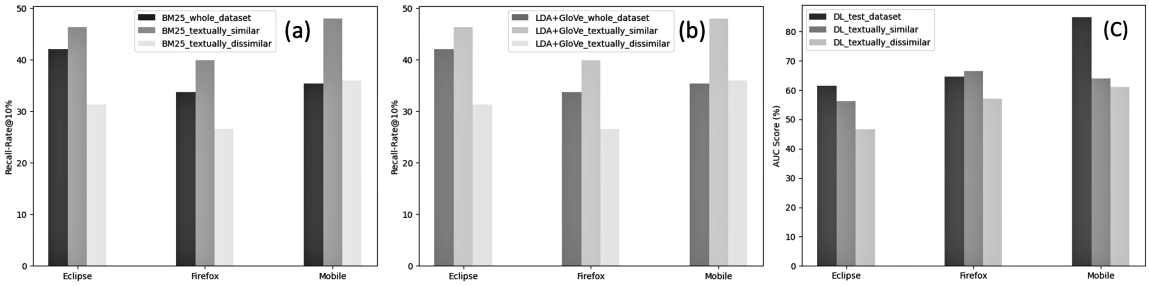


Figure 3.3: Performance of all the techniques for textually similar and dissimilar duplicate bug reports (a) BM25, (b) LDA+GloVe, (c) Siamese CNN

dataset, it is 8.48%. For Mobile data, the performance difference is 10.00%. We also note that the performance of our ML-based technique is lower for the two subsets of bug reports than for the whole dataset. As shown in Table 3.1, both subsets contain a smaller number of duplicate bugs than the whole dataset. That is, the two experiments use the same trained model but are tested with different numbers of test samples, which might explain the finding.

We also perform statistical tests to determine the significance of the performance gap between textually similar and dissimilar duplicate bug reports (Table 3.7). For each of the three systems, we first evaluate BM25 and LDA+GloVe using Recall-rate@K measures against textually similar and dissimilar duplicate bug reports where we consider various K values (K = 1, 5, 10, 20, 25, 30, 50, 75, 100). Then, we performed *Shapiro-Wilk normality test* [92] to determine the distribution of each set. We got two non-normal distribution pairs (LDA+GloVe for Eclipse and Firefox) out of six sets of pairs. Then we used appropriate parametric, non-parametric tests, and effect size tests to compare the two sets of Recall-rate@k values from textually similar and dissimilar duplicate bug reports. For the normal distribution, we used *paired t-test* as the parametric test [93], and for the non-parametric test, we used the *Wilcoxon Signed-Rank test* [77]. In both types of significance tests, the p-values were less than the threshold (0.05) except for two sets (BM25 in the Eclipse system & LDA+GloVe in the Mobile system). Thus, the null hypothesis can be rejected for all comparisons except for the case of BM25 in Eclipse and LDA+GloVE in the Mobile system. In other words, the performances of BM25 and LDA+GloVE techniques are significantly different between textually similar and dissimilar duplicate bug reports.

While the significance of a result indicates how probable it is that it is due to

Table 3.7: Statistical tests for the performance gap between textually similar and dissimilar duplicates

Dataset	**Method**	**PD**	**Significance (p-value)**	**Effect Size**
Eclipse	BM25	N	0.2865	Medium (0.52)
	LDA+GloVe	NN	0.0113*	Large (0.80)
Firefox	BM25	N	0.0329**	Large (1.10)
	LDA+GloVe	NN	0.0103*	Medium (0.31)
Mobile	BM25	N	0.0586**	Large (0.96)
	LDA+GloVe	N	0.1471	Medium (0.72)

PD=Probability Distribution, **NN**=Non-normal, **N**=Normal,
*=Significant, **=Strongly Significant

chance, the effect size indicates the extent of the difference [94]. Our experiments found different effect sizes ranging from medium to large (Table 3.7). We see that the effect size of BM25 is large for Firefox and Mobile systems and medium for Eclipse. On the other hand, the LDA+GloVe model has a medium to large effect size across the three systems. Thus, our results from effect size tests reinforce the above finding from significance tests. In other words, the existing techniques perform significantly poorly in detecting textually dissimilar duplicate bug reports. Even though our findings above mostly match natural intuition, we performed extensive experiments on three different systems using three different methodologies, which resulted in robust empirical evidence. Thus, we not only reinforce the existing belief about the existing techniques on duplicate bug detection but also substantiate it with solid empirical evidence.

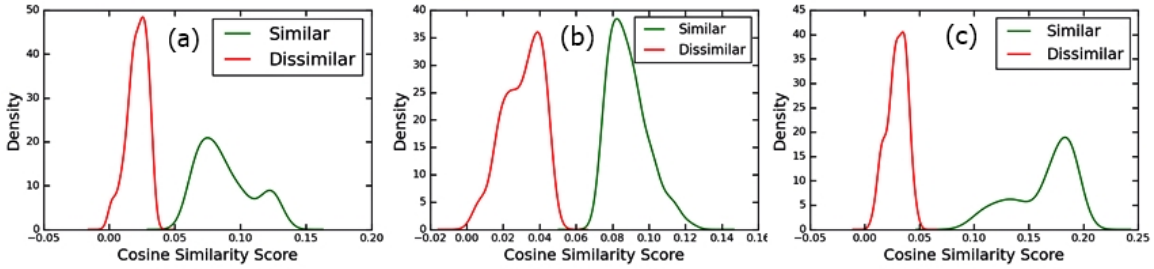


Figure 3.4: Distribution of similarity measures for textually similar and dissimilar duplicate bug reports from (a) Eclipse, (b) Firefox, and (c) Mobile system

Summary of RQ_1: The performances of existing techniques (e.g., BM25, LDA+GloVe) are *significantly* lower in detecting textually dissimilar duplicate bug reports than that of textually similar duplicate bug reports. Our finding also substantiates a common belief about existing techniques on duplicate bug report detection with *solid empirical evidence*.

3.3.2 Answering RQ_2: How do textually similar and textually dissimilar duplicate bug reports differ in their semantics and structures?

In this research question, we investigate how textually dissimilar duplicate bug reports might be different from textually similar duplicate bug reports. We answer this question using three different analyses – descriptive analysis, embedding analysis, and manual analysis – as follows:

Descriptive analysis. Descriptive analysis involves examining the data that helps describe, show, or summarize data points. It helps determine patterns or outliers that might emerge, which could lead to further statistical analyses. After cleaning and preprocessing the bug reports from each subject system, we calculate the *cosine similarity* score between each pair of duplicate bug reports using their TF-IDF measures from the textually similar and dissimilar subsets. Then we perform descriptive analysis on these similarity scores and capture five different statistics: Skewness, Kurtosis, Mean, Median, and Standard Deviation. Fig. 3.4 and Table 3.8 summarize our descriptive analysis for Eclipse, Firefox, and Mobile systems.

Skewness is a measure of symmetry [95]. Distribution is symmetric if it looks the same on the left and right of the center point [95]. From Fig. 3.4, we find the scores of textually similar duplicate bug reports to be positively skewed for Eclipse

Table 3.8: Descriptive analysis of similarity scores between bug reports

Dataset	Skew	Kurt	Mean	Median	Std
Eclipse					
Textually Similar	0.64	-0.83	0.09	0.08	0.02
Textually Dissimilar	-0.70	0.03	0.02	0.02	0.01
Firefox					
Textually Similar	0.89	0.37	0.09	0.09	0.01
Textually Dissimilar	-0.50	-0.67	0.03	0.03	0.01
Mobile					
Textually Similar	-0.88	-0.53	0.16	0.17	0.03
Textually Dissimilar	-0.38	-0.58	0.03	0.03	0.01

and Firefox. The positive skewness indicates that a significant number of duplicate pairs are highly similar [95]. On the other hand, for the textually dissimilar dataset, we found negative skewness for all three datasets, indicating that the cosine similarity measures are very low for most of the textually dissimilar duplicate pairs [95].

Kurtosis is a measure of whether the distribution is heavy-tailed or light-tailed relative to a normal distribution [95]. Distributions with positive kurtosis tend to have heavy tails or outliers, whereas distributions with negative kurtosis tend to have light tails [96]. A uniform distribution would be the extreme case. From Fig. 3.4, we note that textually dissimilar duplicate pairs have a negative kurtosis for Firefox and Mobile systems. The kurtosis for the Eclipse system is also close to zero. That is, the similarity scores from textually dissimilar duplicate pairs have a lighter tail than the normal distribution. In other words, their similarity scores are mostly centered around the mean value, which is also low.

On the other hand, a similar conclusion can be made for the textually similar duplicate pairs with the Eclipse and Mobile systems. However, it should be noted that the Firefox system contains several times more bug reports than the Eclipse and Mobile systems. The remaining two of three statistics - mean and median - are also several times lower for textually dissimilar duplicate bug reports than their counterparts.

Embedding analysis. Word embedding is a frequently used mechanism for detecting duplicate bug reports, representing words as semantically relevant dense

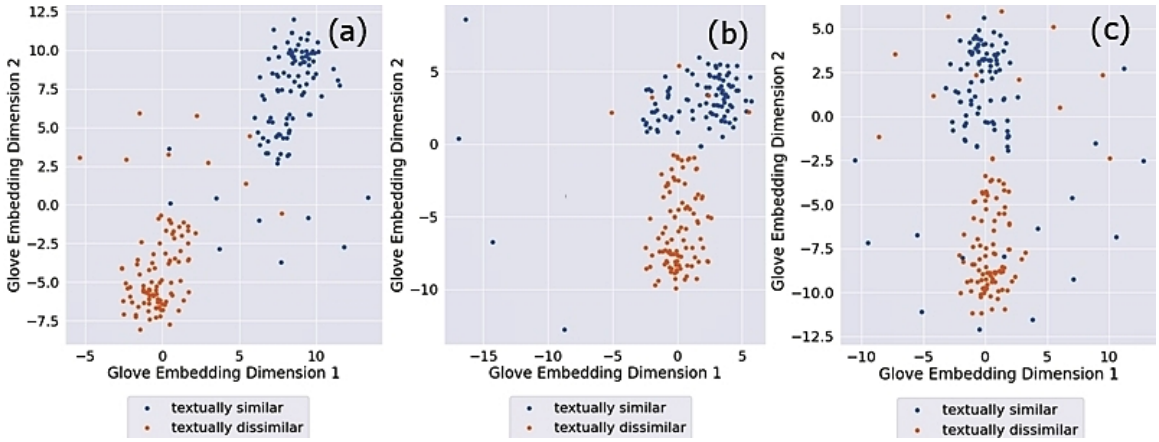


Figure 3.5: t-SNE visualization of GloVe embeddings for 100 random samples from both textually similar and dissimilar duplicate bug reports from (a) Eclipse (b) Firefox (c) Mobile system

real-valued vectors [97]. While our descriptive analysis above focuses on text-level similarity, we now perform embedding analysis to visualize the semantic differences between textually similar and textually dissimilar duplicate bug reports. We employ t-SNE to visualize high-dimensional data representing embedding vectors in lower dimensions [53]. It illustrates high-dimensional data (e.g., bug report embeddings) by projecting them into a two-dimensional space. The intra-cluster detail can be observed by measuring the pairwise distances in the higher and lower dimensions spaces [53].

First, we collect 100 random pairs of bug reports from both textually similar and textually dissimilar datasets. Then, we apply pre-trained word embeddings (GLoVe) to each pair of duplicate bug reports and capture their respective embeddings. We used padding to ensure uniform vector dimensions and concatenated the embeddings from each duplicate pair into a matrix. Then, we employ t-SNE, a dimensionality reduction technique for visualizing intricate, high-dimensional data like bug reports in lower dimensions [53]. t-SNE uncovers patterns, clusters, and relationships within complex data. We apply t-SNE to the embedding matrices of each duplicate pair, aiming to find the differences between textually similar and textually dissimilar duplicate bug reports within the embedding space. Using Gaussian distributions [53], t-SNE calculated pairwise similarities of duplicate bug reports based on their embedding matrices, with each data point (dots) representing such a similarity. Our

Table 3.9: Results of manual analysis for duplicate bug reports

Type	EB	OB	S2R	Overall
Prevalence Ratio (%)				
Textually Similar Duplicate Bug Reports	99.00	100.00	87.00	N/A
Textually Dissimilar Duplicate Bug Reports	87.05	100.00	55.39	N/A
Similarity Ratio (%)				
Textually Similar Duplicate Bug Reports	78.00	83.00	55.00	90.00
Textually Dissimilar Duplicate Bug Reports	56.84	38.20	31.65	21.58

EB=Expected behaviour, **OB**=Observed behaviour, **S2R**=Steps to reproduce

t-SNE visualization revealed distinct clusters occupying different coordinates in the lower-dimensional space across all three datasets. This observation signifies differences in the similarity patterns between textually similar and textually dissimilar duplicate bug reports. We use the optimal perplexity and iterations (e.g., perplexity = 40, iterations = 7000) and default similarity metric (i.e., cosine similarity) for our visualization.

From Fig. 3.5, we see the difference in embedding visualization of textually similar and dissimilar bug reports for all three datasets. In embedding space, textually dissimilar duplicate pairs (orange dots) are clustered in a different dimensional area than the textually similar pairs (blue dots). In all three systems, the embedding positions of textually dissimilar bug reports are in lower coordinates than that of textually similar duplicate bug reports. In particular, the changes are noticeable across the vertical dimension between textually similar and dissimilar duplicate bug reports for Firefox and Mobile. Interestingly, for Eclipse, we see that textually dissimilar duplicate bug reports are at different locations, even across the horizontal dimension. All these differences above suggest that the word semantics of duplicate pairs within the same dataset could be similar but very different from that of other datasets. In other words, the semantics of textually dissimilar duplicate bug reports are noticeably different from that of textually similar bug reports.

Manual Analysis. We chose 100 randomly selected duplicate bug report pairs (50 textually similar + 50 textually dissimilar) from each subject system. Then, we manually analyze 150 pairs of textually similar and 150 pairs of textually dissimilar duplicate bug reports. Ideally, each bug report should have three components – Expected Behaviour (EB), Observed Behaviour (OB), and Steps to Reproduce (S2R) [98]. These components have been used by previous research to reformulate queries during duplicate bug report detection using IR [76]. Similarly, we make use of these components from each duplicate pair to understand how textually similar duplicate bug reports and textually dissimilar duplicate bug reports might differ from each other.

First, we go through the *title* and *description* of each bug report and detect the presence of EB, OB, and S2R components in each bug report. Then we analyze the prevalence ratios of these components in both textually similar and textually dissimilar duplicate bug reports. We also calculate the textual similarity between the two bug reports from each pair for each of the three components separately. As a part of manual analysis, we also look for shared terms, keywords, technologies, and overall literary analogies between a duplicate bug report and a master bug report. The primary manual analysis for the paper was conducted by the first author and documented using an Excel sheet, with a total of ≈25 hours spent on the analysis.

Table 3.9 shows the prevalence ratios of all three components from both textually similar and textually dissimilar duplicate bug report pairs. We see that textually dissimilar duplicate bug reports have a higher percentage of missing components. For example, as shown in Table 3.9, 44.61% and 13% of textually dissimilar pairs do not contain any steps to reproduce (S2R) and expected behaviors (EB) in their bug reports. In particular, either both bug reports in those pairs do not have S2R as a component, or only one does not. In comparison, such statistics are 13% and 1%, respectively, for the textually similar duplicate bug reports. Such higher ratios of missing components might explain the lower textual similarity between each duplicate pair of their bug reports.

Table 3.9 also shows the component-level similarity between two bug reports from each duplicate pair. We see a lower component-level similarity for textually dissimilar duplicate bug reports. For example, on average, two bug reports from each of their

pairs are only 38% and 57% similar when OB and EB components are considered, whereas such statistics are 83% and 78%, respectively, for the textually similar duplicate pairs. Thus, even at the component level, textually dissimilar duplicate bug reports displayed lower similarity ratios.

In other words, missing components and component-level differences might have led to their overall textual dissimilarity. We also record several qualitative insights during our manual analysis of textually similar and textually dissimilar duplicate bug reports. They are outlined as follows.

(a) **Shared phrases.** Textually similar duplicate bug reports have more shared phrases (e.g., Bigram, Trigram) than unique words (e.g., unigram). For example, rather than the word "scroll", phrases such as "horizontal scroll", and "horizontal scroll installation" are more prevalent in these reports.

(b) **Components' prevalence.** Observed Behaviors (OB) and Expected Behaviors (EB) are more prevalent in textually similar duplicate bug reports, which could lead to their increased textual similarity. We found up to 90% overall similarity between two duplicate reports from this category.

(c) **Missing components.** We observed a higher percentage of missing components in textually dissimilar duplicate bug reports than in textually similar ones. Bug reports with missing components are likely to have lower similarity scores. Although we noticed minimal keyword overlaps, the root cause was mostly similar for both bugs from the same duplicate pair.

(d) **Unique phrases.** Textually dissimilar duplicate bug reports often use completely unique phrases, which could lead to their dissimilarity. We found that although the EB was somewhat similar, the OB and S2R components were written differently for the two bug reports of the same duplicate pair.

Summary of RQ_2: Textually similar and textually dissimilar duplicate bug reports are different in terms of their descriptive statistics (e.g., skewness), underlying semantics (e.g., t-SNE clusters), and prevalence of structural components. In particular, textually dissimilar duplicate bug reports often *miss important components* such as expected behaviors (EB) or steps to reproduce (S2R), which could lead to their textual dissimilarity within each pair.

3.3.3 Answering RQ_3: Does domain-specific embedding help improve the detection of textually dissimilar duplicate bug reports?

From RQ_1 and RQ_2, we see that textually similar and dissimilar duplicate bug reports could be different in their lexicon, underlying semantics, and structures. Our analysis above also shows that the performance gap between these two sets is the smallest when the ML-based approach is used for duplicate bug report detection (RQ_1). ML-based approaches might be able to capture more contextual information than the other approaches, which could be useful for the detection task [37, 38, 39].

In RQ_1, we used pre-trained word embeddings from GloVe to train our Siamese CNN model [52] for duplicate bug report detection. Pre-trained word embedding has proven to be invaluable for improving the performance of various NLP tasks (e.g., text classification [99], sentiment analysis [80]). However, GloVe has been pre-trained on natural language texts (e.g., Wikipedia) [54], which might not be relevant to the texts from bug reports. Thus, we use domain-specific embedding to retrain our Siamese CNN model. We set the max token size to 20,000 and the embedding dimension to 100, which are similar to the parameters used in RQ_1. We generate the embedding matrix with Skip-gram algorithm [57], use the whole dataset of 92,854 bug reports, and apply the same deep learning architecture to Siamese CNN, as used in RQ_1 [52].

Datasets constructed from bug-tracking systems are often heavily imbalanced. The number of duplicate bug reports is considerably smaller than that of non-duplicate bug reports [15]. Hence, we use *oversampling* [81] to handle the data imbalance problem during model training [81]. To the best of our knowledge, the original work [52] did not use sampling in their Siamese CNN model. However, for an in-depth investigation, we replicate another variant of the original DL-based model [52] applying oversampling. Table 3.10 summarizes the experimental results of our DL-based model for three different scenarios: *pre-trained embedding only* (original work [52]), *pre-trained embedding + oversampling*, and *domain-specific embedding + oversampling*. When compared between these model scenarios– pre-trained embedding + oversampling and domain-specific embedding + oversampling, we see the noticeable impact of domain-specific embeddings on duplicate bug report detection.

From Table 3.10, we see that in terms of F1-measure, the model's performance with textually dissimilar duplicate bug reports has increased by 1.18% for Eclipse,

Table 3.10: Impact of domain-specific embeddings (%) on duplicate bug report detection

Metric	Textually Similar	Textually Dissimilar
Eclipse		
AUC	56.31	53.54
Recall	51.00	51.00
Precision	52.00	51.00
F1	51.49	51.00
Firefox		
AUC	56.03	55.16
Recall	61.00	60.00
Precision	63.00	61.00
F1	61.98	60.50
Mobile		
AUC	64.98	62.25
Recall	70.00	69.00
Precision	73.00	69.00
F1	71.47	69.00

6.56% for Firefox, and 3.21% for Mobile. Furthermore, the AUC improved by 6.56% for Eclipse and remained comparable for the other two systems. Thus, domain-specific embeddings have a positive impact on detecting textually dissimilar duplicate bug reports. However, they have mostly negative impacts, except in a few cases, on detecting textually similar bug reports. From Table 3.10, we see that the model's F1-measure decreased by 7.24% for Eclipse and 4.32% for the Mobile system. Furthermore, the AUC decreased by 10.48% for the Mozilla system. Thus, while domain-specific embeddings have the potential to tackle the challenge of textual dissimilarity, they have a mixed impact on detecting duplicate bug reports.

Summary of RQ_3: The use of domain-specific embeddings (e.g., trained on bug reports) improves our model's performance for textually dissimilar duplicate bug reports (e.g., up to **6.56**% in F1-measure for Firefox). However, these embeddings have either negligible or negative impacts on detecting textually similar duplicate bug reports.

3.4 Threats to Validity

We identify a few threats to the validity of our findings. In this section, we discuss these threats and the necessary steps taken to mitigate them as follows.

Threats to internal validity relate to experimental errors and human biases [100]. Traditional bug tracking systems (e.g., Bugzilla) have thousands of reports whose quality cannot be guaranteed, which could be a source of threat. Bug reports often contain poor, insufficient, missing, or even inaccurate information [23]. To address the issue, we apply standard natural language preprocessing and token threshold to them and also check for missing features in each bug report. Another potential source of threat could be the replication and reproduction of existing work. The replication package was unavailable for the BM25 and Siamese CNN models, and we had to re-implement them. However, we did it carefully using standard libraries and corresponding papers, tuned the parameters, and reported their best results.

We use TF-IDF and cosine similarity to determine the textual similarity between any two duplicate bug reports. TF-IDF and cosine similarity have been frequently used to determine the textual similarity between two documents for the last 50 years [101]. Besides, we also used N-gram-based similarity and quartile analysis to systematically separate the textually similar and textually dissimilar duplicate bug reports (Section 3.2) and report the detailed similarity measures for replication (see Table 3.2 Thus, the threats concerning similarity calculation and construction of two bug report groups might be mitigated. When replicating and reproducing the existing methodologies (BM25, LDA+GloVe, Siamese CNN), we have carefully followed their original work.

Threats to conclusion validity. The observations from our study and the conclusions we drew from them could be a source of threat to conclusion validity [102]. In this research, we answer three research questions using 92,854 bug reports from three different subject systems and re-implementing three existing techniques. We use appropriate statistical tests (e.g., Wilcoxon Signed Rank) and report the test details (e.g., p-value, Cliff's delta) to draw any conclusion. Thus, such threats might also be mitigated.

Threats to construct validity relate to the use of appropriate performance metrics. We evaluate BM25 and LDA+GloVE techniques with Recall-rate@K and ML

model with AUC, precision, recall, and F1-measure, which have been used previously [23]. Thus, such threats might also be mitigated.

3.5 Related Work

3.5.1 IR-based duplicate bug report detection

IR approaches rely on the textual overlap between query bug reports and candidate bug reports for duplicate detection. Runeson et al. [11] first use a simple approach, namely Bag of Words (BOW), to tally the frequency of words and then use BOW-model to detect duplicate bug reports. They determine the similarity between two bug reports using cosine, Jaccard, and dice similarity measures. However, the BOW-based approach could be biased towards large documents and might not be able to capture the semantics of a bug report precisely [103]. Wang et al. [12] later improved this technique using TF-IDF [104] and quantify the similarity of two document vectors.

Later, they used BM25 [75] as a traditional IR-based model for duplicate bug report detection. Aggarwal et al. [15] demonstrate that BM25F, an improvement of BM25, is more suitable for weighting words in diverse domains. Like us, they also use domain-specific, categorical, and textual features. Sureka and Jalote [13] use n-gram models for textual similarity calculation during duplicate bug report detection. In particular, they focus on character-level language models rather than word-level ones. We also use n-gram-based similarity to separate textually similar and textually dissimilar duplicate bug reports.

Chaparro et al. [76] use three strategies to reformulate a query bug report and use IR to detect duplicate bug reports. Later, Cooper et al. [105] propose a duplicate bug report detection for video-based bug reports where they make use of text retrieval and computer vision methods. Since we focus on textual bug reports, their work might not be a great fit. In our research, we thus use BM25, a popular IR baseline, to investigate the impacts of textual dissimilarity on duplicate bug report detection.

3.5.2 Topic modeling-based duplicate bug report detection

IR-based approaches might suffer from Vocabulary Mismatch Problems [25]. Topic Modeling has the potential to tackle such problems concerning textual similarity calculation [66]. Alipour et al. [16] employ the Latent Dirichlet allocation (LDA) model to capture contextual information from history (e.g., prior knowledge on software quality) and leverage the information in duplicate bug report detection. Aggarwal et al. [15] capture domain-specific contextual information to improve duplicate bug report detection.

Nguyen et al. [35] combine IR and topic-based features to improve duplicate bug report detection. Recently Akilan et al. [51] propose a hybrid model that combines the Topic Modeling (e.g., LDA) with pre-trained word embedding (e.g., GloVe) for duplicate bug report detection. We replicate their technique carefully for our experiments, and detailed results can be found in Table 3.3.

In another research, Budhiraja and Shrivastava [106] combines Latent Dirichlet Allocation (LDA) and domain-specific word embeddings. Similarly, we leverage domain-specific embedding to counteract the impact of textual dissimilarity in duplicate bug report detection (RQ_3).

3.5.3 Machine learning and deep learning–based duplicate bug report detection

Unlike the above two methodologies, ML can detect non-linear relationships between any two bug reports for duplicate detection [37, 38, 39]. Sun et al. [19] first used a Support Vector Machine (SVM) to design a discriminative model for detecting duplicate bug reports. However, their approach lacks rigorous validation. Klein et al. [20] design several models using K-NN, Linear SVM, RBF, Decision Tree, Random Forest, and Naive Bayes to classify duplicate bug reports. Although clustering can be a useful technique in some contexts, it is not a suitable method for duplicate bug report detection due to its lack of interpretability, limited ability to handle complex and multi-dimensional data with multiple features (e.g., summary, description), and sensitivity to imbalanced dataset [107].

Deshmukh et al. [52] used the Siamese variations of CNN and RNN to design a

deep-learning model for duplicate bug report detection. We replicate their work for our experiments. Rocha and Carvalho [108] incorporate the attention mechanism into the Siamese network for semantic and context-based embedding. Xie et al. [109] propose an architecture, namely DBR-CNN, where CNN is used to encode the textual data and logistic regression to classify each pair of bug reports as either *duplicate* or *non-duplicate*.

To summarize, we replicate three existing techniques on duplicate bug report detection from IR, Topic-Modeling, and Deep Learning. Then we conduct experiments using a total of 92,854 bug reports to better understand the impacts of textual dissimilarity on duplicate bug report detection. To our best knowledge, this is the first attempt to comprehensively understand the impacts of textual dissimilarity on duplicate bug detection, which makes our work *novel*.

3.6 Summary

To summarize, automated detection of duplicate bug reports has been an active research topic for over a decade. However, existing approaches might not be sufficient to detect textually dissimilar but duplicate bug reports. In this paper, we thus perform a large-scale empirical study using 92,854 bug reports from three open-source systems to better understand the challenges of textual dissimilarity in duplicate bug report detection. First, we empirically demonstrate that existing techniques perform poorly in detecting textually dissimilar duplicate bug reports. Second, we found that textually dissimilar duplicates often miss important components (e.g., steps to reproduce), which could lead to their textual dissimilarity within the same pair. Finally, inspired by the earlier findings, we applied domain-specific embedding to duplicate bug report detection, which provided mixed results. All these findings above warrant further investigation and more effective solutions for detecting textually dissimilar duplicate bug reports.

Approximately 80% of the 327 software practitioners from tech giants (e.g., Google, Meta, Microsoft, Amazon, and Twitter) emphasized on the significance and challenges of two tasks from bug report management: duplicate bug report detection and bug localization [3]. Considering the high level of interest from practitioners, a comprehensive investigation into these tasks is warranted. Therefore, while this chapter

focuses on duplicate bug report detection, in Chapter 4, we conduct our second study targeting bug localization.

Chapter 4

Bug Localization

The first study in Chapter 3 investigates the challenges in detecting textually dissimilar duplicate bug reports, which could be 19%–23% of all duplicate pairs. Our analysis reveals the limitations of three existing approaches when dealing with textually dissimilar duplicates. While detecting duplicate bug reports remains a challenge for the developers, they also need to identify the location of a software bug as a part of bug report management [43]. Duplicate bug report detection and bug localization are interconnected within the bug report management. Once the duplicate bug report detection process effectively filters out redundant reports, the developers can concentrate on only unique and reduced sets of bugs, which could lead to cost-effective bug report management. In recent years, Information Retrieval (IR) methods have been frequently used to detect bugs due to their low cost, but they might not be sufficient for deep learning systems. Bugs in deep learning software systems are different from the typical bugs due to their multifaceted dependencies and extrinsic nature. In this chapter, we present our second study, where we investigate the challenges of detecting bugs in deep-learning software systems using IR methods.

The rest of this chapter is organized as follows. Section 4.1 provides an introduction to the research work. Section 4.2 presents our experimental design, datasets, and performance metrics. Section 4.3 discusses our experimental results and discusses our key findings. Section 4.4 discusses the threats to the validity of our work, and Section 4.5 discusses the related work.

4.1 Introduction

Software bugs are human-made errors in the code that prevent it from working correctly [1]. They are often prevalent in modern software systems and could range from hundreds to thousands in a single system.[2]. Due to the bugs in software systems, the global economy loses billions of dollars every year [4]. Developers also spend about

50% of their programming time dealing with software bugs and failures [4]. To fix any bug, the developers first need to identify the location of a bug within a software system, which is known as *bug localization* [10]. According to a recent survey, 49.20% of 327 software practitioners from several major tech giants (e.g., Google, Meta, Amazon, and Microsoft) consider bug localization as one of the most challenging tasks during software maintenance [3].

While localizing bugs in traditional software applications (a.k.a, non-deep learning software systems) remains a challenge, it could even be more challenging in deep learning applications. Unlike bugs in non-deep learning software systems, deep learning-related bugs could be hidden in the source code, training data, trained models, or even deployment scripts [29, 30, 31]. Besides, the use of popular deep learning libraries (e.g., PyTorch, Caffe, and TensorFlow) could lead to complex bugs [32].

Given the prevalence and costs of software bugs, any automated support to localize the bugs can greatly benefit software practitioners. Over the years, many approaches have been designed to localize bugs in traditional software systems using information retrieval [43, 44, 45, 46], dynamic program analysis [47, 48], and deep learning [33, 34, 49]. However, due to the significant differences between traditional and deep learning bugs, these existing solutions might not be adequate for localizing the bugs in deep learning applications.

To date, there exist only a few techniques for detecting bugs in deep learning systems. Wardat et al. [34] propose to localize bugs in the Deep Neural Network (DNN) using dynamic and statistical analysis. However, the solution focuses on model and training bugs only, strongly depends on the Keras library, and achieves a low accuracy, which presents significant challenges for widespread adoption in the industry. Similarly, Kim et al. [50] use basic IR-based techniques, such as rVSM and BM25, to localize bugs in deep-learning applications but report poor performance without any comprehensive analysis or explanation. Interestingly, at least 30 techniques adopt IR to locate bugs in traditional software systems due to their computational efficiency and lightweight nature [43, 44, 45, 46, 50, 60]. They were also reported to perform comparably to the complex models (e.g., LDA) [110]. Unlike deep learning-based techniques, they rely on the textual similarity between bug reports and source code as a proxy of suspiciousness, which is simple and explainable. Thus, the potential of

existing solutions, especially IR-based techniques, for localizing bugs in deep learning applications is not well understood to date.

In this research, we conduct a large-scale empirical study to better understand the challenges of locating bugs in deep learning applications. First, we collect a total of 2,365 bugs from deep-learning applications and 2,913 bugs from traditional software applications, and *empirically* show how existing techniques (BugLocator [43], BLUiR [44], BLIA [60]) perform in locating bugs from deep learning applications. Second, we categorize our collected bugs based on an existing bug taxonomy [62] and found that certain bugs from deep learning applications (e.g., GPU bugs) are more difficult than others to locate using IR-based techniques. Finally, we found that deep learning bugs are connected to artifacts other than source code (e.g., GPU, training data, external dependencies) and are prone to be extrinsic in nature, which might explain the poor performance of IR-based techniques for these bugs.

(a) **RQ_1: How effective are the existing IR-based approaches in localizing bugs from deep learning software systems?**

We evaluated the performance of three existing IR-based approaches (BugLocator [43], BLUiR [44], and BLIA [60]) using two datasets – Denchmark [59] and BugGL [58]. We found that the performance measures of existing IR-based techniques are poorer (e.g., 14.50% less MAP for BugLocator, 12.80% for BLUiR, and 18.40% for BLIA) in localizing bugs from deep learning software systems than that of non-deep learning software systems. Our statistical tests (e.g., t-test [111], Cohen's D [112]) also report that their performance is significantly lower. Although our findings reinforce the existing understanding and belief about the challenges of the bugs in deep learning software systems, we also substantiate them with solid empirical evidence and demonstrate the performance gap of existing solutions in localizing the two categories of bugs.

(b) **RQ_2: How do different types of bugs in deep learning software systems impact bug localization?**

We use an existing taxonomy [62] to classify the bugs in deep learning systems and evaluated the performance of three existing techniques for each type of bug. We found that 64.80% of the bugs from deep learning software systems are related to deep learning (e.g., model, training), whereas 35.20% of the bugs

are not related to deep learning. We also found BugLocator and BLUiR to be promising for detecting model and tensor bugs, respectively, but all three IR-based techniques experienced difficulty localizing GPU bugs. Our analysis also showed that bug reports from deep learning applications contain more code snippets (83.11%) than traditional software (33.24%). Unfortunately, that does not help much in bug localization, as code snippets alone might not be sufficient to capture the intricacies of the model architecture and training processes. Thus, our analysis offers valuable insights regarding the nature of different bugs in deep learning software systems and highlights the specific strengths and weaknesses of existing IR-based techniques, which could be useful for improving bug localization in deep learning software systems.

(c) **RQ_3: What are the implications of extrinsic bugs in deep learning systems for bug localization?**

Bugs triggered by external entities (e.g., third-party libraries, GPU) are called *extrinsic bugs* [42]. Given the frequent use of deep learning libraries and their external dependencies, the bugs in deep learning applications could be extrinsic [32]. Since the existing techniques, including IR-based ones, mostly focus on intrinsic bugs, we investigate how they deal with the bugs from DL systems. We found deep learning software systems have 40.00% extrinsic bugs, which is almost four times higher than non-deep learning software systems. We also found that the performance of bug localization for extrinsic bugs degrades significantly (e.g., 7.18% less MAP for BLIA) compared to intrinsic bugs (Table 4.11). We also found a significant correlation between the bugs in deep learning software systems and the extrinsic factors, which could be valuable insight for designing effective bug localization solutions for deep learning software systems.

4.2 Study Methodology

Fig. 4.1 shows the schematic diagram of our conducted study. First, we collect bug reports from two benchmark datasets for two different software systems: deep learning software systems [59] and traditional software systems [58]. Then, we contrast the performance of three existing IR-based techniques [43, 44, 60] in locating bugs between

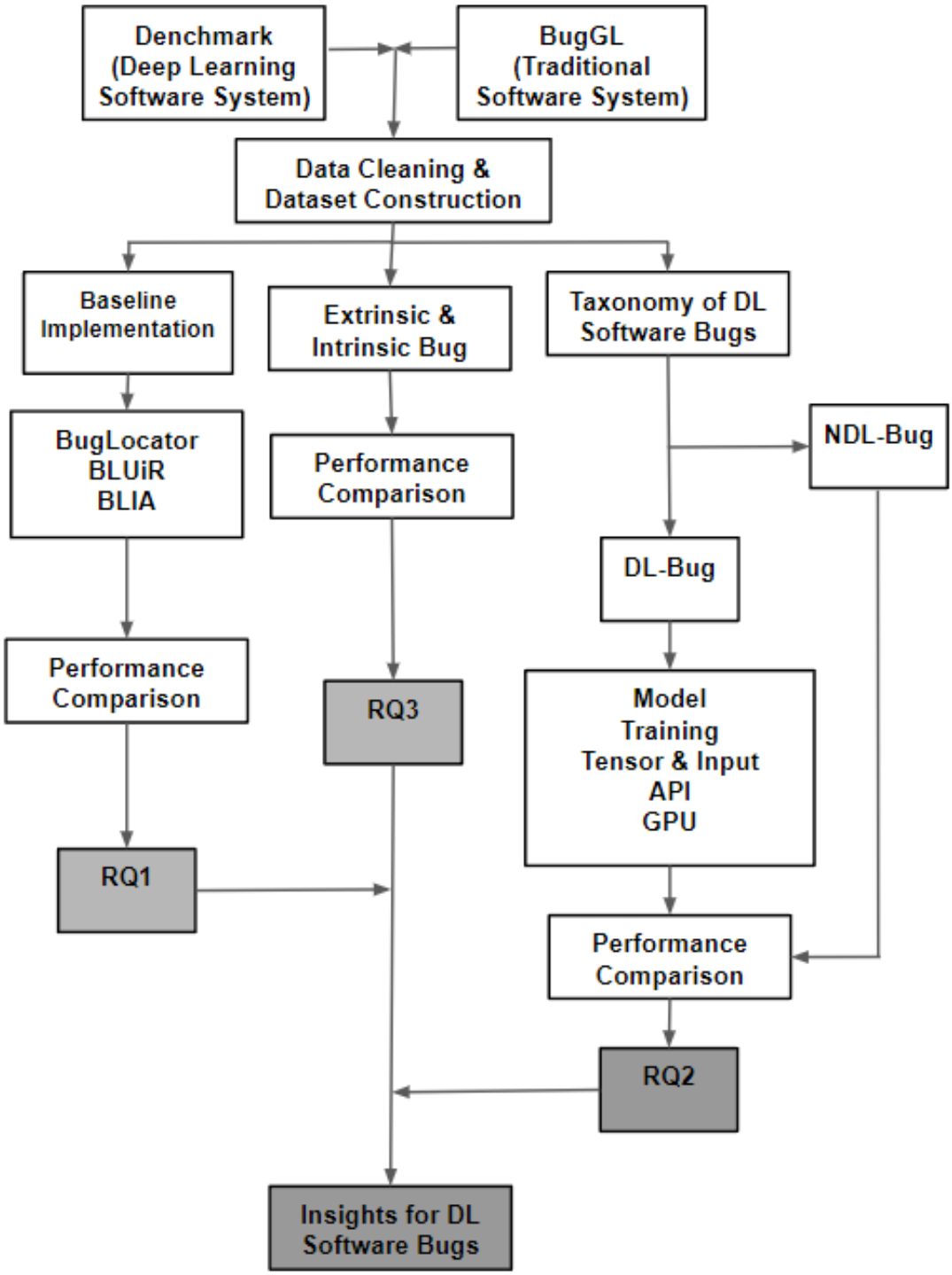


Figure 4.1: Schematic diagram of our conducted study on bug localization

deep learning software systems [59] and traditional software systems [58]. Second, we perform an in-depth analysis to understand the challenges of localizing different types of deep-learning bugs. Finally, we investigate the influence of extrinsic factors on deep learning bugs and their impact on bug localization. This section discusses the major steps of our study design as follows.

4.2.1 Construction of dataset

Dataset collection. In our study, we use two benchmark datasets – BugGL and Denchmark – that have been previously used by the literature [50, 58]. BugGL contains only Python-based traditional software bugs (a.k.a non-deep learning software systems), whereas Denchmark focuses on deep learning-based software bugs. BugGL contains a total of 2,913 bug reports from 12 Python projects [58]. On the other hand, the Denchmark dataset contains a total of 2,365 bug reports from 136 deep learning software projects (Python based) [59].

Our study focuses on Deep Learning Software Systems (DLSW). Here, we mean software systems that incorporate deep learning modules, which go beyond deep learning libraries. Our DLSW dataset includes deep learning frameworks (e.g., Apache MXNet), engines (e.g., Microsoft ONNX Runtime), libraries (e.g., OpenCV), tools (e.g., Fairseq), applications (e.g., PhotoPrism), and platforms (e.g., Kubeflow) [59]. DLSW differs significantly from traditional software systems (a.k.a NDLSW) due to the complexity of model integration, intricate interactions between deep-learning libraries, and multifaceted dependencies.

Data cleaning and pre-processing. After collecting the data from two benchmarks, we cleaned and preprocessed them using a set of steps.

Corpus creation. We begin by downloading the latest version of the code repositories, ensuring that we have the most up-to-date code for analysis. Next, we extract Git tags, which serve as reference points for identifying specific versions of the code. To create a robust search space for bug localization, we detect the project version from each bug report and capture the corresponding version of code from the repositories. We used heuristics from Kim et al. [50] to capture the buggy versions of a project.

Query construction. In IR-based bug localization, bug reports are treated as

Table 4.1: Study dataset for bug localization

Dataset	Projects	Bug Reports	Source Files		Buggy Files	
			Mean	Max	Mean	Max
Denchmark	136	1795	441.60	3,559	2.60	227
BugGL	12	1795	420.50	3,306	2.35	198

queries that are executed to detect the relevant source documents from the corpus. We construct a repository of bug reports by parsing the original datasets (e.g., Denchmark & BugGL) and extracting important information such as bug IDs, descriptions, and timestamps. We use timestamps to divide the bug reports by relevant project versions. We construct the query by extracting tokens from the title and description of bug reports, removing stop words, stemming each word, and splitting the tokens.

Meta data extraction. We also capture the historical context of the bugs by extracting commit history information for the repositories, including commit messages, authors, timestamps, and code changes history, using a set of heuristics provided by Youm et al. [60]. This information provides valuable insights into the evolution of the codebase and the bugs over time.

Ground truth construction. To evaluate the performance of the bug localization approaches, we collect ground truth files that contain the correct locations of bugs in the code from both of the original datasets.

To ensure a fair performance comparison of the bug localization techniques between Denchmark and BugGL, we have selected an equal amount of data from both datasets using probability sampling (1793 bug reports from each dataset), which have $\sim$95% confidence interval and 5% error margin [113]. We use the principle of randomization for selecting the subsets [114] to avoid any bias. We also took measures to prevent any overlap of bugs or projects between the two datasets.

4.2.2 Replicating of existing techniques for experiments

At least 30 approaches adopt IR methods for localizing software bugs due to their computational efficiency and lightweight nature [17]. Unlike Deep Learning (DL)-based techniques, they rely on the textual similarity between bug reports and source code as a proxy of suspiciousness, which is explainable. Our primary objective was to

better understand the challenges of localizing bugs in DL applications. Given their popularity and explainability, we thus used IR-based techniques in our experiments.

To determine the potential of IR-based methods for detecting deep learning bugs, we used three baseline techniques: BugLocator [43], BLUiR [44], and BLIA [60]. IR-based localization can be adapted to different granularity levels (e.g., method, file). We chose file-level granularity since each of the selected baselines frequently used this granularity. Most of the recent IR-based models are based on these three primary approaches, with incremental improvements [43, 44, 60]. Thus, they can be considered as a representative sample of existing IR-based approaches for bug localization. IR-based approaches assume an explainable relationship between bug reports and source documents (e.g., lexical similarity) during bug localization. In our research, we thus deliberately selected IR-based methods to better understand the characteristics and challenges of deep learning bugs while mitigating any effect of the compounding factors.

BugLocator [43] uses rVSM, which is a logarithmic variant of term frequency, to detect relevant source code files against a bug report. It also calculates the SimiScore (a measure of similarity between a newly reported bug and previously fixed bugs based on their bug reports) and combines with rVSM to calculate the final relevancy score. The relevant source code files are then ranked based on their combined scores, and the top-K documents are marked as buggy. Many all subsequent IR-based techniques adopted this method due to its simplicity and explainability. Hence, we chose this method as our first baseline.

BLUiR [44] uses Abstract Syntax Tree (AST) parsing to extract class, method, variable, and comment fields from a source code document. It also captures two fields from each of the bug reports (summary & description). A total of eight separate searches are performed ((summary, description) x (class, method, variable, comment)). The document scores are summed for the final score to rank buggy files. BLUiR technique leverages structured elements from source code and bug reports to localize bugs using IR. We thus choose this as another baseline technique for our study.

BLIA [60] integrates several items such as textual similarity between bug reports and source documents [43], code structures [44], version control history [46],

stack trace analysis [115], and code change analysis in the IR-based bug localization. While bug reports and source code are useful, code change history can also assist in bug localization by identifying the changes likely to induce a bug. BLIA has outperformed several previous techniques: BugLocator [43], BLUiR [44], Amalgam [46], BRTracer [115], which makes it suitable as the third baseline technique for our study.

Since the original authors' replication package for these techniques was unavailable, we used the publicly available replication package of BugLocator and BLUiR from Lee et al. [116]. We carefully adapted the BLIA according to the technique from the original replication package [60] to our datasets.

4.2.3 Performance evaluation

We use three performance metrics for our study — Top-K accuracy (Top@K), Mean Average Precision (MAP), and Mean Reciprocal Rank (MRR). These metrics have been frequently used by the relevant literature [43, 44, 46, 49, 60, 115].

4.2.3.1 Top@K

Top-K accuracy (Top@K) measures the percentage of bug reports for each of which at least one of the buggy files were present in the top-k retrieved files. We have used K= 1, 5, 10 for this research.

4.2.3.2 Mean Average Precision

Precision@K measures precision at each individual buggy source document's position in a ranked list. Average Precision@K (AP) computes the average precision for all buggy documents in a search query list. Mean Average Precision (MAP) is the average AP@K value across all queries in a system.

$$\text{AP} = \frac{1}{D} \sum_{k=1}^{D} P_k \times \text{buggy}(k) \tag{4.1}$$

$$\text{MAP} = \frac{1}{|Q|} \sum_{q \in Q} \text{AP}(q) \tag{4.2}$$

Here, AP represents the Average Precision, and D refers to the number of total results for a query. k represents the position in the ranked list, P_k denotes the precision

calculated at the k-th position, buggy(k) determines whether the k-th result in the ranked list is buggy or not.

4.2.3.3 Mean Reciprocal Rank

Mean reciprocal rank (MRR) calculates the average of the reciprocal ranks for a set of queries.

$$\text{MRR(Q)} = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{\text{firstRank}(q)} \tag{4.3}$$

where MRR(Q) represents the Mean Reciprocal Rank for a set of queries Q, $|Q|$ represents the total number of queries in the set Q, $q \in Q$ represents each query in the set Q, firstRank(q) represents the rank of the first correctly retrieved buggy document for the query q.

4.3 Study Finding

4.3.1 Answering RQ_1: How effective are the existing IR-based approaches in localizing bugs from deep learning software systems?

Table 4.2 compares the performance of three IR-based approaches in bug localization between Deep Learning Software Systems (Deep Learning Software Systems (DLSW)) and Non-Deep Learning Software Systems (Non-Deep Learning Software Systems (NDLSW)). We used three different evaluation metrics – Top@k, MRR, and MAP, for our comparative analysis. We see notable differences in performance between the two types of systems when evaluating BugLocator, BLUiR, and BLIA methods. In particular, for BugLocator, the difference in Top@1 is relatively small – 4.60%, while the differences in MAP and MRR are 14.50% and 15.50%, respectively. On the other hand, for BLUiR, the difference in Top@1 is more apparent (e.g., 11.00%), and the differences in MAP and MRR are also substantial, 12.80% and 9.70%, respectively. Finally, for BLIA, the difference in Top@1 is 8.90%, while the differences in MAP and MRR are 18.40% and 19.90%, respectively. Fig. 4.2 visualizes the MAP measures using bar plots, and the differences are clearly visible. Overall, the results show that all three IR-based approaches for bug localization perform lower when localizing bugs in deep learning software systems, and the trend is consistent across all three metrics.

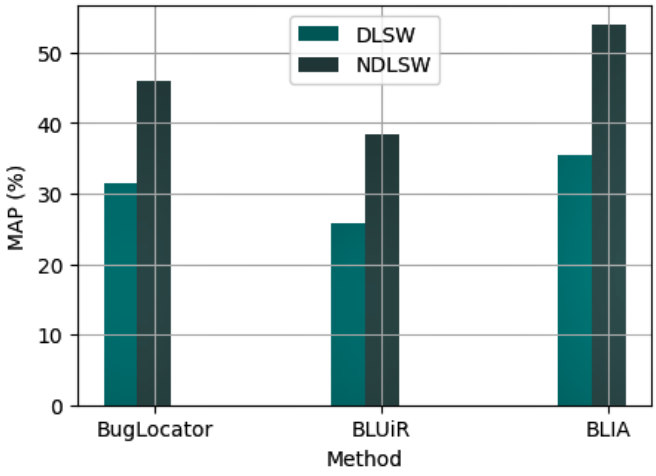


Figure 4.2: Performance comparison of existing IR-based approaches (BugLocator, BLUiR, BLIA) between deep learning software systems and non-deep learning software systems

Table 4.2: Experimental result of existing IR-based approaches (BugLocator, BLUiR, BLIA) for bug localization

Method	**Top@1**	**Top@5**	**Top@10**	**MRR**	**MAP**
DLSW					
BugLocator	0.344	0.647	0.745	0.371	0.314
BLUiR	0.201	0.472	0.585	0.356	0.257
BLIA	0.411	0.690	0.790	0.423	0.355
NDLSW					
BugLocator	0.390	0.671	0.794	0.526	0.459
BLUiR	0.311	0.575	0.680	0.453	0.385
BLIA	0.500	0.768	0.855	0.622	0.539

DLSW= Deep Learning Software Systems, **NDLSW**=Non-Deep Learning Software System

We also perform statistical tests to determine the significance of the performance gap between the two types of systems (DLSW and NDLSW) (Table 4.3). We took the results of MRR and MAP for each of the three approaches. Then, we performed *Shapiro-Wilk normality test* [92], which reported normal distribution for those metrics. Then we used appropriate parametric and effect size tests to compare the result values from the two types of systems. For the normal distribution, we used *paired t-test* as the parametric test [111]. In all significance tests, the p-values were less than the threshold (0.05) for each of the three approaches. Thus, the null hypothesis

Table 4.3: Statistical tests for the performance gap between the deep learning software system and non-deep learning software system using existing IR-based techniques (BugLocator, BLUiR, BLIA)

Method	**Metric**	**Significance (p-value)**	**Effect Size (Cohen's D)**
BugLocator	MRR	0.00009063**	Medium (0.3954)
	MAP	0.000035**	Medium (0.4182)
BLUiR	MRR	0.0395*	Medium (0.2722)
	MAP	0.0298*	Medium (0.3357)
BLIA	MRR	0.00000505***	Medium (0.4625)
	MAP	0.00000576***	Medium (0.4597)

can be rejected for all comparisons. In other words, the performances of all three IR-based techniques significantly differ between the two types of systems.

While the significance of a result indicates how probable it is that it is due to chance, the effect size indicates the extent of the difference [117]. Hence, we performed the Cohen's D effect size test [112], and our experiments found a *medium* effect size for all cases (Table 4.3). Thus, our results from effect size tests reinforce the above finding from significance tests. In other words, the existing techniques perform significantly poorly in localizing bugs from deep-learning software systems. Even though our findings above match natural intuition, we performed extensive experiments using three different baselines, which resulted in strong empirical evidence. Thus, our findings reinforce the existing understanding and belief about the challenges of the bugs in deep learning software systems but we also substantiate it with solid empirical evidence and demonstrate the performance gap of existing solutions in localizing the two categories of bugs.

Summary of RQ_1: We compare the performance of three IR-based bug localization approaches between deep learning software systems and non-deep learning software systems using three evaluation metrics. Our findings show that all three approaches perform significantly lower (e.g., **18.40%** less MAP for BLIA) when localizing bugs from deep learning software systems.

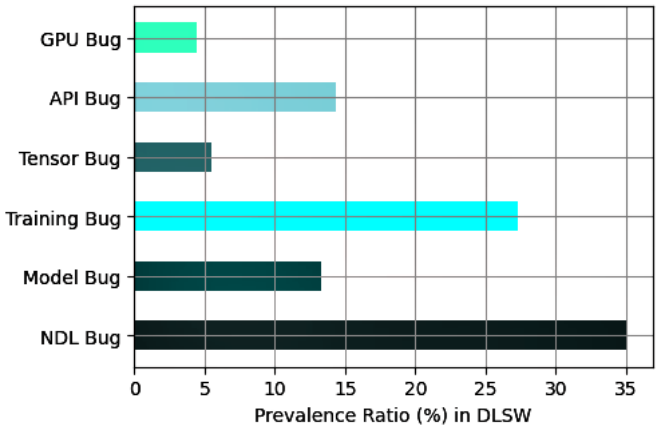


Figure 4.3: Prevalence ratio of each category of bugs from deep learning software systems

4.3.2 Answering RQ_2: How do different types of bugs in deep learning software systems impact bug localization?

In this research question, we investigate the characteristics of bugs in deep learning software systems through manual analysis. First, we employ stratified random sampling to construct a random sample that accurately represents the original data distribution with a balanced presence of instances from each class [118]. We select 385 bugs from both of our datasets that have ∼95% confidence interval and 5% error margin.

We performed our first manual analysis using 385 bug reports from Denchmark. We manually labeled them as deep learning-related (DL) bugs and non-deep learning-related (NDL) bugs. Then we labeled the deep learning-related bugs into five categories: Model, Training, Tensor, API, and GPU, based on the existing taxonomy of Humbatova et al. [62]. We also analyzed the bug reports, associated developers' discussions, and bug-fix code changes as a part of the labeling. Two authors of this work labeled the sample dataset separately and achieved a Cohen's kappa [119] of 0.80, which indicates a substantial agreement between the authors. Our manual analysis above was documented using an Excel sheet, with a total of ≈55 hours spent by each author.

Prevalence ratio of deep learning-related bugs: We found that 64.80% of the bugs from deep learning software systems are related to deep learning algorithms (DL bugs). In particular, we found 27.30% training bugs, 13.30% model bugs, 5.50% tensor bugs, 14.30% API bugs, and 4.40% GPU bugs (Fig. 4.3). This distribution

Table 4.4: Experimental result of existing bug localization techniques (BugLocator, BLUiR, BLIA) of each category of bugs in deep learning software systems

Method	NDL Bug	Model	Training	Tensor	API	GPU
MAP						
BugLocator	0.362	0.368	0.312	0.235	0.446	0.183
BLUiR	0.292	0.293	0.359	0.601	0.258	0.266
BLIA	0.437	0.357	0.345	0.448	0.395	0.290
MRR						
BugLocator	0.417	0.532	0.386	0.358	0.478	0.223
BLUiR	0.334	0.387	0.427	0.682	0.289	0.411
BLIA	0.381	0.472	0.419	0.553	0.447	0.457

MAP= Mean Average Precision, **MRR**=Mean Reciprocal Ranking

informs us where the debugging efforts should be concentrated. Our findings also indicate that the majority of DL bugs are related to model training. Training is a crucial step in deep learning that involves large amounts of data, complex learning algorithms, and optimization techniques, making it more susceptible to bugs.

Prevalence ratio of non-deep learning-related bugs: We found that 35.20% of the bugs from deep learning software systems are not related to deep learning algorithms (NDL bugs). Non-deep learning-related bugs do not directly affect the functionality of the deep learning model, but they still lead to unexpected, erroneous behaviors in a software system. Bug 1426 in Table 2.3 illustrates an example of an NDL bug, which occurs when the tests from the CI pipeline are spread across multiple Windows machines. Although it is not directly connected to the deep learning module, it originated from the PyTorch-Ignite project, which is indeed a deep learning software system.

Localization of bugs in deep learning software systems: To gain a deeper understanding of the challenges in localizing deep learning bugs, we further analyze our results from RQ_1 for each category. To do this, we used stratified random sampling to select 100 samples from each category of bugs and analyze the performance of our baselines. To ensure the robustness of our findings, we repeated this process three times and used different sample data each time. We then averaged the results obtained from the three evaluations and presented them in Table 4.4. We conducted an in-depth analysis of each type of DL bug to understand their inherent challenges and gain a deeper understanding of the factors impacting the overall performance of

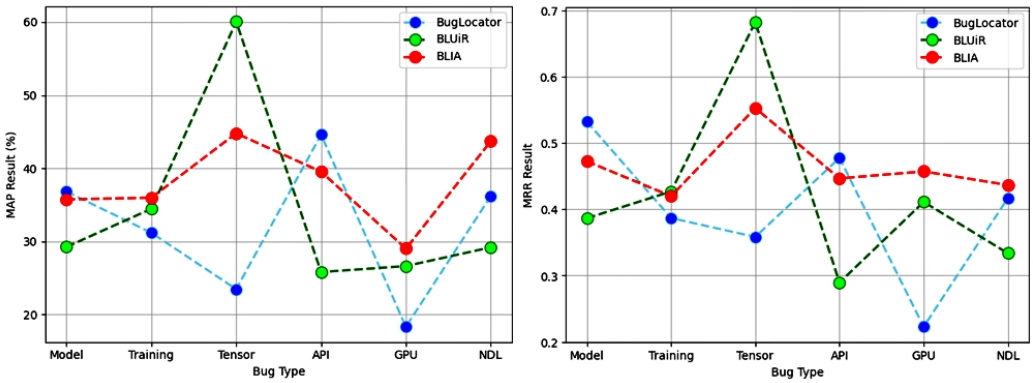


Figure 4.4: Performance of existing IR-based bug localization techniques (BugLocator, BLUiR, BLIA) for each type of bug in deep learning software systems

Table 4.5: Example of a model bug

Model Bug (Bug ID: 313)
Title
A bug in GPT2Tokenizer
Description
GPT2Tokenizer fails to recover a sentence \"BART is a seq2seq model.\" with encoded ids of it. The output sentence is \"BART is a seqseq model.\". It should be related to numbers' processing. A script to show the bug is here: https://github.com/tanyuqian/texar-pytorch /blob/master/examples/bart/gpt2_tokenizer_bug.py

IR-based techniques. The analysis with examples is discussed as follows.

- **Model bugs:** From Table 4.4, we see that BugLocator performs the highest for model bugs, outperforming BLUiR and BLIA. This could be attributed to BugLocator's ability to capture syntactic and semantic information from the bug reports, source code, and similar fixed bugs. Model bugs are often connected to a model's type, properties, and layers. According to our observation, the texts used to describe the issues in bug reports have significant vocabulary overlap with that of the model.

  Table 4.5 shows an example bug report that discusses a model bug from the CASL.ai project. Fig. A.1 shows a code snippet responsible for the bug. The bug in the GPT2Tokenizer lies in the bpe method, causing faulty tokenization

and impacting the functionality of the GPT2 language model, thus being considered a model bug. The incorrect character merging during byte pair encoding leads to faulty tokenization.

BugLocator incorrectly retrieves "SentencePieceTokenizer.py" (Fig. A.2) as the Top@1 result. It uses lexical overlap between bug reports and source code to identify buggy files. We found four main keywords from the bug report (Table 4.5) — GPT2Tokenizer, recover, seq2seq, and model— overlapping with an incorrect file (i.e., SentencePieceTokenizer.py). This lexical similarity could have led to an incorrect localization. Interestingly, the ground truth file was retrieved at the top 8th position (Fig. A.1) by BugLocator. According to our analysis, the bug report contains a specific keyword (e.g., GPT2Tokenizer) that matched the actual bug's characteristics and class. BLIA retrieves the same ground truth file at the 17th position, which is less than ideal.

On the other hand, the BLUiR approach retrieves ground truth code at a very low position (Top@131). Leveraging the similarity between bug reports and source code elements might not be effective for locating model bugs. In this example of a model bug (Table 4.5), the tokenizer bug is illustrated using an example sentence to tokenize – 'BART is a seq2seq model.' in the bug report. BLUiR incorrectly retrieved the source code file with the 'Seq2Seq' class at the Top@1 position (Fig. A.3). One possible explanation might be because some of the important keywords (e.g., 'seq2seq', 'encode', and 'model') from the bug report aligned with incorrect code elements (e.g., 'seq2seq' class) that were not relevant to the actual bug. The misalignment occurs due to BLUiR's heavy reliance on structural elements (e.g., class, method, variable) and less consideration of the semantic similarity with the bug report.

- **Training bugs:** All three IR-based approaches perform poorly in localizing training bugs. As shown in Table 4.4, BLUiR performs slightly better than BugLocator and BLIA. Table 4.6 exemplifies a training bug in the fast.ai project where the combined usage of Gradient Accumulation and the MixedPrecision Callback leads to improperly scaled and artificially high training loss values. To understand why BLUiR performed slightly better than other techniques in

Table 4.6: Example of a training bug

Training Bug (Bug ID: 3048)
Title
Gradient Accumulation + Mixed Precision shows artificially high training loss
Description

OB: The bug occurs when Gradient Accumulation
and the MixedPrecision Callback are both used.
Gradient Accumulation runs before Mixed Precision
and causes the after_backwards to not be run,
meaning that the loss is not unscaled before it is logged.
This means that very large losses, such as 6000000+, are to be logged.
S2R: seed=random.randint(0,2**32-1)
with no_random(seed):
db=synth_dbunch(bs=8,n_train=1,n_valid=1,cuda=True)
learn = synth_learner(data=db)
learn.fit(1, lr=0.01)
#start without gradient overflow
max_loss_scale=2048.0
with no_random(seed):
db=synth_dbunch(bs=1,n_train=8,n_valid=8,cuda=True)
learn = synth_learner(data=db,cbs=[GradientAccumulation(n_acc=8)])
learn.to_fp16(max_loss_scale=max_loss_scale)
learn.fit(1, lr=0.01)
The training loss will be very high, 5000+ for fp16.
fp32 will be reasonable
EB: Similar training loss
between the fp32 and fp16 versions. <2 difference in loss.

EB=Expected Behaviour, **S2R**=Steps to Reproduce, **OB**=Observed Behaviour

locating training bugs, we selected an instance where BLUiR successfully retrieved the ground truth file (Fig. A.4) at the Top@2 ranking. The finding can be explained since BLUiR can leverage structured elements from source code and bug reports. In the given bug report (Table 4.6), the bug causes artificial high training loss when both Gradient Accumulation and MixedPrecision Callback are used. BLUiR was able to locate this training bug from the use of structured IR, which effectively captured the hierarchical structure of code, including classes, methods, and variables. This allowed for the identification of similarities between the code structure and the bug report.

Table 4.7: Example of a tensor bug

Tensor Bug (Bug ID: 13760)
Title
nd.slice does not return empty tensor when begin=end
Description
OB: For mxnet.ndarray.slice(data, begin, end), if begin=end, it does not return an empty tensor. Instead, it returns a tensor with the same shape as the data. Environment info: .... S2R: import mxnet.ndarray as nd a = nd.normal(shape=(4, 3)) nd.slice(a, begin=0, end=0) nd.slice(a, begin=2, end=2) Detailed BR: https://github.com/apache/mxnet/issues/13760

BR=Bug Report, **S2R**=Steps to Reproduce, **OB**=Observed Behaviour

On the other hand, BugLocator has the lowest performance in localizing training bugs, suggesting that similarity analysis between bug reports and source code might not be sufficient for identifying these bugs. For the same bug example (Table 4.6), BugLocator ranked the ground truth file at Top@23 while incorrectly selecting a different file (Fig. A.5) as the Top@1 buggy file. In this case, the bug report and the incorrectly retrieved file by BugLocator share more than 10 important keywords (e.g., "GradientAccumulation", "loss", "after-backward"), which contributes to a high similarity score with the incorrect file than the ground truth file.

- **Tensor bugs:** BLUiR outperforms BugLocator and BLIA in localizing tensor bugs, which may be attributed to its ability to analyze source code information. Tensor bugs are more likely to be connected with code structures (e.g., incorrect or invalid tensor shape) [62]. BLUiR's ability to leverage structural information from the code might have led to its higher performance.

  BLUiR successfully retrieved the buggy file in the Top@1 position for the tensor bug in Table 4.7. The bug report clearly described the issue with the "nd.slice" function in MXNet, where it should return an empty tensor when the "begin"

and "end" parameters are equal. Instead, it returns a tensor with the same shape as the data. By parsing the code's AST, BLUiR identified the relevant code snippet in the testslice() function and found similarity with the bug report, which led to the accurate localization of the bug.

BLIA retrieved the ground truth file at Top@12. However, our analysis revealed that incorporating the stack trace information negatively impacted bug localization performance. Stack trace information might be less effective for tensor bugs because they are typically related to data manipulation and computations rather than code execution flow or call stack. By excluding the stack trace from the BLIA approach, the ranking improved to Top@9 from Top@12.

BugLocator's difficulty in accurately locating tensor bugs is demonstrated by its retrieval of the ground truth file at Top@47 (Fig. A.6) and the incorrect file at Top@1 (Fig. A.7). It can be attributed to its reliance on textual similarity. We found that BugLocator's emphasis on semantic similarity from the overlapping of trivial words (e.g., environment, system, and hardware) with the incorrect file (Fig. A.7) led to the incorrect ranking.

- **API bugs:** From Fig. 4.4, we observe that BugLocator and BLIA achieve relatively high performance in localizing API bugs with MAP values of 44.60% and 39.50%, respectively. In contrast, BLUiR performs poorly, with a MAP value of 25.82%. One possible explanation is that BLUiR relies heavily on the structural information of the source code, which may not be effective in identifying bugs related to APIs (e.g., incorrect API calls). These bugs may not involve changes in the code structure in the source code but instead involve issues in API usage or interaction. BugLocator and BLIA, on the other hand, may capture the textual similarity between bug reports and source code more effectively, enabling them to localize API bugs better. In the example involving an API bug in MXNET 1.4.0 (Table 3.10, BugLocator successfully ranked the buggy code at the 4th position, while BLIA ranked it at the 5th position (Fig. A.8. However, when stack trace information was included and commit history was omitted, BLIA improved its performance and ranked the bug in the 2nd position. This demonstrates that stack trace information is helpful in locating

Table 4.8: Example of an API bug

API Bug (Bug ID: 13862)
Title
[1.4.0] unravel_index no longer works with magic '-1' in shape parameter as in 1.3.1
Description
OB: The unravel_index op seems to no longer correctly work with 'magic' shape values, such as '-1's. The following example still works with mxnet 1.3.1, but does not on the latest master (it returns all zeros in the result without throwing an error) or 1.4.0. We have a use case for this in Sockeye. Environment info (Required): ... S2R: Input data taken from Sockeye unit tests. x = mx.nd.array([335, 620, 593, 219, 36], dtype='int32') mx.nd.unravel_index(x, shape=(-1, 200)) With mxnet==1.5.0b20190111, the result is incorrect: With mxnet==1.3.1, the result is correct: However, if the shape parameter is fully specified (shape=(5,200)), mxnet==1.5.0b20190111 returns the correct values. Detailed BR: https://github.com/apache/mxnet/issues/13862

BR=Bug Report, **S2R**=Steps to Reproduce, **OB**=Observed Behaviour

API bugs by tracing the execution flow and highlighting function calls and interactions within the API. In contrast, commit history provides a high-level overview of code changes but lacks the specific details needed for pinpointing the exact cause of an API bug.

On the other hand, BLUiR performed poorly, as it ranked the buggy file in the top 38. In this example (Fig. A.9), BLUiR retrieved a file containing the class "NDArray" due to parsing the code through the AST, even though the bug was related to the MXNet API. Utilizing the code structure is not particularly advantageous when it comes to identifying bugs in API usage.

- **GPU bugs:** Our investigation into the deep learning bugs reveals that GPU bugs which are related to the usage of GPU devices while working with DL, are the most difficult to localize for all three existing IR-based approaches. One

Table 4.9: Example of a GPU bug

GPU Bug (Bug ID: 1238)
Title
How to use multiple GPUs?
Description
I want to use a single machine with multiple GPUs for training, but it has no actual effect. OB: Only one single GPU is doing all the computations, the other three remain idle. When following @FontTian and inserting distribution_strategy=strat into the initialization of the image classifier, the same error RuntimeError: Too many failed attempts to build the model occurs. The same happens when adding tuner='random' to ak.ImageClassifier. As suggested by @haifeng-jin, I ran a basic KerasTuner example on 4 GPUs which worked just fine. Furthermore, in #440 (comment), I read that the clear_session() before every run might wipe out the GPU configuration. Removing this line from the code did not change anything with respect to the errors/problems stated above. I am specifying 4 GPUs (out of 8) to train the current model in a distributed fashion, using tf.distribute.MirroredStrategy( ) since tf.keras.utils.multi_gpu_model( ) is deprecated and removed since April 2020. S2R: def make_model(ckpt_path, max_try = 1): ..... run_search(checkpoint, max_try = 3) Detailed BR: https://github.com/keras-team/ autokeras/issues/1238

BR=Bug Report, **S2R**=Steps to Reproduce, **OB**=Observed Behaviour

possible explanation could be the complex nature of GPU bugs, as they can be triggered by a variety of factors, such as hardware and software compatibility issues, and might not even be located in the source code (a.k.a extrinsic GPU bug) [62]. We found 17.65% of the GPU bugs that can be found in the source code (a.k.a intrinsic GPU bug) (Table 4.10), i.e., the wrong reference to a GPU device, failed parallelism, incorrect state sharing between subprocesses, and faulty transfer of data to a GPU device.

We demonstrated an example of a GPU bug (Table 4.9) that can be found within the codebase (a.k.a intrinsic). The bug is connected to the use of multiple GPUs during training. However, due to the bug in the distribution strategy, only one GPU is active while others remain idle. All three IR-based techniques performed poorly in locating the actual buggy code for this GPU bug. BLUiR ranked the ground truth in the top 50, BugLocator at 65, and BLIA at 47. We observed that there is almost no keyword overlapping and no structural similarity between the bug report (Table 4.9) and the actual buggy code A.10. This suggests that these techniques struggled due to the lack of both textual and code-wise similarity between the bug report and source code, making it difficult to identify the buggy code for the GPU bug.

- **NDL bugs:** We found that 35.20% of bugs in deep learning applications are not directly related to deep learning, but they impact system behavior (e.g., failed CI build due to GPU compatibility issues). These bugs are known as NDL bugs in deep learning software systems. From Table 4.2 and Table 4.4, we find that NDL bugs in deep learning applications are still more difficult than traditional bugs to locate using IR-based bug localization techniques. BLUiR faces significant challenges in locating NDL bugs (MAP: 0.292 and MRR: 0.334). According to our observation, they are less complex than the deep-learning counterparts, but they are more prone to be extrinsic than the traditional bugs (48.15% from Table 4.10). Since existing baseline techniques focus on code-level artifacts only, they might fall short in detecting these bugs from deep learning systems.

Overall, there exists a significant variation in the performance of existing IR-based approaches when localizing various types of deep-learning bugs. Model and API bugs are the easiest, and GPU bugs are the most difficult to localize using IR-based techniques.

Bug report quality for bugs in deep learning software systems: Our analysis showed that bug reports from deep learning applications contain more code snippets (83.11%) than traditional software systems (33.24%). Unfortunately, that

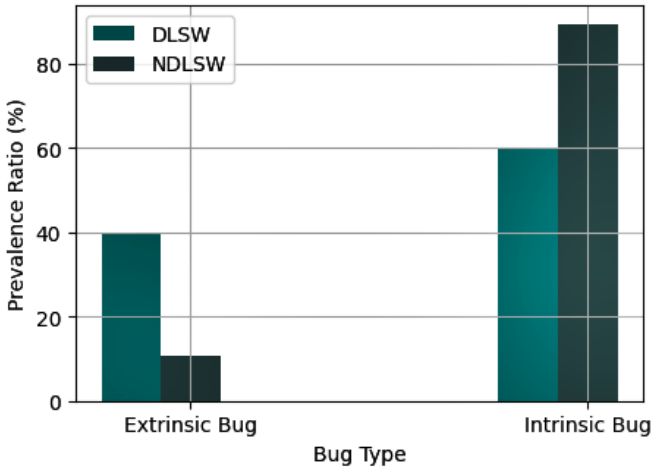


Figure 4.5: Prevalence ratio of extrinsic and intrinsic bugs in deep learning software systems (DLSW) and non-deep learning software systems (NDLSW)

does not help much in bug localization, as code snippets alone might not be sufficient. Deep learning bugs often involve intricate dependencies that extend beyond specific code components (e.g., training data bugs and GPU bugs). Complex bugs (e.g., gradient instability during training) warrant a deeper understanding of the model architecture, its dynamic behavior, and training processes, which the code snippets may not always capture.

Summary of RQ_2: We found that **64.80%** bugs in deep learning software systems are related to deep learning algorithms, whereas the remaining bugs are not related to deep learning. Our analysis shows that model bugs and API bugs are easier to localize than training and tensor bugs. However, GPU bugs are the most difficult to localize for each of the three IR-based approaches. Thus, our results not only inform the distribution of DL bugs but also highlight their challenges through extensive experiments.

4.3.3 Answering RQ_3: What are the implications of extrinsic bugs in deep learning systems for bug localization?

Most of the IR-based bug localization techniques rely on the similarity between bug reports and source code. However, if a bug is of extrinsic nature (e.g., originates from the operating system), simply relying on source code may not be effective for its localization. To investigate the impact of extrinsic bugs in deep learning software systems,

we performed a second manual analysis using the same sample datasets from RQ_2 (385 bugs from DLSW and 385 bugs from NDLSW). We manually labeled them as extrinsic and intrinsic bugs based on the heuristics of Rodriguezperez et al. [42]. Two authors of study analyze the bug reports and associated discussions carefully, consult the heuristics, and then have labeled the sample dataset separately. It achieved a Cohen's kappa [119] of 0.87, which indicates a substantial agreement between the authors. This manual analysis was documented using an Excel sheet, and each author spent a total of ≈20 hours on the analysis.

Prevalence ratio of extrinsic & intrinsic bugs: We found 40.00% extrinsic bugs within a total of 385 bug reports from deep learning software systems (Denchmark dataset). The notion of extrinsic bugs is relatively new, especially in the case of bugs from deep learning applications. For a better comparison, we also manually inspected 385 bugs from non-deep learning software systems (BugGL dataset) and determined the prevalence ratio of extrinsic and intrinsic bugs. We found only 10.65% extrinsic bugs in non-deep learning software systems. Thus, deep learning software systems contain almost four times more extrinsic bugs (Fig. 4.5).

Prevalence ratio of extrinsic & intrinsic bugs from deep learning software systems: We randomly select 100 samples for each type of bug from deep learning software systems (same as RQ_2) and determined the prevalence ratio of extrinsic and intrinsic bugs for each type. Table 4.10 shows the results of our manual analysis for different bug categories in deep learning software systems in terms of extrinsic and intrinsic bugs. In particular, from Fig. 4.6, we see that the prevalence ratio of deep learning-related, where extrinsic bugs range from 21.90% to 82.35%, whereas for non-deep learning-related bugs, the prevalence ratio is 48.15%. This suggests that the deep learning components of a software system might be more likely to trigger extrinsic bugs than non-deep learning components.

Localization of extrinsic & intrinsic bugs from both benchmarks: To further analyze the impact of extrinsic bugs on bug localization, we experimented with our baselines from RQ_1 on extrinsic and intrinsic bugs separately. We chose 100 random bugs from each category, repeated the evaluation three times using three different random subsets, and then calculated the average result for a fair comparison.

From Table 4.11, we notice that bug localization performance from all three ap-

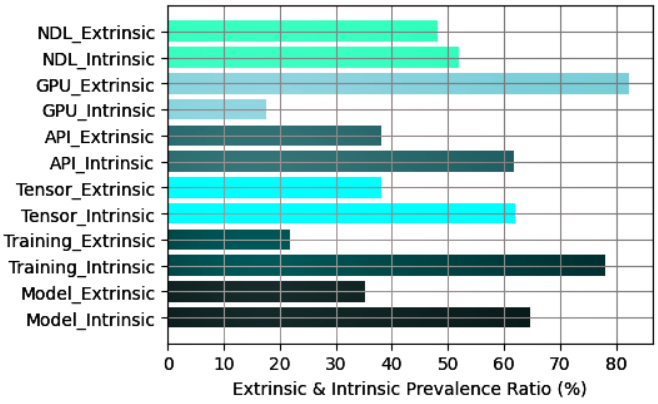


Figure 4.6: Prevalence ratio of extrinsic and intrinsic bugs for each category of bugs from deep learning software systems

Table 4.10: Prevalence ratio of extrinsic and intrinsic bugs in deep learning software systems

Type		**Extrinsic (%)**	**Intrinsic (%)**
NDL		48.15	51.85
DL	Model	35.29	64.71
	Training	21.90	78.10
	Tensor	38.10	61.90
	API	38.19	61.81
	GPU	82.35	17.65

proaches declines for the extrinsic bugs. In particular, BugLocator's performance for extrinsic bugs is lower than that when localizing intrinsic bugs, with a MAP value of 6.30%. On the other hand, BLUiR's and BLIA's performance is lower, with a MAP value of 4.30% and 7.10%, respectively. We also found that BLUiR shows less performance difference between extrinsic and intrinsic bugs. BLUiR extracts different structured items, such as API method names and API class names, from the source code. Thus, even if they reside outside of the current codebase and are invoked from an external library, they can be matched with relevant keywords from a bug report. On the other hand, BugLocator might not be able to do that due to its naive approach, i.e., considering code as regular texts.

Localization of extrinsic & intrinsic bugs from deep learning software systems: We extended our analysis of extrinsic and intrinsic bugs to both DL bugs and NDL bugs in deep learning software systems. We randomly selected 100 sample bugs from each type (DL+Extrinsic, NDL+Extrinsic, DL+Intrinsic, NDL+Intrinsic)

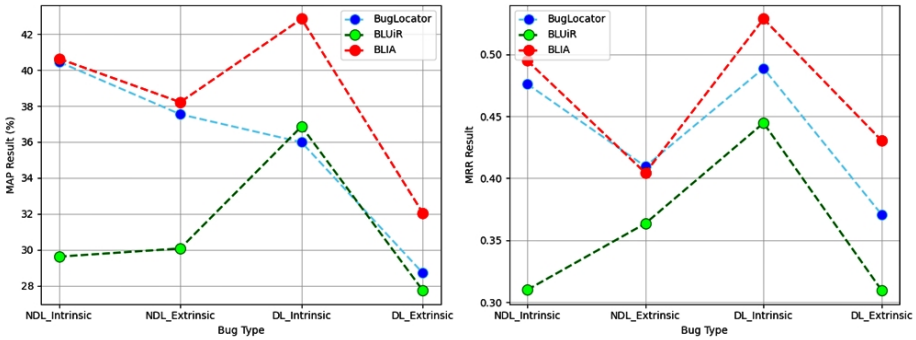


Figure 4.7: Performance of existing IR-based approaches (BugLocator, BLUiR, BLIA) for localizing extrinsic and intrinsic bugs

Table 4.11: Experimental result of existing IR-based bug localization techniques (BugLocator, BLUiR, BLIA) of extrinsic and intrinsic bug

Method	**Extrinsic**		**Intrinsic**	
	MRR	MAP	MRR	MAP
BugLocator	0.366	0.310	0.487	0.373
BLUiR	0.379	0.296	0.402	0.339
BLIA	0.424	0.346	0.497	0.417

to evaluate the performance of all three techniques in bug localization. Table 4.12 also shows that all techniques perform better for intrinsic bugs compared to extrinsic bugs for both DL and NDL bugs.

From Table 4.12, in the case of extrinsic bugs, we notice that using BugLocator the performance for DL+Extrinsic bugs is lower than for NDL+Extrinsic bugs. On the other hand, when using BLUiR, the difference in performance between the two is relatively small, with only a MAP value difference of 2.4%. However, BLIA's performance degrades with a MAP value of 6.61% for NDL+Extrinsic bugs than DL+Extrinsic bugs. Overall, these results suggest that extrinsic bugs are hard to localize whether they are related to deep learning or not. However, deep learning bugs with extrinsic nature are being the most difficult to localize.

On the other hand, from Fig. 4.7, we note that the performance of all three approaches for NDL+Intrinsic bugs is higher compared to that of DL+Intrinsic bugs, which supports the fact that bugs related to deep learning algorithms (a.k.a DL bugs) from deep learning applications are more challenging to localize.

Correlation of extrinsic & deep learning-related bugs: To determine the

Table 4.12: Experimental result of existing bug localization techniques (BugLocator, BLUiR, BLIA) of the extrinsic and intrinsic bug for deep learning related bugs and non-deep learning related bugs

Method	**DL+Extrinsic**	**DL+Intrinsic**	**NDL+Extrinsic**	**NDL+Intrinsic**
		MAP		
BugLocator	0.287	0.359	0.375	0.405
BLUiR	0.277	0.369	0.301	0.309
BLIA	0.321	0.429	0.382	0.407
		MRR		
BugLocator	0.371	0.489	0.409	0.476
BLUiR	0.309	0.445	0.296	0.364
BLIA	0.431	0.528	0.404	0.495

MAP= Mean Average Precision, **MRR**=Mean Reciprocal Ranking

potential correlation between the extrinsic bugs and the bugs in deep-learning software systems, we performed a Chi-Square test to determine any significant association [120]. We conducted three iterations with different sample data to validate the Chi-Square test, averaging the results. We got a p-value of ≈1.79e-14, which indicates a strong statistical dependence between the extrinsic bugs and the bugs in deep-learning software systems. Our manual analysis also supports the hypothesis, showing a higher prevalence of extrinsic bugs in deep learning software systems compared to traditional software systems (Fig. 4.5). The prevalence ratio of extrinsic bugs varies from 21.90% to 82.35% across different deep-learning bug types, confirming a strong correlation between extrinsic factors and bugs from deep-learning software systems. Our experiments also suggest that extrinsic bugs might have an underlying connection with deep-learning bugs, which contributes to poor performance in IR-based bug localization.

Summary of RQ_3: We found that deep learning software systems contain almost *four times* more extrinsic bugs than non-deep learning software systems. The performance of IR-based bug localization techniques for extrinsic bugs is lower (e.g., **7.18%** less MAP for BLIA) compared to that of intrinsic bugs. Our research also shows a strong connection between extrinsic bugs and bugs in deep learning applications.

4.4 Threats to Validity

We identify a few threats to the validity of our findings. In this section, we discuss these threats and the necessary steps taken to mitigate them as follows.

Threats to internal validity relate to experimental errors and human biases [100]. Traditional bug tracking systems (e.g., Bugzilla, GitHub, Jira) contain thousands of bug reports, and their quality cannot be guaranteed. This could be a source of threat as the bug reports are used as queries in IR-based approaches to locate the buggy files. Bug reports often contain poor, insufficient, missing, or even inaccurate information [23]. To address the issue, we apply standard natural language preprocessing to bug reports.

Another potential source of threat could be the replication of existing work. The original replication package was unavailable; hence we used the publicly available version of BugLocator and BLUiR [116]. For BLIA, we reused the author's replication package [60]. We validated our implementation of the existing methods using their original dataset and achieved comparable results (e.g., with differences $\approx$ 2.00%–3.00% using MAP).

Threats to conclusion validity. The observations from our study and the conclusions we drew from them could be a source of threat to conclusion validity [102]. In this research, we answer three research questions using two different datasets and re-implement three existing techniques. We use appropriate statistical tests (e.g., t-test) and report the test details (e.g., p-value, Cohen's D) to conclude. Thus, such threats might also be mitigated.

Threats to construct validity relate to the use of appropriate performance metrics. We evaluate all the methodologies using MRR, MAP, and Top@K, which have been used widely by the related work [33, 43, 44, 46, 49, 115, 121]. Thus, such threats might also be mitigated.

4.5 Related Work

4.5.1 Software bug

Understanding the nature and characteristics of bugs is essential for effective debugging and testing. They can be different across different programming languages

Table 4.13: Summary of IR-based bug localization techniques from literature review

Method	BR	SF	BRH	VCH	HH	BRS	ST	CH	MAP
BugLocator	1	1	1						0.30
BLUiR	1	1	1						0.32
Amalgam	1	1	1		1				0.35
Locus	1	1	1		1				0.36
Blizzard	1	1	1			1			0.47
BRTracer	1	1	1	1	1		1		0.33
BLIA	1	1	1	1	1		1	1	0.51

1=Present, **BR**=Bug Report, **SF**=Source Code File, **BRH**=Bug Report History, **VCH**=Version Control History, **HH**=Hunk History, **BRS**=Bug Report Structure, **ST**=Stack Trace History, **CH**=Commit History, **MAP**=Mean Average Precision

and development frameworks [122]. Over the last 50 years, hundreds of studies were conducted to tackle bugs in traditional software systems. Recently, bugs from deep learning systems have garnered much attention due to their great interest and significance. Humbatova et al. [62] proposed a taxonomy of bugs from deep learning software systems with five main categories - model, training, tensor & input, API, and GPU. Chen et al. [29] focused on the unique obstacles for deep learning-based software deployment. According to Islam et al. [123], data bugs and logic bugs are the most severe in deep-learning software systems. Another study by Islam et al. [124] showed that deep learning models' bug and repair patterns significantly differ from traditional software systems. As a result, research concentrating on deep learning-based software bug benchmarks is necessary for creating or developing automatic debugging strategies for deep learning-based software systems.

4.5.2 Information Retrieval-based bug localization

One of the crucial steps toward fixing a software bug is to detect its location within the software code. Many existing approaches [43, 44, 45, 46, 60] use IR to locate bugs by matching keywords between a query and the source code.

Zhou et al. [43] introduce BugLocator, which leverages textual similarity between bug reports and source code using rVSM for bug localization. Although it has improved the bug localization process, the performance of BugLocator is still low. Saha et al. [44] propose BLUiR, which determines the textual similarity between source

code and bug reports using the Okapi-BM25 algorithm [64]. BLUiR also leverages structural items from both bug reports and source code, which boosts localization performance. Later, Wang and Lo [46] propose AmaLgam, which incorporates the textual similarity from BugLocator, structured items from BLUiR, and version control history into IR-based bug localization.

The quality of bug reports makes traditional IR-based bug localization challenging. As a result, Rahman and Roy [45] propose BLIZZARD, which leverages the quality aspect of bug reports and introduces context-aware query reformulation into bug localization. Wong et al. [115] proposed BRTracer, which improves upon BugLocator by combining source document segmentation and stack-trace analysis.

Another technique, namely Locus [121], uses the software change information from commit logs and change histories to improve bug localization. Youm et al. [60] proposed BLIA, which integrates bug reports, structured information of source files, and source code change history. It localizes bugs in two granularity levels - file level and method level – and outperforms prior approaches.

All these IR-based approaches have been designed with a focus on traditional software bugs. Bugs in deep learning applications pose unique challenges, as follows – (a) non-deterministic behavior due to factors like random initialization and stochastic optimization [125], (b) complex relationships between high-dimensional data and model behavior and the influence of data-specific issues without direct code-level manifestations [123], (c) strong external dependencies on hardware (e.g., PyTorch leverages GPU) [32]. Although IR-based bug localization techniques have shown promising results in traditional software projects, their performance might decline while localizing bugs in deep learning applications. Our experiments also show relevant evidence to support this observation.

Recently, Kim et al. [50] used basic IR-based techniques (e.g., VSM, rVSM, BM25) for locating bugs in deep learning software systems but reported poor performance without any comprehensive analysis or explanation. Thus, the potential of existing IR-based solutions for bugs in deep-learning applications is not well understood yet. Unlike Kim et al. [50], our study extends beyond bug localization from deep learning systems. We not only assess the effectiveness of IR-based bug localization techniques but also categorize deep-learning bugs and analyze their prevalence and localization

difficulty. We also examine the strengths and weaknesses of existing IR-based bug localization techniques for each type of deep-learning bug. Furthermore, deep learning bugs often involve complex external dependencies [32] and may even manifest as extrinsic bugs that are not apparent in the source code [42]. Hence, we attempt to better understand these challenges through extensive manual analysis and deliver actionable insights.

4.5.3 Machine learning and deep learning-based bug localization

Ye et al. [126] use API descriptions and train their model parameters using previously fixed bug reports. Through a ranking model, they capture domain-dependent relationships between bug reports and source code files. Lam et al. [49] propose DNNLOC, which combines IR and deep learning for bug localization.

Xiao et al. [33] propose DeepLocator, where they use the CNN and AST to extract features from bug reports and source files, respectively. To learn unified features from natural language and source code during bug localization, Huo et al. [127] propose NP-CNN, which integrates both lexical and program structure information. Liang et al. [128] propose CAST, combining a tree-based CNN (TB-CNN) with customized AST to locate buggy files. However, these deep learning-based techniques are developed and evaluated using the source code from traditional software systems (e.g., JDT, SWT, Tomcat, AspectJ). These software systems do not represent deep learning applications, and thus designed techniques above might not be sufficient to tackle all the challenges of deep learning-related bugs.

Wardat et al. [34] propose an approach to locate Deep Neural Network (DNN) bugs through dynamic and statistical analysis. However, their method's sole focus on model and training bugs, low accuracy, and over-reliance on the Keras library pose challenges for practical implementation. Deep learning-based approaches also lack explainability and heavily rely on source code, which may not be sufficient for the bugs with external dependencies (a.k.a extrinsic bugs) in deep learning applications.

Deep learning-based techniques for bug localization from the literature (CNN, DBN, CFG, and DFG) [129] are not inherently explainable on their own [130]. LSTM has a few inherent characteristics that can aid in explanation [130]. LSTM can highlight the importance of different tokens in a sequence, which can provide a limited

explanation for their predictions. In short, the explainability of deep learning-based techniques could only be achieved with additional techniques layered on top of their models and structures [131].

Although explainable deep learning-based techniques could offer insights into the workings of their models, they might not be the best fit for addressing the specifics of our research questions in this work. Explainable deep learning techniques often provide insights into a model's decisions. However, our empirical study on bug localization involves a broader context of matching bug reports to relevant source code, which might be beyond individual model decisions. They might not be sufficient to interpret bugs that involve interactions among multiple files or components. Bug localization in deep learning systems often requires a holistic view of codebase interactions as well as external dependencies. In particular, understanding extrinsic bugs might require techniques that can capture the broader system's behaviors and can go beyond individual model's decisions. Thus, existing DL-based techniques, despite their potential for explainability, were not sufficient to answer our research questions about the effectiveness of bug localization in deep-learning software systems.

To address the above gap, in this empirical study, we replicated three existing IR-based techniques [43, 44, 60] to detect bugs in deep learning software systems. Unlike the DL-based solution above, the IR-based approaches assume a simple notion of suspiciousness, which is easy to understand. We also conduct extensive manual analysis and explain they are difficult to localize (e.g., extrinsic factors, multifaceted dependencies), which makes our work *novel*.

4.6 Summary

To summarize, identifying the location of a bug within a software system (a.k.a., bug localization) is crucial to correct any bug. In recent years, IR-based bug localization techniques have received considerable attention in the context of traditional software debugging due to their low computational cost and minimal external dependencies, but they might not be sufficient for deep learning systems. Deep learning bugs pose a greater challenge due to their multifaceted dependencies. However, the potential of IR-based approaches for localizing bugs in deep learning applications is not well

understood to date. In this work, we replicated three existing IR-based localization approaches and found that they show poor performance in localizing bugs from deep-learning applications. Secondly, through an in-depth analysis, we found that localizing certain categories of bugs (e.g., training bugs & GPU bugs) is more difficult than other bugs in deep learning systems. Finally, we investigate and find that deep learning bugs are more likely to be extrinsic, i.e., connects to non-code artifacts (e.g., training data). Our research thus offers empirical evidence and actionable insights for deep learning software bugs, advancing automated software debugging research.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Software bugs claim billions of dollars from the global economy annually. They also consume approximately 50% of developers' time. A recent survey with major tech giants (e.g., Google, Meta, Microsoft, Amazon) underscores the importance of various bug report management tasks, including duplicate bug report detection and bug localization. In this work, we aim to better understand the challenges of these two bug report management tasks. First, traditional methods for detecting duplicate bug reports mainly focus on textually similar duplicates, often overlooking the presence of textually dissimilar duplicates commonly observed in bug tracking systems. To address this gap, we collect a dataset consisting of 92,854 bug reports and construct two separate datasets comprising textually similar and textually dissimilar duplicate bug reports. We evaluate the performance of three existing techniques and answer three research questions. Our findings highlight the limitations of current approaches in detecting textually dissimilar duplicate bug reports and suggest that these reports often miss crucial information (e.g., steps to reproduce). Second, pinpointing the location of a bug within the software code poses another challenge in bug report management, where IR-based techniques have been employed. However, limited efforts have been made to detect bugs, specifically in deep learning systems. To address this gap, we collect a dataset comprising 2,365 bugs from deep learning applications and 2,913 bugs from traditional systems. We evaluate the performance of three existing IR-based techniques in localizing software bugs. Our findings reveal that IR-based methods perform poorly when applied to localizing bugs in deep learning applications. We also found that deep learning bugs are often associated with artifacts beyond the source code (e.g., GPU, training data, and external dependencies), which may explain the poor performance of the existing technique since they can analyze the source code.

5.2 Future Work

Given our conducted studies and findings, there are several potential directions for future work. We discuss them in detail as follows:

5.2.1 Duplicate bug report detection

Our work demonstrates the limitations of existing approaches in detecting textually dissimilar duplicate bug reports. We also found that these reports might miss crucial information, which leads to textual dissimilarity. We thus pose the following questions as future work.

- How can we effectively incorporate multimedia attachments (e.g., screenshots, screen recordings) into bug reports to complement their missing components and improve the overall quality of bug reports?

- Can we leverage multimedia attachments and domain-specific embeddings alongside other features (e.g., bug report metadata, code snippets, and stack traces) to detect textually dissimilar duplicate bug reports?

Our investigation highlights the significant impact of bug report quality on the effectiveness of duplicate bug report detection, even when employing neural networks. In particular, for textually dissimilar duplicates, we observed a higher occurrence of missing components (e.g., steps to reproduce and observed behaviors). In several cases, we found that bug reports in a duplicate pair mainly contain multimedia attachments like screenshots, screen recordings, and noisy elements (e.g., stack trace history) but not enough text. This situation poses challenges in finding similarities with the master report since existing approaches are designed around texts. Cooper et al. [105] leverage multimedia attachments, specifically screen recordings, for duplicate bug report detection. However, their approach assumes that both bug reports within a duplicate pair will be in video format, which we found to be impractical based on our observations. Therefore, the questions posed in this section remain open and require further investigation in the future.

5.2.2 Bug localization

Our work demonstrates the limitations of existing approaches in localizing bugs from deep-learning software systems. Our analysis reveals that different types of deep learning bugs pose distinct challenges, and we observe that these bugs tend to be predominantly influenced by external factors. We thus pose the following questions as future work.

- Is the embedding sufficient to capture the required information from bug reports? Can we design a richer embedding for DL-related bug reports?

  Word embeddings (e.g., GLoVE) offer a valuable representation of text data, including bug reports. However, more context-aware and specialized embeddings from DL bug reports can enhance the performance of duplicate bug report detection and bug localization. Designing a richer embedding for DL bug reports is possible by considering the following.

  (1) Domain-specific features: Incorporating software-specific jargon or technical terms that are prevalent in bug reports can enhance the embedding's contextual relevance [55, 78, 132, 133].
  (2) Hierarchical structures: Utilizing hierarchical attention mechanisms to give varying weights to different segments of a bug report (e.g., title, description, comments, tags) to preserve the inherent structure and relationships.
  (3) External knowledge sources: Integrating embeddings with information from external sources like API documentation, stack traces, version control history, or code repositories can enhance the understanding of bug context and potential causes.

  This enriched embedding can capture finer nuances and complexities specific to software bugs, improving model performance in tasks like duplicate bug report detection and localization.

- Can meaningful structured items be extracted from deep learning code using AST-based parsing, and how can domain-specific knowledge of deep learning models and neural network layer architectures be leveraged for this purpose?

  Future research could focus on more efficient and compact encodings for the

structural information in deep learning source code. Exploring techniques such as graph-based representations, data flow graphs (DFGs), control flow graphs (CFGs), and program dependency graphs (PDGs) could offer a nuanced understanding of the code's structure [134]. Investigating how to efficiently capture both the high-level architecture and low-level operations within these representations would improve bug localization capabilities. Moreover, tailoring AST-based parsing techniques to the intricacies of deep learning model architecture is an intriguing direction. Designing parsing methods that capture domain-specific patterns, such as layer interactions, tensor operations, and optimization procedures, would provide a more granular representation of the code. This could help identify the code segments prone to bugs and thus could lead to precise localization.

Integration of domain-specific knowledge about deep learning models in the bug localization process is a promising direction [55, 78, 132, 133]. Creating a repository of layer architectures, common model design issues, and hyperparameter sensitivities would enable bug localization methods to leverage this expertise. For instance, understanding the impact of specific hyperparameters on training stability could guide bug detection algorithms to focus on regions of code where these parameters are manipulated. As traditional IR-based techniques struggle with multifaceted bugs due to their limited scope within the codebase, another promising direction involves hybrid methods that combine structural code analysis with system-level observations [135]. By integrating code-level analysis with system-level monitoring, these hybrid techniques could uncover intricate dependencies between code artifacts and external factors, providing a comprehensive perspective on bug localization for categories such as data bugs and GPU-related issues.

Finally, future research could develop bug-specific strategies from the existing taxonomy of deep-learning bugs [62]. For instance, for bugs associated with external dependencies, techniques could leverage dependency analysis and version tracking to identify discrepancies. Likewise, data lineage analysis could be employed for data-related bugs [136].

- How can we automatically identify the extrinsic bugs in deep learning applications based on the heuristics of Rodriguezperez et al. [42]?
- How can we automatically classify different types of deep learning bugs (such as model bugs, training bugs, and GPU bugs)? Can we determine the most appropriate bug localization technique for each specific bug category?
- What strategies can be devised to effectively handle the challenges posed by external factors (e.g., GPU, training data) in deep learning bugs?

We found that bugs from deep learning applications often exhibit extrinsic characteristics with multifaceted dependencies such as external frameworks (e.g., third-party libraries), external environments (e.g., OS, GPU), and non-code artifacts (e.g., training data). Traditional bug localization techniques, including IR-based ones, might not be equipped well to tackle these multifaceted dependencies, as they are not typically found within the software codebase. Hence, identifying extrinsic bugs is crucial in determining whether IR-based bug localization techniques can be effectively employed. We also found that IR-based techniques are not suitable for several categories of deep learning bugs from the taxonomy, such as bugs with multifaceted external dependencies and extrinsic nature (e.g., data bugs, GPU bugs). Interestingly, IR-based bug localization techniques were effective in localizing several categories of deep learning bugs, such as model bugs and tensor bugs. Model bugs are connected to a model's type, properties, and layers, which might be mentioned in a bug report. Similarly, training and tensor bugs are structured items from code, such as incorrect tensor shapes or incorrect loss functions, which could be useful for structured IR-based bug localization techniques. Thus, based on our findings and existing literature on bug localization for deep learning applications, the presented questions above remain open and warrant further investigation.

Bibliography

[1] A. Arcuri. On the automation of fixing software bugs. In *ICSE*, pages 1003–1006, 2008.

[2] R. M. Karampatsis and C. Sutton. How often do single-statement bugs occur? the manysstubs4j dataset. In *MSR*, pages 573–577, 2020.

[3] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu. How practitioners perceive automated bug report management techniques. *IEEE TSE*, 46(8): 836–862, 2020.

[4] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proc. ICSE*, pages 361–370, 2006.

[5] N. Shrikanth, S. Majumder, and T. Menzies. Early life cycle software defect prediction. why? how? In *ICSE*, pages 448–459. IEEE, 2021.

[6] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *Proc. DSN*, pages 52–61, 2008.

[7] M. M. Rahman, F. Khomh, and M. Castelluccio. Why are some bugs non-reproducible?:–an empirical investigation using data fusion–. In *ICSME*, pages 605–616. IEEE, 2020.

[8] J. Kanwal and O. Maqbool. Bug prioritization to facilitate bug report triage. *Journal of Computer Science and Technology*, 27:397–412, 2012.

[9] I. M. Rodrigues, D. Aloise, E. R. Fernandes, and M. Dagenais. A soft alignment model for bug deduplication. In *Proc. MSR*, pages 43–53, 2020.

[10] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *ICSE*, pages 14–24, 2012.

[11] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proc. ICSE*, pages 499–510, 2007.

[12] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proc. ICSE*, pages 461–470, 2008.

[13] A. Sureka and P. Jalote. Detecting duplicate bug report using character n-gram-based features. In *APSEC*, pages 366–374, 2010.

[14] C. Z. Yang, H. H. Du, S. S. Wu, and X. Chen. Duplication detection for software bug reports based on bm25 term weighting. In *Proc. TAAI*, pages 33–38, 2012.

[15] K. Aggarwal, F. Timbers, T. Rutgers, A. Hindle, E. Stroulia, and R. Greiner. Detecting duplicate bug reports with software engineering domain knowledge. *Journal of Software: Evolution and Process*, 29(3):e1821, 2017.

[16] A. Alipour, A. Hindle, and E. Stroulia. A contextual approach towards more accurate duplicate bug report detection. In *MSR*, pages 183–192, 2013.

[17] R. P. Gopalan and A. Krishna. Duplicate bug report detection using clustering. In *ASWEC*, pages 104–109, 2014.

[18] Y. Tian, C. Sun, and D. Lo. Improved duplicate bug report identification. In *Proc. CSMR*, pages 385–390, 2012.

[19] C. Sun, D. Lo, X. Wang, J. Jiang, and S. C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proc. ICSE*, volume 1, pages 45–54, 2010.

[20] N. Klein, C. S. Corley, and N. A. Kraft. New features for duplicate bug detection. In *Proc. MSR*, page 324–327, 2014.

[21] J. He, L. Xu, X. Yan, M. an Xia, and Y. Lei. Duplicate bug report detection using dual-channel convolutional neural networks. In *Proc. ICPC*, pages 117–127, 2020.

[22] A. Budhiraja, K. Dutta, R. Reddy, and M. Shrivastava. Dwen: deep word embedding network for duplicate bug report detection in software repositories. In *Proc. ICSE*, pages 193–194, 2018.

[23] S. Gupta and S. K. Gupta. A systematic study of duplicate bug report detection. *International Journal of Advanced Computer Science and Applications*, 12(1), 2021.

[24] I. Chawla and S. K. Singh. Performance evaluation of vsm and lsi models to determine bug reports similarity. In *Proc. IC3*, pages 375–380, 2013.

[25] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30(11):964–971, 1987.

[26] S. Y. Ho, K. Phua, L. Wong, and W. W. B. Goh. Extensions of the external validation for checking learned model interpretability and generalizability. *Patterns*, 1(8):100129, 2020.

[27] S. A Akbar and A. C Kak. A large-scale comparative evaluation of ir-based tools for bug localization. In *MSR*, pages 21–31, 2020.

[28] D. Binkley and D. Lawrie. Information retrieval applications in software maintenance and evolution. *Encyclopedia of software engineering*, pages 454–463, 2010.

[29] Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu. A comprehensive study on challenges in deploying deep learning-based software. In *ESEC/FSE*, pages 750–762, 2020.

[30] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann. Software engineering for machine learning: A case study. In *ICSE-SEIP*, pages 291–300, 2019.

[31] D. Gonzalez, T. Zimmermann, and N. Nagappan. The state of the ml-universe: 10 years of artificial intelligence & machine learning software development on github. In *MSR*, pages 431–442, 2020.

[32] L. Nganyewou Tidjon, B. Rombaut, F. Khomh, and A. E. Hassan. An empirical study of library usage and dependency in deep learning frameworks. *arXiv e-prints*, pages arXiv–2211, 2022.

[33] Y. Xiao, J. Keung, Q. Mi, and K. E. Bennin. Improving bug localization with an enhanced convolutional neural network. In *APSEC*, pages 338–347, 2017.

[34] M. Wardat, W. Le, and H. Rajan. Deeplocalize: Fault localization for deep neural networks. In *ICSE*, pages 251–262, 2021.

[35] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proc. ASE*, pages 70–79, 2012.

[36] B. S. Neysiani and S. M. Babamir. Duplicate detection models for bug reports of software triage systems: A survey. *Current Trends In Computer Sciences and Applications*, 1(5):128–134, 2019.

[37] O. Obulesu, M. Mahendra, and M. ThrilokReddy. Machine learning techniques and tools: A survey. In *ICIRCA*, pages 605–611, 2018.

[38] J. S. Almeida. Predictive non-linear modeling of complex data by artificial neural networks. *Current opinion in biotechnology*, 13(1):72–76, 2002.

[39] Z. Bitvai and T. Cohn. Non-linear text regression with a deep convolutional neural network. In *Proc. ACL*, pages 180–185, 2015.

[40] L. Deng and Y. (Eds.) Liu. *Deep learning in natural language processing*. Springer, 2018.

[41] L. Feng, L. Song, C. Sha, and X. Gong. Practical duplicate bug reports detection in a large web-based development community. In *APWEB*, pages 709–720, 2013.

[42] G. Rodriguezperez, M. Nagappan, and G. Robles. Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the openstack project. *IEEE TSE*, 2020.

[43] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *ICSE*, pages 14–24, 2012.

[44] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *ASE*, pages 345–355, 2013.

[45] M. M. Rahman and C. K. Roy. Improving ir-based bug localization with context-aware query reformulation. In *ESEC/FSE*, pages 621–632, 2018.

[46] S. Wang and D. Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *ICPC*, pages 53–63, 2014.

[47] L. Moreno, J. Treadway, J, A. Marcus, and W. Shen. On the use of stack traces to improve text retrieval-based bug localization. In *ICSME*, pages 151–160, 2014.

[48] A. Perez, R. Abreu, and A. Riboira. A dynamic code coverage approach to maximize fault localization efficiency. *Journal of Systems and Software*, 90: 18–28, 2014.

[49] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *ICPC*, pages 218–229, 2017.

[50] M. Kim, Y. Kim, and E. Lee. An empirical study of ir-based bug localization for deep learning-based software. In *ICST*, pages 128–139, 2022.

[51] T. Akilan, D. Shah, N. Patel, and R. Mehta. Fast detection of duplicate bug reports using lda-based topic modeling and classification. In *Proc. SMC*, pages 1622–1629, 2020.

[52] J. Deshmukh, K. M. Annervaz, S. Podder, S. Sengupta, and N. Dubash. Towards accurate duplicate bug retrieval using deep learning techniques. In *Proc. ICSME*, pages 115–124, 2017.

[53] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[54] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Proc. EMNLP*, pages 1532–1543, 2014.

[55] T. L. Chen, M. Emerling, G. R. Chaudhari, Y. R. Chillakuru, Y. Seo, T. H. Vu, and J. H. Sohn. Domain specific word embeddings for natural language processing in radiology. *Journal of biomedical informatics*, 113:103665, 2021.

[56] A. Roy, Y. Park, and S. Pan. Learning domain-specific word embeddings from sparse cybersecurity texts. *arXiv:1709.07470*, 2017.

[57] F. Nooralahzadeh, L. Øvrelid, and J. T. Lønning. Evaluation of domain-specific word embeddings using knowledge resources. In *Proc. LREC)*, 2018.

[58] S. Muvva, A. E. Rao, and S. Chimalakonda. Bugl–a cross-language dataset for bug localization. *arXiv preprint arXiv:2004.08846*, 2020.

[59] M. Kim, Y. Kim, and E. Lee. Denchmark: A bug benchmark of deep learning-related software. In *MSR*, pages 540–544, 2021.

[60] K. C. Youm, J. Ahn, and E. Lee. Improved bug localization based on code change histories and bug reports. *Information and Software Technology*, 82: 177–192, 2017.

[61] M. M. Rahman, F. Khomh, S. Yeasmin, and C. K. Roy. The forgotten role of search queries in ir-based bug localization: an empirical study. *EMSE*, 26(6): 116, 2021.

[62] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella. Taxonomy of real faults in deep learning systems. In *ICSE*, pages 1110–1121, 2020.

[63] C. D Manning, P Raghavan, and H Sch"utze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[64] S. Robertson and H. Zaragoza. *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc, 2009.

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

[66] H. M. Wallach. Topic modeling: beyond bag-of-words. In *Proc. ICML*, pages 977–984, 2006.

[67] H Yu and J Yang. A direct lda algorithm for high-dimensional data—with application to face recognition. *Pattern recognition*, 34(10):2067–2070, 2001.

[68] S. Syed and M. Spruit. Full-text or abstract? examining topic coherence scores using latent dirichlet allocation. In *DSAA*, pages 165–174, 2017.

[69] J Inglesfield. A method of embedding. *Journal of Physics C: Solid State Physics*, 14(26):3795, 1981.

[70] O. Levy and Y. Goldberg. Neural word embedding as implicit matrix factorization. *Advances in neural information processing systems*, 27, 2014.

[71] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *EMNLP*, pages 1532–1543, 2014.

[72] O. I Abiodun, A Jantan, A. E Omolara, K. V Dada, N. A Mohamed, and H Arshad. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11):e00938, 2018.

[73] J Gu, Z Wang, J Kuen, L Ma, A Shahroudy, B Shuai, T Liu, X Wang, G Wang, J Cai, et al. Recent advances in convolutional neural networks. *Pattern recognition*, 77:354–377, 2018.

[74] R. R Varior, M Haloi, and G Wang. Gated siamese convolutional neural network architecture for human re-identification. In *ECCV*, pages 791–808, 2016.

[75] S. Robertson and H. Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *FnTs in Information Retrieval*, 3(4):333–389, 2009.

[76] O. Chaparro, J. M. Florez, U. Singh, and A. Marcus. Reformulating queries for duplicate bug report detection. In *SANER*, pages 218–229, 2019.

[77] R. F. Woolson. Wilcoxon signed-rank test. *Wiley encyclopedia of clinical trials*, pages 1–3, 2007.

[78] V. Efstathiou, C. Chatzilenas, and D. Spinellis. Word embeddings for the software engineering domain. In *Proc. MSR*, pages 38–41, 2018.

[79] B. S. Li, T. Estlander, R. Jolanki, and H. I. Maibach. Advantages and disadvantages of gloves. *Kanerva's Occupational Dermatology*, pages 2547–2561, 2020.

[80] S. M. Rezaeinia, R. Rahmani, A. Ghodsi, and H. Veisi. Sentiment analysis based on improved pre-trained word embeddings. *Expert Systems with Applications*, 117:139–147, 2019.

[81] B. W. Yap, K. A. Rani, H. A. A. Rahman, S. Fong, Z. Khairudin, and N. N. Abdullah. An application of oversampling, undersampling, bagging and boosting in handling imbalanced datasets. In *Proc. DaEng*, pages 13–22, 2014.

[82] I. Žliobaitė. Learning under concept drift: an overview. *arXiv:1010.4784*, 2010.

[83] D. Buscaldi, R. Tournier, N. Aussenac-Gilles, and J. Mothe. Irit: Textual similarity combining conceptual similarity with an n-gram comparison method. In *SemEval*, pages 552–556, 2012.

[84] F. Peng, D. Schuurmans, V. Keselj, and S. Wang. Language independent authorship attribution with character level n-grams. In *EACL*, 2003.

[85] K. Bansal and H. Rohil. Literature review of finding duplicate bugs in open source systems. In *CCICT*, pages 389–396, 2021.

[86] A. Hindle and C. Onuczko. Preventing duplicate bug reports by continuously querying bug reports. *EMSE*, 24(2):902–936, 2019.

[87] S. Gupta and V. Varma. Scientific article recommendation by using distributed representations of text and graph. In *Proc. WWW*, pages 1267–1268, 2017.

[88] A. Lazar, S. Ritchey, and B. Sharif. Improving the accuracy of duplicate bug report detection using textual similarity measures. In *Proc. MSR*, 2014.

[89] M. Hossin and M. N. Sulaiman. A review on evaluation metrics for data classification evaluations. *IJDKP*, 5:01–11, 03 2015.

[90] J. Lafferty and D. Blei. Correlated topic models. *Advances in neural information processing systems*, 18:147, 2006.

[91] S. Lai, L. Xu, K. Liu, and J. Zhao. Recurrent convolutional neural networks for text classification. In *Proc. AAAI*, 2015.

[92] P. Royston. Approximating the shapiro-wilk w-test for non-normality. *Statistics and computing*, 2(3):117–119, 1992.

[93] T. K. Kim. T test as a parametric statistic. *Korean journal of anesthesiology*, 68(6):540, 2015.

[94] R. Rosenthal, H. Cooper, and L. Hedges. Parametric measures of effect size. *The handbook of research synthesis*, 621(2):231–244, 1994.

[95] R. A. Groeneveld and G. Meeden. Measuring skewness and kurtosis. *Journal of the Royal Statistical Society Series D (The Statistician)*, 33(4):391–399, 1984.

[96] L. T. DeCarlo. On the meaning and use of kurtosis. *Psychological methods*, 2 (3):292, 1997.

[97] M. Kusner, Y. Sun, N. Kolkin, and K. Weinberger. From word embeddings to document distances. In *ICML*, pages 957–966, 2015.

[98] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proc. SIGSOFT/FSE*, pages 308–318, 2008.

[99] A. Alwehaibi and K. Roy. Comparison of pre-trained word vectors for arabic text classification using deep learning approach. In *ICMLA*, pages 1471–1474, 2018.

[100] Y. Tian, D. Lo, and J. Lawall. Automated construction of a software-specific word similarity database. In *Proc. CSMR-WCRE*, pages 44–53, 2014.

[101] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 1972.

[102] M. A. García-Pérez. Statistical conclusion validity: Some common threats and simple remedies. *Frontiers in psychology*, 3:325, 2012.

[103] L. Wu, S. C. Hoi, and N. Yu. Semantics-preserving bag-of-words models and applications. *IEEE TIP*, 19(7):1908–1920, 2010.

[104] A. Aizawa. An information-theoretic perspective of tf–idf measures. *Information Processing and Management*, 39(1):45–65, 2003.

[105] N. Cooper, C. Bernal-Cárdenas, O. Chaparro, K. Moran, and D. Poshyvanyk. It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports. In *ICSE*, pages 957–969. IEEE, 2021.

[106] R. Reddy Budhiraja and M. Shrivastava. Lwe: Lda refined word embeddings for duplicate bug report detection. In *Proc. ICSE*, pages 165–166, 2018.

[107] A. H. Beg and M. Z. Islam. Advantages and limitations of genetic algorithms for clustering records. In *ICIEA*, pages 2478–2483, 2016.

[108] T. M. Rocha and A. L. D. C. Carvalho. Siameseqat:a semantic context-based duplicate bug report detection using replicated cluster information. *IEEE Access*, 9:44610–44630, 2021.

[109] Q. Xie, Z. Wen, J. Zhu, C. Gao, and Z. Zheng. Detecting duplicate bug reports with convolutional neural networks. In *Proc. APSEC*, pages 416–425, 2018.

[110] M. M. Rahman and C. K. Roy. Improving ir-based bug localization with context-aware query reformulation. In *ESEC/FSE*, pages 621–632, 2018.

[111] T. K. Kim. T test as a parametric statistic. *Korean Journal of Anesthesiology*, 68(6):540, 2015.

[112] M. E. Rice and G. T. Harris. Comparing effect sizes in follow-up studies: Roc area, cohen's d, and r. *Law and human behavior*, 29:615–620, 2005.

[113] P. A. Hernandez, C. H. Graham, L. L. Master, and D. L. Albert. The effect of sample size and species characteristics on performance of different species distribution modeling methods. *Ecography*, 29(5):773–785, 2006.

[114] A. S. Acharya, A. Prakash, P. Saxena, and A. Nigam. Sampling: Why and how of it. *Indian Journal of Medical Specialties*, 4(2):330–333, 2013.

[115] C. P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *ICSME*, pages 181–190, 2014.

[116] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. Le Traon. Bench4bl: reproducibility study on the performance of ir-based bug localization. In *ISSTA*, pages 61–72, 2018.

[117] R. Rosenthal, H. Cooper, and L. Hedges. Parametric measures of effect size. *The handbook of research synthesis*, 621(2):231–244, 1994.

[118] H. Aoyama. A study of stratified random sampling. *Ann. Inst. Stat. Math*, 6 (1):1–36, 1954.

[119] P. F. Watson and A Petrie. Method agreement analysis: a review of correct methodology. *Theriogenology*, 73(9):1167–1179, 2010.

[120] M. L. McHugh. The chi-square test of independence. *Biochemia medica*, 23(2): 143–149, 2013.

[121] M. Wen, R. Wu, and S. C. Cheung. Locus: Locating bugs from software changes. In *ASE*, pages 262–273, 2016.

[122] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, et al. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *ESEC/FSE*, pages 1556–1560, 2020.

[123] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan. A comprehensive study on deep learning bug characteristics. In *ESEC/FSE*, pages 510–520, 2019.

[124] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan. Repairing deep neural networks: Fix patterns and challenges. In *ICSE*, pages 1135–1146, 2020.

[125] M. Sajjadi, M. Javanmardi, and T. Tasdizen. Regularization with stochastic transformations and perturbations for deep semi-supervised learning. *Advances in neural information processing systems*, 29, 2016.

[126] X. Ye, R. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *FSE*, pages 689–699, 2014.

[127] X. Huo, M. Li, and Z. H. Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *IJCAI*, volume 16, pages 1606–1612, 2016.

[128] H. Liang, L. Sun, M. Wang, and Y. Yang. Deep learning with customized abstract syntax tree for bug localization. *IEEE Access*, 7:116309–116320, 2019.

[129] Yunhua Zhao, Kostadin Damevski, and Hui Chen. A systematic survey of just-in-time software defect prediction. *ACM Computing Surveys*, 55(10):1–35, 2023.

[130] W. Samek, T. Wiegand, and K.R. Müller. Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. *arXiv preprint arXiv:1708.08296*, 2017.

[131] G. Ras, N. Xie, Van G. M., and D. Doran. Explainable deep learning: A field guide for the uninitiated. *Journal of Artificial Intelligence Research*, 73: 329–396, 2022.

[132] K. Aggarwal, F. Timbers, T. Rutgers, A. Hindle, E. Stroulia, and R. Greiner. Detecting duplicate bug reports with software engineering domain knowledge. *JSEP*, 29(3):e1821, 2017.

[133] F. Nooralahzadeh, L. Øvrelid, and J. T. Lønning. Evaluation of domain-specific word embeddings using knowledge resources. In *Proc. LREC*, 2018.

[134] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather. Programl: Graph-based deep learning for program optimization and analysis. *arXiv preprint arXiv:2003.10536*, 2020.

[135] I. Stefanakos, S. Gerasimou, and R. Calinescu. Software performance engineering with performance antipatterns and code-level probabilistic analysis. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 249–253, 2021.

[136] A. Jurčo. Data lineage analysis for pyspark and python orm libraries. 2023.

Appendix A

Supplementary details

A.1 Replication Package

A.1.1 Duplicate bug report detection

Github repository: https://github.com/SigmaJahan/Towards-Understanding-the-Impacts-of-Textual-Dissimilarity-on-Duplicate-Bug-Report-Detection

A.1.2 Bug localization

Github repository: https://github.com/SigmaJahan/Bug-Localization-for-Deep-Learning-Software-Bugs

A.1.3 Source Code and Bug Report

- Details of Model bug —
  1. Ground truth file: https://bit.ly/3XJCdqq
  2. Incorrect retrieved file by BugLocator: https://bit.ly/43klnj9
  3. Incorrect retrieved file by BLUiR: https://bit.ly/3PQo88U
  3. Bug report: https://github.com/asyml/texar-pytorch/issues/313

- Details of Training bug —
  1. Ground truth file: https://bit.ly/3NKfRAH
  2. Incorrect retrieved file: https://bit.ly/3O5yEIa
  3. Bug report: https://github.com/fastai/fastai/issues/3048

- Details of Tensor bug —
  1. Ground truth file: https://bit.ly/3XLe82y
  2. Incorrect retrieved file: https://bit.ly/3Dr5LA9
  3. Bug report: https://github.com/apache/mxnet/issues/13760

- Details of API bug —
  1. Ground truth file: https://bit.ly/3JT1JE8
  2. Incorrect retrieved file: https://bit.ly/43oMl9r
  3. Bug report: https://github.com/apache/mxnet/issues/13862
- Details of GPU bug —
  1. Ground truth file: https://bit.ly/44llZXc
  2. Bug report: https://github.com/keras-team/autokeras/issues/1238

```
class GPT2Tokenizer(TokenizerBase, PretrainedGPT2Mixin):
    r"""Pre-trained GPT2 Tokenizer.
    Args:
        pretrained_model_name (optional): a `str`, the name of
            pre-trained model (e.g., `117M`). Please refer to
            :class:`~texar.torch.modules.PretrainedGPT2Mixin` for
            all supported models.
            If None, the model name in :attr:`hparams` is used.
        cache_dir (optional): the path to a folder in which the
            pre-trained models will be cached. If `None` (default),
            a default directory (``texar_data`` folder under user's home
            directory) will be used.
        hparams (dict or HParams, optional): Hyperparameters. Missing
            hyperparameter will be set to default values. See
            :meth:`default_hparams` for the hyperparameter structure
            and default values.
```

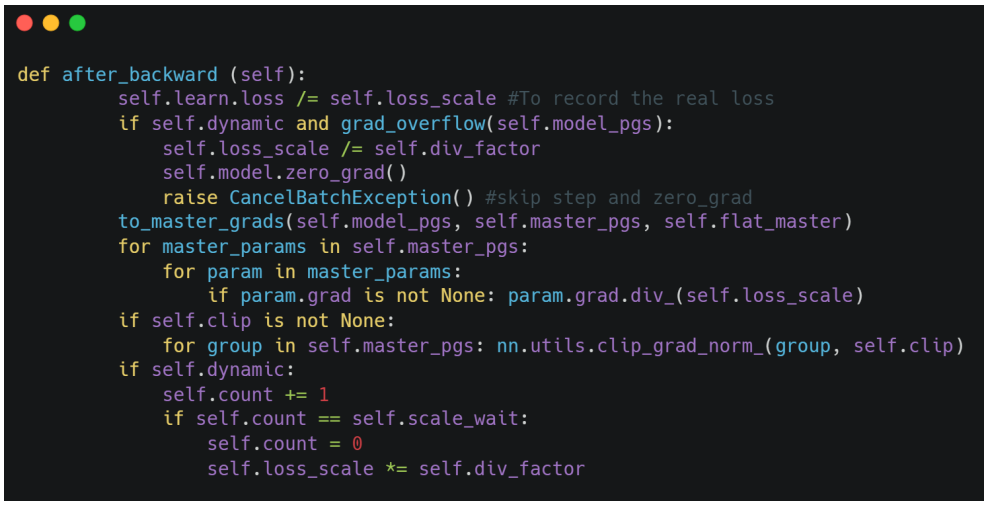
Figure A.1: Code snippet of the ground truth for the model bug

```
class SentencePieceTokenizerTest(unittest.TestCase):

    def setUp(self):
        self.tmp_dir = tempfile.TemporaryDirectory()
        self.SAMPLE_VOCAB = maybe_download(
            'https://github.com/google/sentencepiece/blob/master/'
            'python/test/test_model.model?raw=true', self.tmp_dir.name)

        self.tokenizer = SentencePieceTokenizer.load(self.SAMPLE_VOCAB)

        self.tokenizer.save(self.tmp_dir.name)

    def tearDown(self):
        self.tmp_dir.cleanup()

    def test_load(self):
        tokenizer = SentencePieceTokenizer.load(self.tmp_dir.name)
```

Figure A.2: Code snippet of the incorrect source file for the model bug retrieved by BugLocator

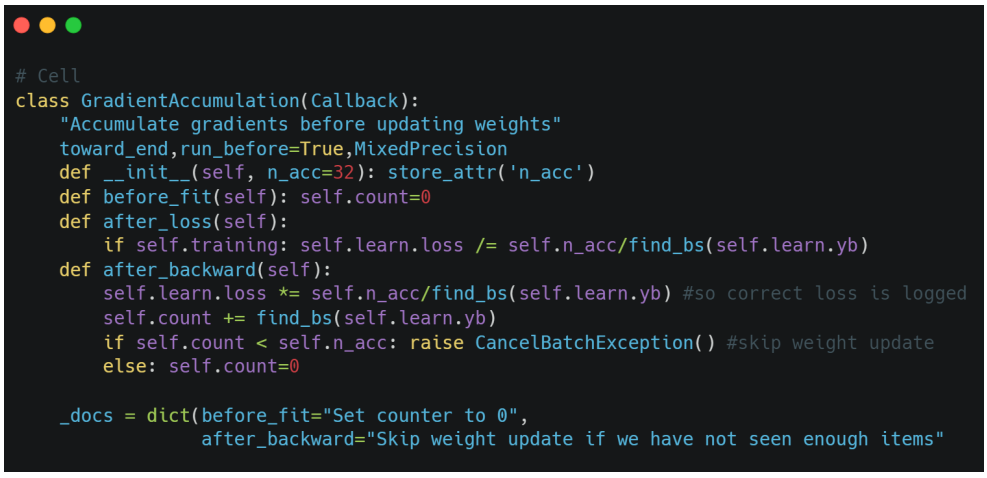
```
class Seq2Seq(nn.Module):

    def __init__(self, train_data):
        super().__init__()

        self.source_vocab_size = train_data.source_vocab.size
        self.target_vocab_size = train_data.target_vocab.size

        self.bos_token_id = train_data.target_vocab.bos_token_id
        self.eos_token_id = train_data.target_vocab.eos_token_id

        self.source_embedder = tx.modules.WordEmbedder(
            vocab_size=self.source_vocab_size,
            hparams=config_model.embedder)

        self.target_embedder = tx.modules.WordEmbedder(
            vocab_size=self.target_vocab_size,
            hparams=config_model.embedder)

        self.encoder = tx.modules.BidirectionalRNNEncoder(
            input_size=self.source_embedder.dim,
            hparams=config_model.encoder)

        self.decoder = tx.modules.AttentionRNNDecoder(
            token_embedder=self.target_embedder,
            encoder_output_size=(self.encoder.cell_fw.hidden_size
+
self.encoder.cell_bw.hidden_size),
            input_size=self.target_embedder.dim,
            vocab_size=self.target_vocab_size,
            hparams=config_model.decoder)
```

Figure A.3: Code snippet of the incorrect source file for the model bug retrieved by BLUiR



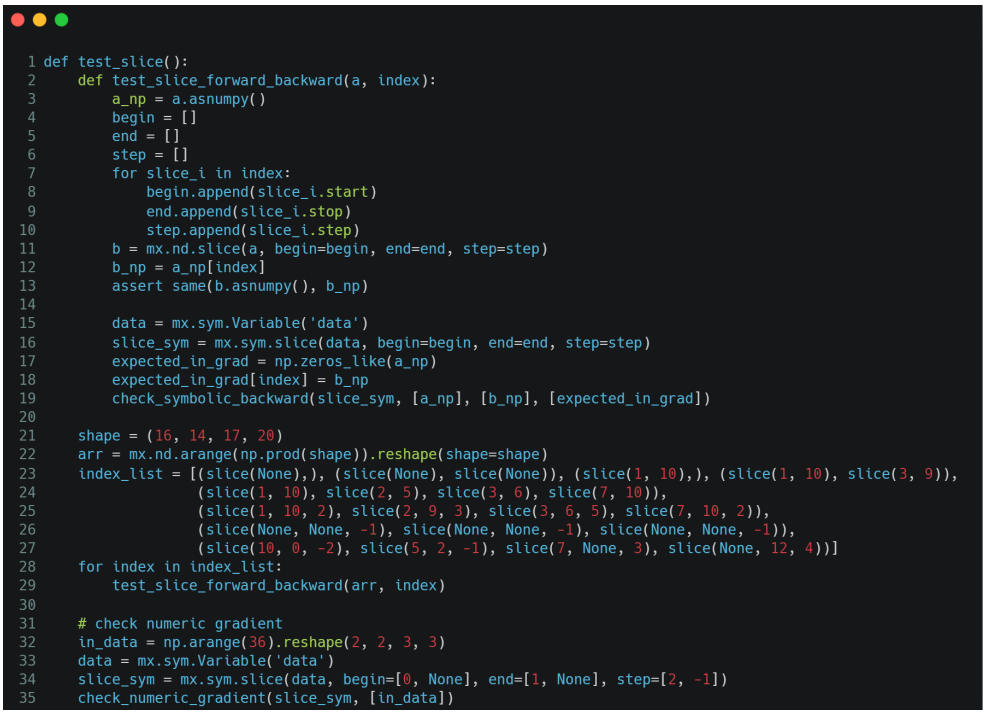
```
def after_backward (self):
        self.learn.loss /= self.loss_scale #To record the real loss
        if self.dynamic and grad_overflow(self.model_pgs):
            self.loss_scale /= self.div_factor
            self.model.zero_grad()
            raise CancelBatchException() #skip step and zero_grad
        to_master_grads(self.model_pgs, self.master_pgs, self.flat_master)
        for master_params in self.master_pgs:
            for param in master_params:
                if param.grad is not None: param.grad.div_(self.loss_scale)
        if self.clip is not None:
            for group in self.master_pgs: nn.utils.clip_grad_norm_(group, self.clip)
        if self.dynamic:
            self.count += 1
            if self.count == self.scale_wait:
                self.count = 0
                self.loss_scale *= self.div_factor
```

Figure A.4: Code snippet of the ground truth for training bug



```
# Cell
class GradientAccumulation(Callback):
    "Accumulate gradients before updating weights"
    toward_end,run_before=True,MixedPrecision
    def __init__(self, n_acc=32): store_attr('n_acc')
    def before_fit(self): self.count=0
    def after_loss(self):
        if self.training: self.learn.loss /= self.n_acc/find_bs(self.learn.yb)
    def after_backward(self):
        self.learn.loss *= self.n_acc/find_bs(self.learn.yb) #so correct loss is logged
        self.count += find_bs(self.learn.yb)
        if self.count < self.n_acc: raise CancelBatchException() #skip weight update
        else: self.count=0

    _docs = dict(before_fit="Set counter to 0",
                 after_backward="Skip weight update if we have not seen enough items"
```

Figure A.5: Code snippet of the incorrect source file for the training bug retrieved by BugLocator

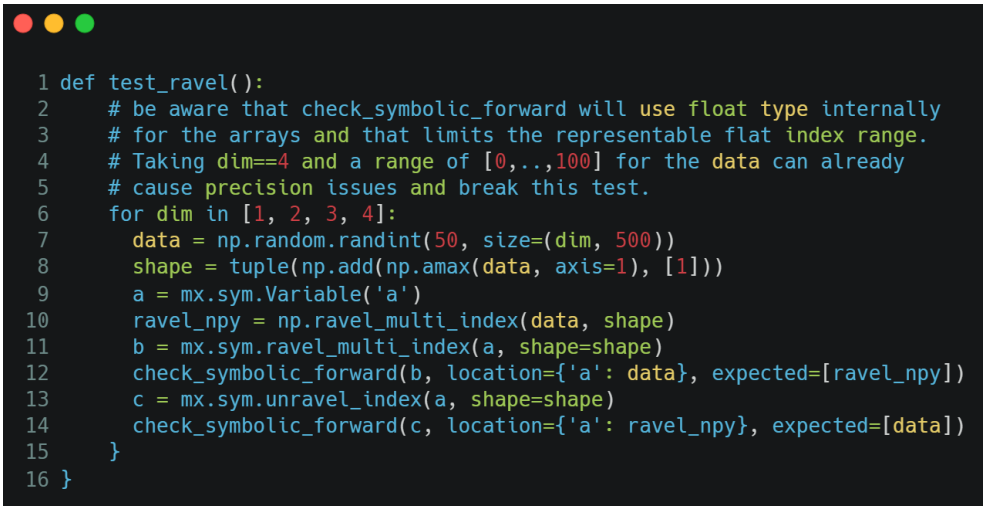


```python
def test_slice():
    def test_slice_forward_backward(a, index):
        a_np = a.asnumpy()
        begin = []
        end = []
        step = []
        for slice_i in index:
            begin.append(slice_i.start)
            end.append(slice_i.stop)
            step.append(slice_i.step)
        b = mx.nd.slice(a, begin=begin, end=end, step=step)
        b_np = a_np[index]
        assert same(b.asnumpy(), b_np)

        data = mx.sym.Variable('data')
        slice_sym = mx.sym.slice(data, begin=begin, end=end, step=step)
        expected_in_grad = np.zeros_like(a_np)
        expected_in_grad[index] = b_np
        check_symbolic_backward(slice_sym, [a_np], [b_np], [expected_in_grad])

    shape = (16, 14, 17, 20)
    arr = mx.nd.arange(np.prod(shape)).reshape(shape=shape)
    index_list = [(slice(None),), (slice(None), slice(None)), (slice(1, 10),), (slice(1, 10), slice(3, 9)),
                  (slice(1, 10), slice(2, 5), slice(3, 6), slice(7, 10)),
                  (slice(1, 10, 2), slice(2, 9, 3), slice(3, 6, 5), slice(7, 10, 2)),
                  (slice(None, None, -1), slice(None, None, -1), slice(None, None, -1)),
                  (slice(10, 0, -2), slice(5, 2, -1), slice(7, None, 3), slice(None, 12, 4))]
    for index in index_list:
        test_slice_forward_backward(arr, index)

    # check numeric gradient
    in_data = np.arange(36).reshape(2, 2, 3, 3)
    data = mx.sym.Variable('data')
    slice_sym = mx.sym.slice(data, begin=[0, None], end=[1, None], step=[2, -1])
    check_numeric_gradient(slice_sym, [in_data])
```

Figure A.6: Code snippet of the ground truth for the tensor bug

```python
def parse_args():
    """Parse arguments."""
    parser = argparse.ArgumentParser(
        formatter_class=argparse.ArgumentDefaultsHelpFormatter,
        description='Diagnose script for checking the current system.')
    choices = ['python', 'pip', 'mxnet', 'os', 'hardware', 'network']
    for choice in choices:
        parser.add_argument('--' + choice, default=1, type=int,
                            help='Diagnose {}.'.format(choice))
    parser.add_argument('--region', default='', type=str,
                        help="Additional sites in which region(s) to test. \
                        Specify 'cn' for example to test mirror sites in China.")
    parser.add_argument('--timeout', default=10, type=int,
                        help="Connection test timeout threshold, 0 to disable.")
    args = parser.parse_args()
    return args

URLS = {
    'MXNet': 'https://github.com/apache/incubator-mxnet',
    'Gluon Tutorial(en)': 'http://gluon.mxnet.io',
    'Gluon Tutorial(cn)': 'https://zh.gluon.ai',
    'FashionMNIST': 'https://apache-mxnet.s3-accelerate.dualstack.amazonaws.com/gluon/dataset/fashion-mnist/train-labels-
idx1-ubyte.gz',
    'PYPI': 'https://pypi.python.org/pypi/pip',
    'Conda': 'https://repo.continuum.io/pkgs/free/',
}
REGIONAL_URLS = {
    'cn': {
        'PYPI(douban)': 'https://pypi.douban.com/',
        'Conda(tsinghua)': 'https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/',
    }
}
```

Figure A.7: Code snippet of the incorrect source file for the tensor bug retrieved by BugLocator



```
def test_ravel():
    # be aware that check_symbolic_forward will use float type internally
    # for the arrays and that limits the representable flat index range.
    # Taking dim==4 and a range of [0,..,100] for the data can already
    # cause precision issues and break this test.
    for dim in [1, 2, 3, 4]:
      data = np.random.randint(50, size=(dim, 500))
      shape = tuple(np.add(np.amax(data, axis=1), [1]))
      a = mx.sym.Variable('a')
      ravel_npy = np.ravel_multi_index(data, shape)
      b = mx.sym.ravel_multi_index(a, shape=shape)
      check_symbolic_forward(b, location={'a': data}, expected=[ravel_npy])
      c = mx.sym.unravel_index(a, shape=shape)
      check_symbolic_forward(c, location={'a': ravel_npy}, expected=[data])
    }
}
```

Figure A.8: Code snippet of the ground truth for the API bug

```
class NDArray(NDArrayBase):
    """An array object representing a multidimensional, homogeneous array of
fixed-size items.
    """
    __slots__ = []
    # make numpy functions return NDArray instead of numpy object array
    __array_priority__ = 1000.0
    # Extension type code for TVM function.
    # See C++ side of definition(kTVMNDArrayTypeCode) at include/mxmet/tensor_blob.
    _tvm_tcode = 19
    # pylint: disable= no-member, undefined-variable

    @property
    def _tvm_handle(self):
        return self.handle.value

    def __repr__(self):
        """Returns a string representation of the array."""
        shape_info = 'x'.join(['%d' % x for x in self.shape])
        return '\n%s\n<%s %s @%s>' % (str(self.asnumpy()),
                                      self.__class__.__name__,
                                      shape_info, self.context)

    def __reduce__(self):
        return NDArray, (None,), self.__getstate__()
```

Figure A.9: Code snippet of the incorrect source file for the API bug retrieved by BLUiR

```python
def build(self, hp):
      """Build the HyperModel into a Keras Model."""
      tf.keras.backend.clear_session()
      self._register_hps(hp)
      self.compile()
      real_nodes = {}
      for input_node in self.inputs:
          node_id = self._node_to_id[input_node]
          real_nodes[node_id] = input_node.build()
      for block in self.blocks:
          temp_inputs = [real_nodes[self._node_to_id[input_node]]
                         for input_node in block.inputs]
          outputs = block.build(hp, inputs=temp_inputs)
          outputs = nest.flatten(outputs)
          for output_node, real_output_node in zip(block.outputs, outputs):
              real_nodes[self._node_to_id[output_node]] = real_output_node
      model = tf.keras.Model(
          [real_nodes[self._node_to_id[input_node]] for input_node in
           self.inputs],
          [real_nodes[self._node_to_id[output_node]] for output_node in
           self.outputs])

      return self._compile_keras_model(hp, model)
```

Figure A.10: Code snippet of the GPU bug