C02029: Doctor of Philosophy

33874: PhD Thesis: Software Engineering

Sep 2022

Multi-triage: A Multi-Task Learning Approach to Bug Triaging

Thazin Win Win Aung

School of Computer Science

Faculty of Eng. & IT

University of Technology Sydney

NSW - 2022, Australia

Multi-triage : A Multi-Task Learning approach to Bug Triaging

A thesis submitted in fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Software Engineering

by

Thazin Win Win Aung

to

School of Computer Science
Faculty of Engineering and Information Technology
University of Technology Sydney
NSW, Australia

Sep 2022

Certificate of Original Authorship

I, *Thazin Win Win Aung*, declare that this thesis is submitted in fulfilment of the requirements for the award of the award of Doctor of Philosophy, in the *Faculty of Engineering and IT* at the University of Technology Sydney, Australia.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.

This research is supported by the Australian Government Research Training Program.

SIGNATURE: ______________________________

[Thazin Win Win Aung]

DATE: 30th Sep, 2022

PLACE: Sydney, Australia

Dedication

To my family who brings out the best in me.
To my supervisors for their inspiration and endless support.
To my late partner, my best friend, and my little buddy Albert, for their unconditional love and encouragement.

Abstract

Bug triage plays a significant role in software maintenance activities, including optimization, error correction, and feature enhancement. Triage is the procedure of assigning the severity, issue type, and developer to resolve the issue in the most effective order. Performing triage is time-consuming and challenging, depending on the system's complexity. Thus, it is time-consuming and hinders the effectiveness of linkages between two triage tasks. An automated approach to assisting the issue allocation process to relevant category and developer benefits bug triages. A large body of previous work aims to address the allocation problem by conjecturing the extensive list of approaches ranging from the heuristics-based approach, text retrieval approach, and machine learning approach. However, these studies treated the issue of categorization and assignment tasks as a single task learning model and developed a multiple recommendation system.

This dissertation aims at leveraging the bug triage process by adopting the multi-task learning approach. We developed a multi-triage model, a system for predicting developers and issue kinds for a brand-new issue report. In our approach, we split issue reports, the text description, and code snippets into two separate tokens to conjecture the contributions of each context in the learning model. Furthermore, we addressed the class-unbalanced challenge of data sets by generating synthetic issue reports using the contextual data-augmentation approach.

We conducted four studies in this thesis. The first was an empirical study of the automatic traceability link recovery approach to analyze how previous studies addressed the linkages between software artifacts (e.g., requirements, issue reports, test cases, and source code). The second was the experimental studies about visualizing the linkages of software artifacts using the hierarchical trace map. The first two studies were mainly focused on understanding the broad concepts of how software artifacts can be linked together and presented to stakeholders effectively. Based on this accumulated knowledge, we designed the multi-triage model to identify the linkages between developers, issue types, and issue reports to leverage the bug triage process. Lastly, we conducted a case study of the issue tracking system used by the software consulting company to conjure the process of introducing the automatic developer assignment and labeling recommendation model in the bug triage process.

Our study led to several key findings. We found that the multi-triage model training time and performance are better than single-task learning models. We also uncovered that including the contextual data augmentation-based synthetic bug reports in training data sets can improve the learning model's performance noticeably. Lastly, we presented the ability to learn issue descriptions and code snippets in two separate tokens and their effect on the learning model.

Acknowledgments

First and foremost, thank you to my principal supervisor, Professor Yulei Sui, for his mentorship, guidance, and support throughout this long Ph.D. journey and his positive attitude toward life and study encouragement when it was most needed.

Thank you to my kind advisor, Dr. Huan (Angela) Huo and Dr. Yao Wan, for their invaluable lessons, guidance, and encouragement during my studies. Also, I am sincerely grateful to Dr. Christy Jie Liang, who directed me to the brighter side of the research world while I lost my way.

I am extremely grateful to them for their unconditional support, advice, and patience over the years. They opened my eyes to what research in software engineering is all about and taught me how exciting and challenging academia truly is. This achievement would not have been possible without their incredible feedback and mentorship.

I must thank the Australian Postgraduate Research Intern (APR. Intern) industrial internship grant for providing me with funding for the research work. I am also thankful to Dialog IT company for allowing me to conduct a field study in real-world software projects for my thesis writing. I would also like to thank all the software consultants involved in the field study for this research project. The field study could not have been successfully conducted without their participation and input.

Special thanks are due to my beloved late partner Kyi Thar Hain for his continuous support and understanding until his last moment. My thanks are extended to my family members and my friends for their constant source of inspiration. Last but not least, I would like to thank my little energizer Albert (chocolate labrador).

PUBLICATIONS

RELATED TO THE THESIS :

1. T. W. W. AUNG, H. HUO, AND Y. SUI, Interactive traceability links visualization using hierarchical trace map, in 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2019, pp. 367-369.

2. T. W. W. AUNG, H. HUO, AND Y. SUI, A Literature Review of Automatic Traceability Link Recovery for Software Change Impact Analysis, Association for Computing Machinery, New York, NY,USA, 2020, p. 14-24.

3. T. W. W. AUNG, Y. WAN, H. HUO, AND Y. SUI, Multi-triage: A multi-task learning framework for bug triage, Journal of Systems and Software, 184 (2022),p. 111133.

Table of Contents

List of Figures

LIST OF TABLES

Chapter 1

Introduction

During the software development and maintenance stages, managing the software bug is the primary factor affecting the cost and time of software delivery. Software projects ranging from open-source and commercial projects to in-house organizational projects need an issue tracking system to elicit feature requests or defects that need to be tracked. Despite the variety among the issue tracking systems, the core of the portal is to facilitate the primary need of the end user and project team by documenting the bug details, grouping the bug to the appropriate system area, assigning it to the appropriate developer, and subsequently keeping track of the progress until the bug has been triaged. This process is known as bug triage.

Depending on the project settings, either a group or an individual can execute the triaging. Bug triggers are geographically dispersed in open source projects since it is anticipated that developers will independently go through the list of reported bugs and assign them to the appropriate ones. In open-source project communities, it is uncommon to find that contributor triggers are often performing as the front-line bug triages and go through thousands of newly opened and re-opened bug reports to categorize and assign them to the relevant developer. Bug triage days are periodically planned in the Mozilla [115] and open-stack [41] projects to allow contributors to add further comments to an issue report that are specific enough for developers to correct the bug. However, in a commercial area, the projects are usually maintained by a dedicated project team as triaging is commonly performed in a regular group meeting. Bug reports are handled following the team members' agreements, which include discussions on who should work on them and which category they should be assigned to.

1.1 Challenges

The initial assessment of a bug report using individual knowledge rather than group knowledge has a high chance of assessing the root cause of the bug inaccurately and assigning it to the wrong developer. As a consequence, the work needs to reassign to another developer. The process of software application triaging can be a complex activity that requires knowledge of the development process as well as the rules and norms of the project structure. However, with the distributed culture of the open source community and the dynamic nature of software development, it is impossible to expect front-liners triggers to be familiar with all these rules and norms of the project structure. A structural project like Open Stack provides the following bug triage guidelines for contributor triages [41]:

> "Check if a bug was filed for a correct component (project). If not, either change the project or mark it as invalid. For example, if the bug impacts the project-specific dev-ref, then mark it as invalid. If a bug is reported against the nova installation guide, ensure Open Stack-manuals are removed, and the nova project is added. Check if a similar bug was filed before. You may also check already verified bugs to see if the bug has been reported. If so, mark it as a duplicate of the previous bug."

This guideline provides heuristics to apply when triaging a bug. Tagging a bug as invalid, grouping it to the correct project component, and verifying duplicates are key pieces of instructions that a triage follows to work through the list of bug reports.

Similarly, the Mozilla project steps of triaging guild recommend the following [115]:

> "We suggest checking that you are confirming a bug on a relevant platform - at the beginning of the bug report there is a Platform field that should list the Operating System the bug was initially found on (Please click here to see an example). If there is no specified platform, no worries! just go on with the next steps. Read the description and comments in the bug to understand the problem. If you clearly understand the bug, move it out of the Untriaged component into something more appropriate. If you don't understand the bug

or can't reproduce it, add a comment to the report asking follow-up questions. If you can reproduce the bug, mark it as NEW."

This triage guideline is another example of how various factors are considered when processing bug reports. OpenStack and Mozilla triage guidelines focused on categorizing bug reports to appropriate project components as the first step by providing project component information to contributors for reference. Although the references are helpful for contributors, it is overwhelming to go through over a dozen project components.

In general, the content of the bug report includes two types of information; 1) text description and 2) code snippets. The text description, expressed in natural language, includes unexpected behaviours of the system observed by the user. In some cases, the user provides the steps to reproduce the error in a code snippet expressed in a programming language. A bug triage reviews this baseline information to perform issue types classification and developer assignment activities. Thus, it is tedious and resources intensive, especially in large-scale software development projects. Recently, studies in automated traceability link recovery approaches have received broad attention in the bug triage community to overcome the challenges of the manual bug triage process.

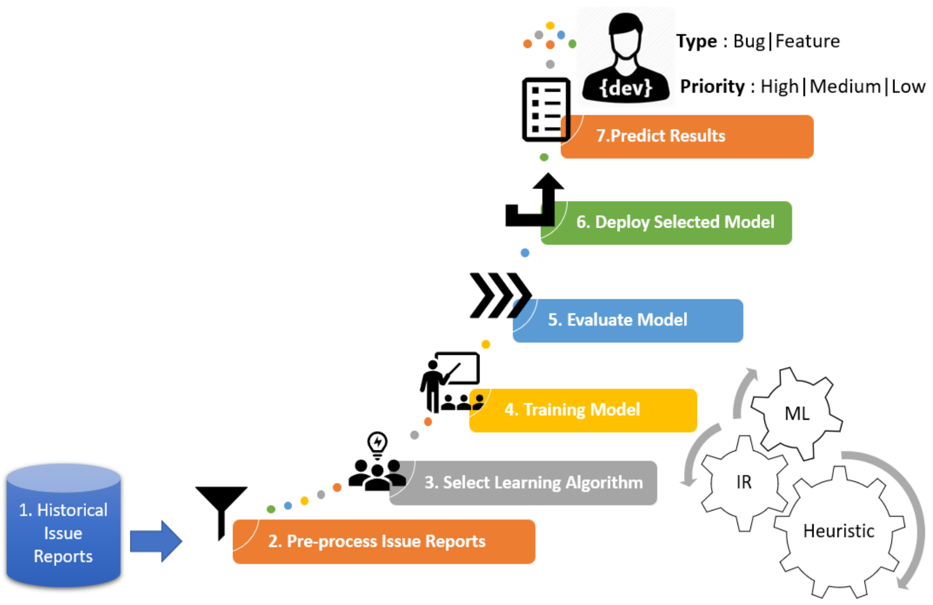


Figure 1.1: Bug Triage Recommendation Model

To improve bug triage performance, a large body of work sought to create issue types and developers' recommendation models to assist issue trackers in the classification and assignment process. Figure 1.1 presents the fundamental process of building the bug triage recommendation model. The first step is the data collection phase, where historical bug reports are collected from the issue tracking system (e.g., GitHub, Bugzilla). Next, bug report descriptions are normalized using text processing techniques (e.g., stop word removal, stemming, and lemmatization). In previous studies, researchers filtered outstack traces, code snippets, hyperlinks, and special symbols from the description to reduce the noise in learning the representation process. Most studies set the minimum threshold to eliminate bug reports with minority categorical values (e.g., developers who fix a few bug reports). After the filtering process, the labelling process is performed on each bug report. For example, a bug report is marked as a bug or feature and labelled with the developer's name. The quality of datasets affects the overall performance of the recommendation model. After pre-processing, researchers applied various natural language processing ranging from the heuristics (rule-based) approach, information retrieval (e.g., text similarity, semantic similarity) approach, to the machine learning (text classification) approach to training the model. The learning algorithm selection, training, and evaluating phases are turned incrementally until the model is ready for deployment. Although these approaches reduce the frequency of assigning the bug to an incorrect category and developer, the performance of the bug triage recommendation model mainly relied on the quality and availability of historical bug reports. Additionally, low-quality bug descriptions cause noise in training the recommendation model. The performance of the model is impacted by a shortage and an imbalance in past bug reports. Last but not least, the existing methods fall short in handling numerous jobs, like issue categories and assimilating developers for a new bug report.

All in all, this thesis intends to address four main challenges. Firstly, there are no prior studies of a Multitask learning (MTL) approach in the context of the bug triage process. However, it has been used in solving complex problems in various areas, including computer vision, natural language processing, and natural language processing understanding, for several years. Interestingly, there are no recent review studies on insights into the automation approaches conjured in various software maintenance activities, including software traceability, bug triage, concept location, impact analysis, program comprehension, and verifying test coverage. Thus, it is important to perform a review study to build up field knowledge to leverage the bug triage automation process.

Second, previous studies are trained to solve one task, such as developers and issue types, although both recommendation models relied on the historical bug reports [69, 153]. The MTL approach can optimize a single model of multiple related tasks by exploiting useful information between learning tasks to optimize decision boundaries on the original task. Therefore, it hinders the parallel distributed learning process. Furthermore, prior studies only use the contextual information from bug reports to train the model; it cannot utilize the information provided in structured language (e.g., code snippet). Therefore, it is essential to leverage the bug recommendation model to maximize its capability to its full potential.

Third, prior approaches present the top selection lists in textual labels (e.g., potential developers, relevant issue types) as it needs to be more informative for the project team to select the most relevant developers or issue types in a timely manner [78, 153]. To select the most relevant one from the list, the project teams require additional information, such as the list of similar bug reports fixed by these developers. As the system evolved over time, it was challenging to read a large volume of historical bug reports to validate the results. In recent years, software traceability studies have tackled the high volume data challenges by embedding visualization techniques in presenting the relationships between various software artifacts (e.g., requirements, design, and code) to facilitate software maintenance and evolution [14, 80]. A similar approach can leverage the bug triage resource assignment automation process.

Lastly, the majority of studies explore the effect of bug triage recommendation model in open-source applications as it may not cover the industrial software standards and requirements. Performing case studies on industrial projects is important for bug triage researchers to understand the complexity and diversity of bug triage system.

1.2 Our Approaches

In this thesis, we proposed a multitask recommendation model called multi-triage, which can predict relevant developers and issue types (e.g., bug or enhancement) for a new bug report. We leverage the recommendation model to perform multiple tasks that will assist triages in working with an extensive list of bugs. Compared to the traditional single-task learning model, our model training time is faster while maintaining the same level of precision. In our approach, we treat a bug report in two forms: 1) contextual and 2) structural to extract the representation of the bug report precisely by including the code snippets information. In addition, we introduced a synthetic bug report generator to overcome the imbalanced dataset challenge. We applied the contextual data augmentation approach to existing bug reports to generate synthetic ones with similar contexts for novel words. Initially, we implemented the model by turning it into a collection of bug reports from open-source projects. We implemented an automated triage model for industrial projects and learned the feasibility of working with bug reports and the impact on triaging work.

In summary, we conducted a four-part study in this work. The first two studies were an initial exploratory to broaden our understanding of designing our multi-triage model. First, we have conducted a systematic literature review (SLR) related to automatic traceability links recovery approaches to identify the remaining challenges in state-of-the-art systems. Thus, it contributes a body of knowledge to traceability research and bug triage communities. Second, we conjured a hierarchical trace map visualization technique to interactively present relationships among software artifacts. Our hierarchical trace map visualization technique provides functionalities to explore interrelationships between various artifacts at once naturally and intuitively. Our presenting methodology can assist the project teams to understand systems and issue triage to perform immediate system verification. In our third study, we investigated the impacts of a synthetic issue report and a multitask learning model based on the collected knowledge. Our multi-triage model resolves both tasks simultaneously via MTL approach. A critical aspect of our approach is that it transforms the code snippet into an abstract syntax tree (AST) token and uses it as part of the input tokens. Therefore, our approach provides a precise bug report feature representation. Furthermore, we applied a contextual data augmentation approach to generate synthetic bug reports for over-sampled imbalanced datasets, thereby increasing model accuracy. Lastly, we performed a bug triage process model case study on real-world industrial projects and implemented the artificial intelligence (AI) bug triage process model based on the requirements of industrial projects.

1.3 Thesis Organization

Figure 1.2 presents the flow chart of the thesis chapters.The rest of this dissertation is organized as follows:

Chapter 2 describes our work of literature review in automatic traceability link recovery published in [15]. This chapter takes a broad look at the various types of automatic traceability recovery approaches proposed by previous studies published during 2014–2018. We provide insights into traceability approaches, traces artifacts, changes in impact analysis coverage, trace links granularity, and recovery of traceability links between two software artifacts.

Chapter 3 describes our work of hierarchical trace map visualization technique published in [14]. The chapter gives the preliminaries of a hierarchical trace map and a motivating example, then demonstrates interrelationships between various software

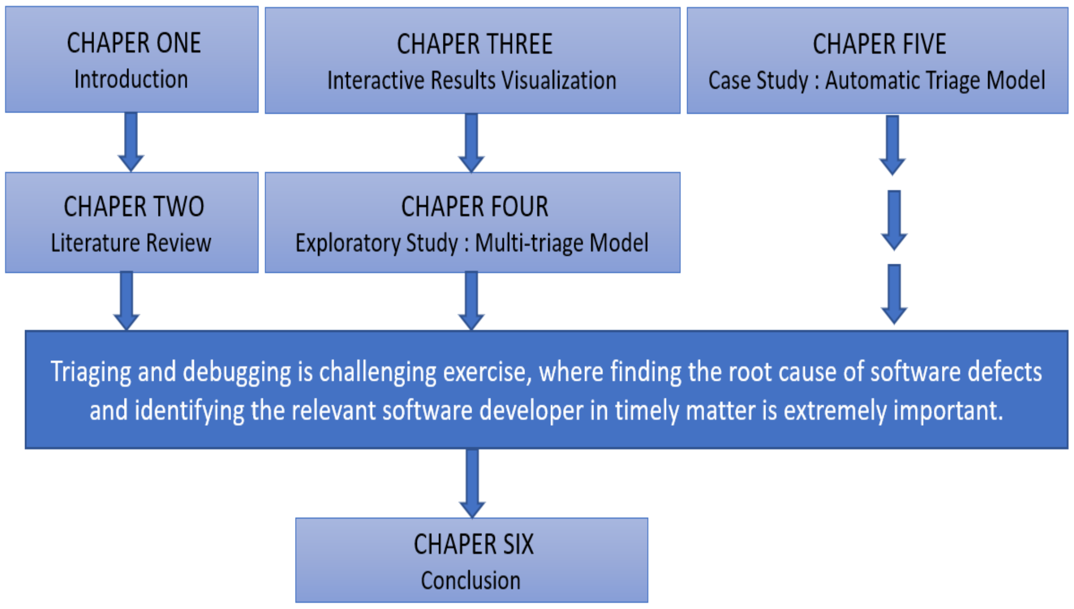

CHAPER ONE
Introduction
CHAPER THREE
Interactive Results Visualization
CHAPER FIVE
Case Study : Automatic Triage Model
CHAPER TWO
Literature Review
CHAPER FOUR
Exploratory Study : Multi-triage Model
Triaging and debugging is challenging exercise, where finding the root cause of software defects and identifying the relevant software developer in timely matter is extremely important.
CHAPER SIX
Conclusion

Figure 1.2: Thesis Flow Chart

artifacts using a trace map.

Chapter 4 describes our work of a multi-triage model that simultaneously predicts developers and issue types for a new bug report via the multitask learning (MTL) approach published in [16]. This chapter explains the overall design, then presents the motivating example, the algorithm to generate synthetic bug reports and over-sampled the minority group, and finally analyses the experimental results.

Chapter 5 describes the automatic bug triage model evaluated in real-world industrial projects. This chapter presents an in-depth study of the legacy bug triage model of one of the leading Australian IT consulting companies. We designed a centralized AI-based automatic bug triage process model during the study and evaluated the new model with the company's historical bug reports. Our model's outcomes open up a new way of thinking about bug triage design and materials for the company.

Chapter 6 presents our concluding remarks, summarizes the contributions of this dissertation, and finally sheds some light on the direction of future work.

CHAPTER 2

A LITERATURE REVIEW OF AUTOMATIC TRACEABILITY LINK RECOVERY

2.1 Overview

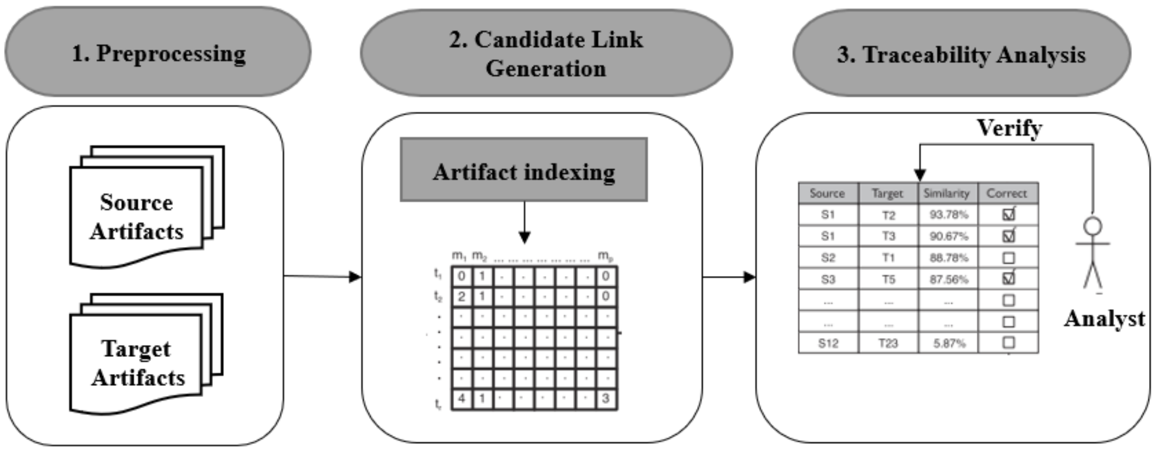


Figure 2.1: High-level overview of IR-based Traceability Recovery Process

In large-scale software development projects, Change Impact Analysis (CIA) plays an important role in controlling the software design evolution. CIA is the process of identifying the impact areas of a change in a software system to estimate the time or efforts to make a change [27]. During the software development cycle, it is common practice to use traceability relationships across different software artifacts to identify and access the impacts of software changes. Recently, research in automated traceability link recovery has received broad attention in the software maintenance community, aiming to reduce the cost of manual maintenance of trace links by developers. Bug triage is one of the crucial software maintenance activities. The aim of this thesis is to leverage the automation of the bug triage process, as it is important to understand the broader concepts of automation within the software maintenance field. The bug triage recommendation model is based on the idea of recovering the links between historical bug reports and a new bug report to predict the potential developers and relevant issue types.

In this chapter, we have presented a Systematic Literature Review (SLR) related to automatic Traceability Link Recovery (TLR) approaches with a focus on the CIA. We identified 32 relevant studies and investigated deep insights into the following aspects of CIA: traceability approaches, trace artifacts, CIA coverage, trace links granularity, and ways of recovering traceability links between two artifacts. The results of our SLR indicate that there are rooms for further research to leverage the traceability process to support end-to-end CIA in continuous software development.

Over the past years, several empirical and experiential traceability studies highlighted how trace links are used to accelerate project management processes and improve overall software quality. Requirements traceability is defined as "the ability to describe and follow the life of a requirement, in both a forward and backward direction (i.e., from its origins, through its development and specification, to its subsequent development and use, and all periods of ongoing refinement and iteration in any of these phrases" [62, p. 94]. To ensure that stakeholder requirements are met properly, traceability connects requirements with subsequent development artifacts. For quality control and product certification in critical safety systems, building traceability relationships between software artifacts is essential [38]. However, manually setting up and maintaining trace links is time-consuming and prone to errors.

Among some earlier attempts at automating the traceability recovery process, the IR-based trace recovery technique is commonly used in this context. The IR-based approach retrieves the trace links based on the textual similarity between the two artifacts. The fundamental concept of this approach is the recognition of the semantic similarity between artifacts, assuming two artifacts having high textual similarity share similar concepts [7, 8, 58]. During software development, artifacts produced throughout the projects are mostly written in natural languages, such as requirements, user stories, test cases, bug reports, and source code comments.

Figure 2.1 presents the high-level structure of the IR-based traceability recovery process. Generally, IR-based tracing consists of three steps 1) pre-processing, candidate link generation, and traceability analysis [70]. For example, consider the recovery trace link between requirements and source code. In pre-processing parsing phase, the required artifacts are first parsed to extract search keywords using standard pre-processing operations (e.g. Stop words removal, stemming, part of speech tagging, and identifier splitting, lowercase, tokenization) [19, 58, 90, 100]. Second, IR-based term-matching techniques such as Vector Space Model(VSM), Latent Semantic Indexing, Jensen-Shannon Divergence (JD), Latent Dirichlet Allocation (LDA), and BM2 are used to generate lists of possible candidate links. In the traceability analysis phase, analysts verify these trace results manually and make the judgment. Thus, the final trace analysis phase of IR-based techniques requires human efforts, as it is error-prone and resource-intensive.

In [8], the study exploits what factors influence the creation of traceability links in source code artifacts. The experiments involve monitoring developer's eye movements to analyze which source code entities most developers pay attention to in verifying trace links. Their findings highlight that class names, method names, variable names, and comments are the key entities developers use in creating trace links manually. Following their experimental inspiration, similar kinds of hypothesis research are performed to identify the effects of using co-changes files information in the trace recovery process. Co-changed files are “two or more files frequently co-changed together (in source code repository) for a longer period” [7, p. 361]. Theoretically, these co-changed files have a conceptual linkage between them and could be related to the same requirements'specifications. The positive outcomes of this research validated the practicality of this assumption. Similarly, in [100], the study conjured the effects of refactoring the code systematically on trace recovery process performance. In their experiments, source code is refactored in three techniques namely; restore information, move information, and remove the information and observed the effects of each technique. Out of three techniques, restoring information provides benefits to the trace recovery process, enhancing the source code quality by updating outdated terms and taxonomies. However, removing information has a negative impact on the trace generation process as well as moving information does not provide noticeable benefits. Likewise, in [118], the study presented the requirements and source code changes monitoring mechanism to remind developers to refactor the code to maintain trace links effectively. Apart from constructing the links between different artifacts, the study in [59] presented the sorting algorithm to generate trace links between source code and source code comments.

Our key interest is how previous studies assist the CIA process in identifying the relationships between software artifacts. We are keen on exploiting the underlying challenges in constructing the trace links between artifacts and how studies addressed problems in past years. Thus, this empirical study can be a starting point to identify the remaining works left for future research to leverage the traceability links recovery process.

Our interests led us to conduct a systematic review of peer-reviewed studies related to automatic Traceability Link Recovery(TLR) approaches focusing on the CIA. We followed the systematic literature review guidelines provided by Kitchenham and Charters [84]. We reviewed the primary studies published during 2012–2019, focusing on how these primary studies assist the CIA process by measuring the level of support based on trace artifacts, approaches, the granularity of trace links, and common ways to recover trace links between artifacts. Furthermore, we presented our findings to cover standard software artifacts produced throughout the software development. This body of knowledge led to the design of our hierarchical trace map visualization technique and multi-triage approach presented in chapters 3 and 4.

2.2 Background

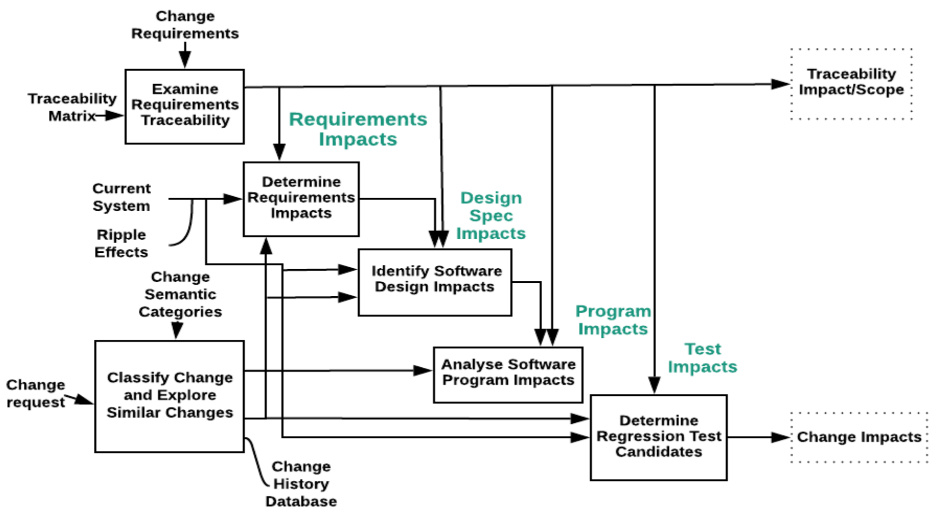

Change Requirements
Traceability Matrix
Examine Requirements Traceability
Requirements Impacts
Traceability Impact/Scope
Current System
Ripple Effects
Determine Requirements Impacts
Design Spec Impacts
Identify Software Design Impacts
Change Semantic Categories
Program Impacts
Test Impacts
Change request
Classify Change and Explore Similar Changes
Analyse Software Program Impacts
Determine Regression Test Candidates
Change Impacts
Change History Database

Figure 2.2: Identify Software Change Impacts [27]

Figure 2.2 presents Bohner's CIA process model. The process model represents the key activities in identifying software impacts. The process starts with *examining requirements traceability* activity. This activity is based on the assumption that traceability information is previously defined and maintained during development. The output of this activity is a set of current impacts. The traceability activity is broken down into five sub-activities as follows.

- **Classify change and explore similar changes** - This activity assesses the change history data to compare the current change request with similar software changes from the system's change history database.
- **Determine requirements impacts** - This activity determines the requirements impacts by associating new requirements with current system requirements history information passed from *classifying change and exploring similar changes* activity.
- **Identify software design impacts** - This activity examines the current system architecture and program design information passed from *classifying change and exploring similar changes* and identifies design impacts based on the information guided by *examining requirements traceability* and *determining requirements impacts* activities.
- **Analyze software program impacts** - This activity uses various program analysis techniques such as program slicing, data flow analysis, control flow analysis, and dependency analysis to determine program impact sets.
- **Determine regression test candidates** - To determine test implications, this activity uses information from the classified change and explores similar changes activity and how the program impacts activity and requirements traceability activity.

In summary, there are a total of four change impact sets needing examination to generate the complete scope of changes. Therefore, recovering traceability links between software artifacts is essential for a CIA process to determine the scope of changes efficiently.

As mentioned in the previous section, recovering and maintaining traceability information throughout software development is important for the CIA process. It is necessary to build traceability relationships between heterogeneous artifacts according to Bohner's impact analysis model (e.g., requirements, design, source code, and test case) [27]. However, one of the main challenges of recovering traceability links between software artifacts of different types is a knowledge gap problem. Knowledge in this context refers to the syntax and semantics of the artifacts [45]. The knowledge gap between software documentation and source code is very large. The formal one is usually expressed in a natural language, whereas the latter follows program syntax and language. Recovering knowledge-based traceability links between these heterogeneous artifacts required data normalization as well as human expert verification [20, 57, 89].

In the last decades, information retrieval (IR) has been widely adopted as a core technology to address the problem of recovering knowledge-based traceability links between artifacts of different types [19, 20, 57, 89, 122]. This approach established traceability relationships based on the assumption that if two artifacts have high textual similarity between them, there is a high chance they are related to one another. However, an IR-based approach requires human experts to verify the candidate trace links manually, as it is error-prone and time-consuming.

Recently, several studies have proposed approaches to training machine learning (ML) classification models to verify the validity of trace links, which are generated with IR approaches [1, 54, 55, 107, 134]. Furthermore, several authors reported the benefits of choosing deep learning (DL) approach over an IR-based recovery approach. Deep learning approaches, in general, can learn unstructured data of any format, as it can be trained to understand the domain knowledge of the system [66, 107], particularly, understanding the correlations between requirements and design documents [65, 149]. To be able to utilize the right traceability link recovery approach, software engineers need to understand which traceability recovery approach is suitable for which change impact sets. Furthermore, it is important to understand how these trace links can be recovered and the availability of support tools.

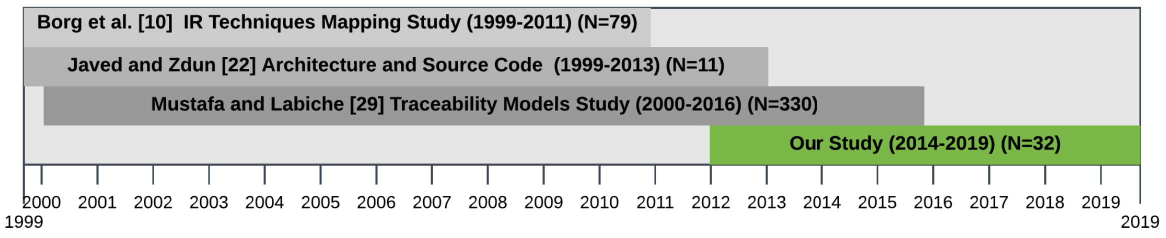


Figure 2.3: Overview of Traceability Literature Reviews Timeline

Figure 2.3 presents an overview of the timeline, the domain of automatic traceability link recovery approach, and the total number of primary studies in existing literature reviews in comparison to the one presented in this paper. In [29], the authors presented a systematic mapping study of IR-based traceability links recovery approach and enhancement strategies, covering the articles published during 1999–2011. They highlighted that there is no empirical evidence of any IR—based model consistently outperforming another. In contrast, in [74], the authors reviewed the studies on traceability published between

1999 and 2013, capturing the correlations between software architecture and source code. This study highlighted that semi-automatic traceability approaches appeared to be the most appropriate way to create trace links between software architecture and source code. Recently, in [113], the researchers reviewed the studies published between 2000 and 2016 on traceability models. However, none of these studies discussed these traceability link recovery approaches under the scenario of the CIA. In addition, there are no recent empirical studies on the current state of automatic traceability links recovery. Therefore, we carried out a review study of automatic traceability link recovery approaches under software CIA context to report empirical evidence and identify research gaps.

2.3 Research Method

We followed the guidelines of a systematic review by [84] and developed a protocol to plan, execute and report our results. The following sections outline the processes included in our planning phases.

2.3.1 Objectives and Research Questions

Our goal was to gather the state of the literature related to automatic traceability links recovery under the context of the CIA. Therefore, we answered the three complementary research questions (RQs), which are specified by the following criteria:

RQ1. What approaches have been used in recovering traceability links between artifacts to support the CIA process? We analyzed the approaches used to recover traceability links between software artifacts and the types of software artifacts that have been most frequently linked in trace recovery studies. We also investigated trace direction and degree of evaluation. Further, we studied whether the studies introduced any supporting tools.

RQ 2. Which change impact sets are covered? Based on Bohner's [27] impact analysis model, we analyzed four change impact sets, (i.e., requirements impact, design impact, program impact, and test impact) and reported which traceability studies widely cover impact sets.

RQ 3. What are the ways of recovering traceability links between artifacts to support the CIA process? We investigated what are the possible ways to recover traceability links between artifacts (e.g., transitive tracing).

The main difference between RQ 1 and 3 is the former focuses on identifying which automation approaches are used to generate the trace links between artifacts. In contrast, the latter investigates the case scenarios where transitive tracing is more beneficial to apply than direct tracing.

2.3.2 Protocol Development

In our SLR planning phase, we conducted an initial search for other SLRs concerning a similar scope of the traceability field. During our preliminary search, we found a few relevant studies, which fit our research objectives. The relevant ones are presented in Section 4.2. Accompanied by the already identified studies, we have used these SLRs as the basis to create our RQs and to develop our review protocol, which was conducted iteratively. The protocol document includes SLR research questions, search strategy, study selection criteria, quality assessment, data extraction strategy, data synthesis, and analysis guidelines, which are mentioned briefly in the following sections.

2.3.3 Search Strategy and Data Sources

Following the research objectives and the RQs, we selected four important terms for searching our literature: (1) Traceability, (2) Recovery, (3) Software Artifacts, and (4) CIA. Following that, we chose a variety of online databases, performed some straightforward searches in the publication's title, keywords, and abstracts, and then evaluated the coverage. While running the simple search, we created the search strings. The search strings were modified for various online databases.

ON ABSTRACT: (Abstract: trace*) AND (Abstract: recover* OR Abstract: maintain OR Abstract: link OR Abstract: establish) AND (Abstract: requirement OR Abstract: specification OR Abstract: architecture OR Abstract: design OR Abstract: code OR Abstract: implementation OR Abstract: test OR Abstract: bug) AND (Abstract: change OR Abstract: impact OR Abstract: analysis OR Abstract: system comprehension)

2.3.4 Study Selection

We are interested in collecting peer-reviewed articles, published between 2012-2019, focused on the automatic traceability links recovery used by the CIA. In the first step, we defined the selection criteria below.

- The publications are written in English
- Research explicitly mentioned that they are targeting trace recovery relating to software maintenance and CIA

- Studies containing empirical results (e.g., case studies, experiments, and surveys)

Figure 2.4 shows a summary of the search and selection process.

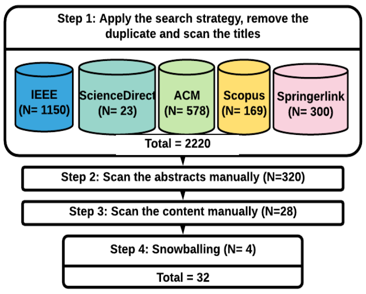

Step 1: Apply the search strategy, remove the duplicate and scan the titles
IEEE (N= 1150)
ScienceDirect (N= 23)
ACM (N= 578)
Scopus (N= 169)
Springerlink (N= 300)
Total = 2220
Step 2: Scan the abstracts manually (N=320)
Step 3: Scan the content manually (N=28)
Step 4: Snowballing (N= 4)
Total = 32

Figure 2.4: Search and selection process

- **Step 1:** We applied our search strings in the following five databases: (1) ACM Digital Library, (2) IEEE Xplore, (3) Science Direct, (4) SpringerLink, and (5) Scopus. Next, we discarded the duplicate papers. We scanned the title and excluded the irrelevant papers. The search results showed a high number of documents (2220 findings).

- **Step 2:** To extract the results, one author manually scanned the abstracts based on selection criteria. Then, the other authors checked a sample of papers randomly. The differences were resolved in the discussion between the authors. We included 320 studies in this step.

- **Step 3:** At the beginning, one author scanned the content of the papers based on the selection criteria and marked the records as relevant/irrelevant studies. The other authors reviewed the findings and selected the relevant studies together. We found 28 studies in this step.

- **Step 4:** Finally, one author performed a forward snowballing on our included studies that have high citations. Other authors reviewed the new findings. We found four additional papers in this step. In total, we identified 32 relevant studies.

2.3.5 Quality Assessment

We performed a quality assessment in two stages as follows:

- **Assessment of research design:** To assess the quality of studies, we reviewed the research design mentioned in the papers. This includes assessing the details of research objectives, design, and evaluation. Therefore, we filtered out studies that have poor research methods and evaluations, as well as research objectives not related to automatic traceability link recovery. We used the quality assessment checklist from [84] (See the quality assessment checklist here[1]).

- **Assessment of publication source and impact:** We checked the professional computer science CORE[2] journal/conference ranking site to evaluate the quality of the publications'sources.

2.3.6 Data Extraction and Analysis

To answer the research questions, we extracted the following demographic data for review: title, authors, type of outlet (journal or conference), name of outlets, publication year, type of trace artifacts, trace direction, trace recovery technique, quality of evaluation (e.g., research design), CORE[2] ranking and CIA coverage.

2.4 Results

We have included 32 relevant studies in this review. Concerning the publication channel, the studies were published in conference proceedings, workshops, and scientific journals. In comparison, 18 papers (56%) of the included studies were published in conference proceedings, 11 papers (34%) appeared in journals and 3 papers (9%) belonged to workshops. We identified that 29 papers of the included studies were published in high-ranking conferences and journals. Only three papers cannot verify ranking, but these papers were published in traceability-specific journals and workshops. Please see the primary classification here[3].

2.4.1 Traceability Links Recovery Approaches Between Software Artifacts

Several approaches have been proposed to recover knowledge-based traceability links between software artifacts of different types to assist in the impact analysis process. Please see our primary classification here[3]. Figure 2.5 illustrates the publications trend, grouped by traceability approaches and artifacts. We found a total of four different

[1]https://figshare.com/articles/Quality_Assessment_List/11775006

[2]http://www.core.edu.au

[3]https://figshare.com/articles/Primary_Studies_List/11775009

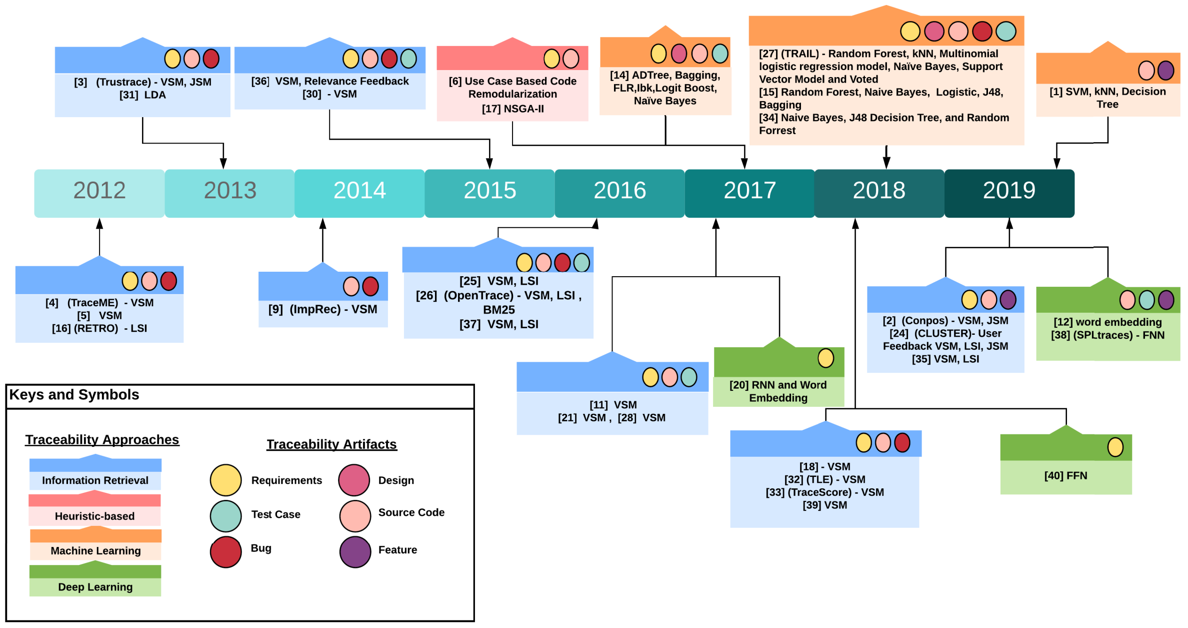

[3] (Trustrace) - VSM, JSM
[31] LDA
[36] VSM, Relevance Feedback
[30] - VSM
[6] Use Case Based Code Remodularization
[17] NSGA-II
[14] ADTree, Bagging, FLR,Ibk,Logit Boost, Naïve Bayes
[27] (TRAIL) - Random Forest, kNN, Multinomial logistic regression model, Naïve Bayes, Support Vector Model and Voted
[15] Random Forest, Naive Bayes, Logistic, J48, Bagging
[34] Naive Bayes, J48 Decision Tree, and Random Forrest
[1] SVM, kNN, Decision Tree
2012
2013
2014
2015
2016
2017
2018
2019
[4] (TraceME) - VSM
[5] VSM
[16] (RETRO) - LSI
[9] (ImpRec) - VSM
[25] VSM, LSI
[26] (OpenTrace) - VSM, LSI, BM25
[37] VSM, LSI
[11] VSM
[21] VSM, [28] VSM
[20] RNN and Word Embedding
[2] (Conpos) - VSM, JSM
[24] (CLUSTER)- User Feedback VSM, LSI, JSM
[35] VSM, LSI
[12] word embedding
[38] (SPLtraces) - FNN
[18] - VSM
[32] (TLE) - VSM
[33] (TraceScore) - VSM
[39] VSM
[40] FFN
Keys and Symbols
Traceability Approaches
Information Retrieval
Heuristic-based
Machine Learning
Deep Learning
Traceability Artifacts
Requirements
Design
Test Case
Source Code
Bug
Feature

Figure 2.5: Traceability Links Recovery Publications Trend (A rectangular box is colored according to traceability approaches. A circle symbol inside the top-right corner of the rectangle box represents the trace artifacts applied in the studies. Texts inside the rectangular box present the traceability approaches or tool names and techniques.)

approaches used in the studies (i.e., information retrieval, heuristic-based, deep learning, and machine learning). The reasons for adopting these distinct approaches in the context of traceability are discussed in the sections that follow.

2.4.1.1 Information Retrieval-Based Approaches

In the early years, several studies applied information retrieval (IR) approaches to tackle the challenges of recovering knowledge-based traceability links between various software artifacts (i.e., requirements, test cases, source codes, bug reports, and features). In total, 21 out of 32 studies used IR approaches to identify various change impact sets. The majority of software artifacts are written in natural language, which makes the IR-based trace recovery approach a logical choice because it allows for the recovery of links between artifacts of various sorts by calculating their textual similarity. The high textual similarity between two artifacts is assumed to be related to each other in these approaches. We identified that vector space model (VSM) [19, 20, 57, 122], latent semantic indexing (LSI) [57, 101, 145], Jensen and Shannon model (JSM) [6, 89] and latent Dirichlet allocation (LDA) [122] is the most commonly applied techniques in the studies, due to their easy implementation and set up.

In [20], the study presented the IR-based approach to identify outdated requirements by monitoring the source code changes. All too often, software engineers directly modified on the source code without updating the corresponding requirements. As a consequence, traceability links between these two artifacts are obsolete and cannot be used in impact analysis effectively. To mitigate the problem, the authors proposed to extract the trace query terms from the recent source code changes (e.g., the addition of a new method or a new class or a new package, deletion of an existing method or class or packages) to establish the links with the corresponding requirements. Similarly, in [57], the study illustrated the use of LSI to recover traceability links between bug reports and source code to estimate the impact set. In contrast, their approach extracted the source code query terms based on the commit change-sets co-occurrence concept (e.g., method A and method B are committed to three commit transactions together, these methods are considered as co-occurrence methods). Based on this concept, the authors established the most relevant trace links between bug reports and source code. In [6], the authors enhanced the way to identify outdated requirements by monitoring source code changes in bug fix history to identify the impact on original requirements. Similarly, in [145], the authors experimented with the use of VSM and LSI techniques to recover traceability links between source code and test case to identify the test case impact set.

Differently, in [122], the study proposed the source code class-based topic clustering approach (i.e., LDA) to establish traceability links between source code and requirements. The rationale behind this approach is that a class is an abstraction of a domain/solution object, and a use case is homogeneous and related to one specific topic. Later that year, in [28, 133], the researchers presented the idea of recovering traceability links between a new issue report and the previous issue reports in an issue repository to identify a set of potentially impacted artifacts [28, 133]. The study is predicted on the notion that older issue reports have closer textual ties to the most recent implementation artifacts than more recent issue reports. In [143], the authors proposed an approach to combine two similarity relevance scores from two sets of traceability links (i.e., one between requirements and source code, another one between requirements and commit messages) to improve the accuracy when establishing links between requirements and source code. Similar to [28], the study in [117] presented an approach called connecting link method (CLM) to recover transitive traceability links between two artifacts using the third artifact to overcome the problem of source and target artifact with no textual similarity between them. The approach is based on the assumption that if requirement 1 is implemented in test case 1 and test case 1 is related to source code 1, then source code 1 will include the implementation of requirement 1.

Similarly [108], the study presented the impact of trace retrieval direction on the accuracy in recovering trace links between requirements and code classes. They used standard VSM and established the trace links between these artifacts in a bidirectional way. Interestingly, their results indicated that there is a high correlation between the accuracy of the IR-based traceability recovery approach and trace direction. Recently, in [130], the authors presented an approach called trace link evolver (TLE) to detect the obsoleted trace links between the requirements and the source code by identifying changes between the two consecutive source code versions. In their approach, they used a source code refactoring tool to detect a wide range of source code change scenarios (e.g., add, remove, move, rename and merge) at both the method and class levels. Once the code changes were identified, they used the standard VSM to recover update-to-date links between the requirements and the source code. According to their research, the trace connections produced using TLE have a higher degree of precision than those produced using the standalone VSM technique.

In [101], the researchers presented a taxonomy-based traceability links recovery approach to establish trace links between the non-functional requirements and the source code. In their approach, non-functional requirements query terms are manually defined and maintained in a custom taxonomy database. Their IR engines, which are built on top of VSM and LSI models, used these taxonomy terms to establish trace links between non-functional requirements and the source code. Likewise, in [66], the study proposed a taxonomy-based traceability links recovery approach to establish the traceability links between Health Insurance Privacy and Portability Act (HIPAA) regulations and system requirements documents. A similar taxonomy-based approach is proposed in [148] where the authors create a requirement-based taxonomy approach to recover trace links between the requirements and the source code.

Different from [6, 57], the authors proposed an approach to extracting observed behaviors terms from the bug reports to leverage the IR-based approach. Observed behaviors are texts that describe the misbehavior of the system (e.g., “the menu doesn't open when I click the button”). In their approach, they manually extracted observed behaviors terms from the bug reports and used the standard VSM model to recover traceability links between bug reports and source code. Similarly, in [5], the authors conjectured a Parts-Of-Speech (POS) tagging method with a constraint-based pruning approach to increase accuracy. In their approach, all POS categories, i.e., nouns, verbs, adjectives, adverbs, and pronouns are first extracted from the required documents and the textual similarity between source code identifiers (e.g., class and method) is computed using VSM and JSM.

Recently, [89] proposed an approach called CLoseness-and-USer-feedback-based TracEability Recovery (CLUSTER), which established the trace links between requirements and source code based on code dependencies among classes to increase the accuracy between the links. In their approach, the code dependencies are calculated based on the degree of direct interaction (e.g., method calls, inheritance, and class usage) and indirect interaction (e.g., reading or writing the same data). Recently, in [138], the authors proposed a feature tagging approach to recover traceability links between requirements and source code. A feature is a short textual description of functionality that presents business value, whereas a feature tag summarizes the feature shortly and concisely. Their approach recommends labelling requirements and source code with corresponding tags during development and using these tags.

2.4.1.2 Heuristic-Based Approaches

In earlier IR-based methods, a human expert was required to manually and frequently check the candidate link list because it was time-consuming and fallible. To reduce the time and effort of the verification process [23], researchers presented the concept of multi–search criteria–based traceability recovery approach to establishing trace links between the requirements and the source code. They experimented with the Non-dominated Sorting Algorithm (NSGA–II) using similarity scores, change frequency, and change recency as three weighing criteria. They used cosine similarity to calculate the similarity scores between the requirements and the source code. The metadata for the remaining two criteria —source code's frequency of change and recency of change —are extracted from the source code version history. This method optimized the accuracy of candidate link lists by creating trace linkages based on both the history of modifications and the semantic similarity of software objects. All of their experimental results harmonically achieved high precision results. A unique technique called Re-modularization, which converts source code syntax into sentences in natural language, was used by the authors of [23] to improve the IR-based approach. The trace linkages between the source code and use cases are then created using these words. Converting source code syntax to natural language format reduced the knowledge gap between artifacts when calculating the similarity between them.

2.4.1.3 Machine Learning-Based Approaches

In this review, we identified that most primary studies applied machine learning approaches to automatically verify the candidate link list generated from IR approaches [54, 55, 107]. In [1], the author's employed ML classifiers to suggest relevant attributes to source code comments.

The study described an approach called Estimation of the Number of Remaining Linkages (ENRL) in [54] to find the number of positive trace links that are still present in the ranked list generated by IR approaches. They trained the seven machine learning classifiers (i.e., ADTree, Bagging, Fuzzy Lattice Reasoning, IBk, Naive Bayes, LogitBoost, and ZeroR) with a set of classified gold standard trace links to identify positive and negative links from the ranked lists. The results of their findings indicated that the ZeroR classifier produced the lowest accuracy results, with a Mean Relative Error (MRE) of 1. Similarly, Mills et al. [107] presented a framework called TRAceability Links Classifiers (TRAIL) to automatically verify the validity of the ranked trace links, which are generated with the IR model. They used three features, namely: cosine similarity, query quality metrics, and document statistics to validate the trace links. With six machine learning classifiers, including Random Forest, k-Nearest Neighbor (kNN), Multinomial Logistic Regression Model, Naive Bayes, Support Vector Model (SVM), and Voted, they conducted TRAIL experiments. Their findings indicated that the Random Forest classifier outperformed the other five classifiers. Also in [148], the authors trained Random Forest, Naive Bayes, Logistic, J48, and Bagging Models to verify the validity of trace links between requirements and source code. However, the authors did not report clearly which ML classifiers perform best. Likewise, in [134], the authors proposed an approach by training ML classification models to predict the potential trace links between issue reports and source code. They used Naive Bayes, J48, Decision Tree (DT), and Random Forest classification models to validate the possible trace links. The results of their findings highlighted that Random Forest outperformed the other two classifiers.

Differently, in [1], the authors proposed a feature-annotation recommendation approach to suggesting a developer add an annotation feature in the newly added code before check-in into a version-control system. They used three machine learning classification models, i.e., SVM, kNN, and DT to predict the source code location where the feature annotations are missing. The method created a list of potential associated features for the new modifications using the history data from earlier change-sets. After training with 60 commits change-set histories, their findings showed that the kNN classifier achieved the maximum accuracy.

2.4.1.4 Deep Learning-Based Approaches

Recently, deep learning techniques have become popular in the traceability context to address the knowledge gap problem. We found four studies that used recurrent neural networks [65], feed-forward neural networks [149] and word embedding [43] to recover traceability links between software artifacts of different types (i.e., requirements, source code, test case, and features). In [43], the authors demonstrated that a word embedding approach outperformed an LSI model when establishing trace links between the test case and source code.

In [65], the study combined word embedding and the recurrent neural network (RNN) approaches to eliminate the knowledge gap in recovering traceability links between requirements and design documents. The approach can be divided into two layers, namely: the word embedding mapping layer and the semantic relation evaluation layer. In the word embedding mapping layer, they first converted the collection of requirements documents to the word embedding vectors format and identified the semantic relations between terms using RNN. Conversely, the same process ran for the design documents. Next, the outcomes of the embedding layer are passed to the evaluation layer to recover the links between them. In the evaluation layer, it first calculated the vector distance and vector direction for two semantic vectors, then passed the resulting vectors to the sigmoid and softmax (e.g., 1- valid, 0- invalid) functions to calculate the relations between the two vectors.

Similarly, in [149], the study used the word embedding and the feedforward neural network (FNN) approaches to address the polysemy issue in recovering trace links between requirements. Polysemy in this context refers to "the coexistence of multiple meanings for a term appearing in different requirements" [149]. In their approach, the authors extended the standard IR recovery approach with the term-pair ranking model and cluster ranking model. When collecting requirements, the term-pair ranking model used word embedding and FNN to identify the list of polysemy terms. The cluster ranking model used these polysemy terms and updated the term-to-requirements matrix accordingly. They evaluated their approach with two baselines IR models (i.e., VSM and LSI). Their results indicated that the accuracy of the results increased by eliminating the polysemy issue. Recently in [43], the researchers trained the word embedding model with raw source code, the abstract syntax tree structure of source code, and the test case to predict the relevance test cases. The authors compared their word embedding model with the standard LSI model and reported that word embedding outperformed LSI when recommending the top 1 most similar class.

Table 2.1: Traceability Tools

Tool/Approach	**Information Retrieval**	**Deep Learning**
TraceME	✓	-
RETRO	✓	-
OpenTrace	✓	-
SPLTrace	✓	✓

In terms of tool support, we identified four traceability tools (i.e., TraceME, RETRO, OpenTrace, and SPLTrace). Table 2.1 presents the traceability tools proposed in primary studies. In [19], the study proposed the traceability links recovery tool called TraceMe eclipse-plugin to recover the traceability links between source code and requirements during software development. The tool supports the graph view to visualize the trace links between source code and requirements. Similarly, in [20], the study presented the standalone traceability recovery tool called RETRO to establish trace links between artifacts of different types. In [106], the authors used an open-source traceability workbench framework called OpenTrace to experiment with the effects of three IR techniques. OpenTrace provides a reusable IR component to perform the analysis task easily. Lastly in [146], the study presented the automatic feature-code traceability tool called SPLTrace. The tool is built upon four IR models (i.e., Class Vector, Extended Boolean, Latent Semantic index, and BM25) and one deep learning model (i.e., Feed Forward Neutral Network). In SLRTrace, trace links between the features and the source code can be generated using one of the models.

In summary, TraceMe and SPLTrace can be categorized as special purpose tools especially focused on identifying the impact set between requirements and source code. By contrast, RETRO can be considered as a general purpose tool suited for analyzing the impact between software artifacts of different types. Finally, OpenTrace is a traceability workbench framework with various IR components (e.g., data preparation, experiment execution, and evaluation) as applicable for experimental analysis of various IR approaches.

2.4.2 Traceability Direction and Evaluation

In [108], the authors highlighted the impact of trace direction on the links recovery process. Thus, we grouped the studies based on two traceability directions (i.e., unidirectional and bidirectional). Figure 2.6 shows the classification of the studies by trace direction.

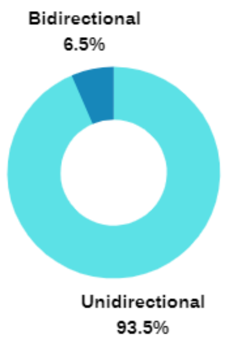


Figure 2.6: Distribution of Publications by Traceability Direction

In the unidirectional traceability group, we included the studies that evaluated their approach to recovering trace links from one artifact to another either in one direction (e.g., requirements to design, requirements to source code). We identified that 30 (93.5%) out of 32 studies evaluated their approaches in a unidirectional way. Only two studies [108, 117] evaluated their approaches in a bidirectional way. To the best of our knowledge, the approaches focused on tracing between two natural language artifacts (e.g., requirements, test case, design) are possible to trace in a bidirectional way [65, 149]. However, the approaches such as feature-based tracing [1, 66, 138] and code to requirements tracing [20, 43] required a specific data tokenization process for source code, as these approaches may not be applicable to trace in a bidirectional way. Due to the lack of evaluation in the studies, we cannot conclude which approaches are applicable for bidirectional tracing.

In terms of the degree of evaluation, we reported the research methods and the type of datasets used in the studies. We used the quality assessment checklist from [84] (See the quality assessment checklist here[1]) to assess the research methods.

Figure 2.7 presents the research methods and datasets used in the primary studies. We identified that 29 (90.6%) out of 32 studies applied experimental research to study the effectiveness of their approaches. Only (9.4%) of studies [28, 143, 145] used case study methods. In [28], the researchers studied the impact of automatic traceability-based CIA approaches in two industrial domains (i.e., automation, and telecommunication).

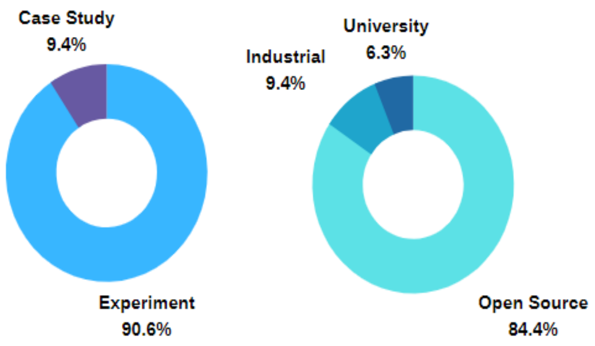


Figure 2.7: Distribution of Publications by Research Methods (on the left) and Datasets (on the right)

Their findings indicated that their approach can identify 40% of the potential program change set effectively in real-world datasets. Similarly, in [143], the study applied a standard VSM model in a Japanese software company to establish the trace links between requirements and source code. Due to data confidentiality, the study anonymized the system details. In [145], the authors experimented with the two IR models (i.e., VSM and LSI) in an embedded system, which produces both hard and software products. The study evaluated IR models by establishing links between the source code and the test case. Based on their findings, an IR-based approach can only identify 38% of the potential test cases in an industry setting. We identified that 27 (84.4%) out of 32 studies used open source projects to evaluate their approach. Presumably, open source projects are close to industrial projects. Only two studies used university projects to evaluate their traceability tools (RETRO and TraceMe) [19, 20].

2.4.3 Support Change Impact Set

In this section, we classified the primary studies based on Bohner's four impact sets (i.e., requirements impact set, design impact set, program impact set, and test impact set) and presented the results in Table 2.2 [27]. The last row of Table 2.2 describes the lists of the acronym used in the table.

2.4.3.1 Requirements Impact Set

In this section, we included the studies that used requirements artifacts as target artifacts in their trace recovery approach to identify the impact on the requirement level. We identified four source artifacts (i.e., regulatory requirements [66, 148], low

Table 2.2: Distribution of Publications by Change Impact Sets

Impact Set	Artifacts Links	Studies
RIS	SC-R, RR-R, R-R, F-R	[20],[19], [6], [117], [66], [108], [54], [65], [107], [148], [149], [138]
DIS	R-D	[117], [54]
TIS	SC-TC, R-TC, D-TC	[117], [145], [54],[107]
PIS	R-SC, Bug-SC, TC-SC, F-SC	[57],[122], [28], [143], [117], [101], [106], [23], [59], [34], [108], [54], [133], [60], [55], [148], [134],[130], [1], [5], [146], [138], [43]
Acronym : RIS - Requirements Impact Set, DIS - Design Impact Set, TIS - Test Impact Set, PIS - Program Impact Set, R- Requirements, RR - Regulatory Requirements, F - Features, D - Design, SC - Source Code, TC - Test Case		

level requirements [65, 149], source code [6, 19, 20, 54, 107, 108, 117] and features [138]) used in the studies to assess the requirements impact scope. In [66, 148], the authors recovered links between regulatory requirements and requirements using a taxonomy-based approach to identify the impact on the requirements level. Similarly, [65, 149] used a deep learning-based traceability links recovery approach to recover trace links between high-level requirements and low-level requirements. Similarly in [6, 19, 20, 108, 117], the authors mined the class-level and method-level changes in source code from commit histories and established the links with existing requirements to identify the outdated requirements. The authors of [54, 107] used machine learning models to anticipate the trace linkages between requirements and source code. Recently, a feature tagging approach has been introduced in [138] to maintain the links between requirements and other software artifacts. Interestingly, we cannot find studies that used design and test cases as source artifacts. All too often, software engineers might update these artifacts directly and forget to update the original requirements. As a consequence, traceability links between these artifacts become obsolete and cannot perform the CIA effectively.

2.4.3.2 Design Impact Set

In this section, we included studies that used source code artifacts as target artifacts in the studies. Interestingly, only two studies [54, 117] focused on recovering links between requirements and design artifacts, possibly due to the scarcity of datasets. Both studies [54, 117] evaluated their approaches with the same test datasets, due to the availability of golden standard answer sets. In [117], the authors used the standard VSM model to recover the links between requirements and design artifacts. The work in [54] employed a machine learning-based methodology to validate potential connections between requirements and design artifacts that are produced by an IR engine.

2.4.3.3 Test Impact Set

In this section, we included studies that used test artifacts as target artifacts in the studies. We identified two source artifacts (i.e., design [54, 117] and source code [107, 145]). In [54, 117], the authors evaluated the same datasets to recover links between design and test case artifacts to identify the test impact set. In [145], the authors used two IR models (i.e., VSM and LSI) to identify the impact test case set by linking with the source code artifacts. Similarly, in [107], the study used a machine learning approach to automatically verify the candidate links between source code and test artifacts, which are generated with the IR approach.

2.4.3.4 Program Impact set

In this section, we included studies that used source code artifacts as target artifacts in the studies. We identified four source artifacts (i.e., requirements [5, 23, 54, 55, 57, 59, 89, 101, 108, 122, 130, 143, 148], bug report [28, 34, 60, 106, 133, 134], test case [43, 117] and features [1, 138, 146]) used in the studies to identify the program impact set. In [5, 57, 89, 101, 108, 122, 130, 143, 148], the authors applied an IR-based approach to establish trace links between requirement and source code artifacts to identify the program impact set. To increase the accuracy, the study conjectured a heuristic-based approach in [23, 59]. In [54, 55], the authors trained the machine learning models to validate the trace links between requirements and source code artifacts. Similarly, the authors conjectured the IR-based approach [28, 34, 60, 106, 133] and the machine learning approach [134] while recovering trace links between a bug report and source code. Likewise, the study applied the standard VSM model to recover trace links between the test case and source code in [117]. In [43], the study used the word embedding model to establish trace links between the test case and source code to identify the program impact sets efficiently. Recently, the study introduced a feature tagging approach to maintaining trace links between source code and other software artifacts ubiquitously during software evolution [1, 138, 146].

2.4.4 Traceability Links Recovery Ways

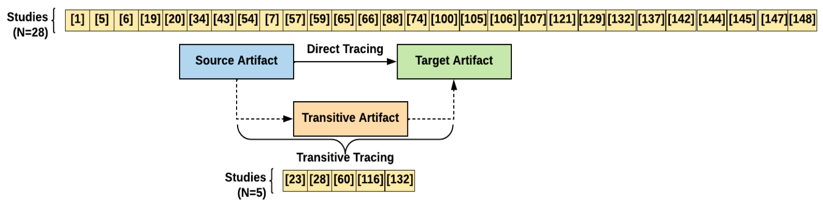


Figure 2.8: Traceability Links Recovery Ways

Figure 2.8 illustrates the two types of traceability links recovery ways used in the primary studies. The majority of studies used direct tracing methods to create explicit trace links between the source artifact and the target artifact. We identified five studies [23, 28, 60, 117, 133] that used the transitive tracing approach, where traceability links between two artifacts are recovered by joining with a third artifact.

In [28], the study presented the transitive tracing approach to identifying a set of potentially impacted artifacts for a new issue report. In this approach, the author used the existing issue reports as the transitive artifact, in recovering traceability links between a newly issued report and its related source code. The assumption is that the previous issue reports have more textual similarity relations with the current state of source code artifacts than a new issue report. Similarly, in [117], the study proposed an approach called Connecting Link Method (CLM), in which the trace links between two artifacts are established by mapping transitively sourced artifacts to the target artifacts via the third artifact. The authors attempted to overcome the textual similarity gap between source and target artifacts using the CLM approach. In the CLM approach, the search terms were extracted from the third artifact to recover trace links between the source and the target artifacts.

Likewise in [23], the study used the existing issue report as a transitive artifact, recovering traceability links between source code and use cases. In their approach, they first extracted the lists of corresponding issue reports related to source code using source code version history. Next, the textual similarity between issue reports and use cases is calculated to identify the impacted source code. Gharibi' [60] presented a similar approach. Their approach established the traceability links between a new feature request and the existing feature requests to identify the impact on source code. Similarly, in [55], the authors established the trace links between a new requirement and the existing requirements to recover the lists of impacted source code classes. In [133], the researchers used the existing requirements and similar bug reports as transitive artifacts in localizing related source code areas to fix the new bug report.

2.5 Discussion

The discussion elaborated on the findings by grouping identified traceability links recovery approaches according to our RQs. This study examined the literature concerning the use of traceability links recovery approaches in the CIA context, noting the underlying challenges and limitations of the current studies.

2.5.1 Findings of RQs

We identified that four different approaches (i.e., IR-based, heuristic-based, machine learning, and deep learning) could be used in recovering traceability links between software artifacts of different types. The majority of the primary studies focused on enhancing IR-based approaches to identify outdated trace links and generate an impact analysis report. VSM, LSI, JSM, and LDA are the most popular IR models reported in the studies. There are no studies reported in which IR models outperform others. Approaches like machine learning and heuristic-based approaches are used to predict the validity of candidate trace links generated with the IR engine. Among multiple ML classifiers, the Random Forest provides promising accuracy in most studies. Recently, some studies conjectured to substitute the IR-based approach with a deep learning approach in recovering trace links. So far, one study [43] reported that word embedding outperforms LSI in establishing trace links between test cases and source code. In terms of trace artifacts, we identified six artifacts (i.e., requirements, design, source code, test case, bug report, and feature). Among them, requirements, source code, and bug reports are the most frequently linked artifacts in the studies. Very few studies focused on design and test case artifacts linking, presumably due to scarcity of datasets. Recently, three studies [1, 138, 146] a featured artifact as a transitive artifact to recover trace links between requirements and source code. In terms of trace direction, 30 out of 32 studies evaluated their approaches in a unidirectional way, as it is hard to conclude whether their approaches could feasibly be applied in a bidirectional way. In terms of tool support, all four tools are built upon an IR-based approach, where only one tool supports a deep learning approach. In terms of evaluation, 29 out of 32 studies used experiential research and assessed their methodology using either university projects or open source projects.

Only three studies applied a case study and evaluated with industrial data. Due to data confidentiality, the context of datasets is not provided in detail.

Our review highlighted that the majority of studies focused on identifying program impact sets due to source code, making it the most frequent change area in software development. The studies used requirements, bug reports, test cases, and feature artifacts as source artifacts to identify the program impact area. Interestingly, design artifact is left out of this impact set study. The second most frequent study area is the requirements impact set. The studies evaluated their approaches with either forward tracing (e.g., regulatory requirements to requirements) or backward tracing (e.g., source code to requirements) direction. Test impact set studies followed the third position and were evaluated with source code, requirements, and design artifacts. Only two studies [54, 117] focused on assessing the design impact area, where both studies used requirements as a source artifact. The analysis of the existing studies highlighted that recovering trace links between artifacts of different types can be established either directly or transitively. The direct tracing approach is applicable for explicit tracing scenarios where the source and target artifacts have a high textual similarity between them. Hence, the direct tracing approach can be used in recovering trace links between existing artifacts of the system, whereas the transitive tracing approach is useful in establishing trace links between a new artifact (e.g., a new feature request, a bug report) and the existing artifacts (e.g., source code). However, the challenging part of these approaches is finding the right transitive artifact for various CIA tasks. To the best of our knowledge, there is no traceability tool available to support a transitive artifact approach, as it is challenging to extend the research in this area.

2.5.2 Limitations and Future Work

Based on our findings, we reported the gaps and challenges of the current studies, which need improvement to leverage the automatic traceability links recovery approach to support impact analysis effectively. To further advance the automatic traceability research, we recommend the key improvements below:

- Focus on tool enhancements to support trending approaches (e.g., machine learning and deep learning)
- Focus on building a feature recommendation system to remind software engineers to annotate a new artifact (i.e., use case, design, test case, and source code) consistently during software evolution

- Emphasize design and test impact sets area, as identifying design and test impacts are equally important as the other two impact sets in the CIA process

- Focus on building a traceability system beyond the text-based recovery approach (e.g., recovering traceability links between design images and requirements)

- Evaluate industrial datasets and survey the practitioners to gain valuable feedback for further improvements

2.6 Threats To Validity

We assessed the threats to the validity of our review based on construct, reliability, and internal validity measures. We reported our review deviations from the study guidelines [84].

The first threat to validity is the construct validity concerned with the selection of the studies and the relevance of review in the field. To mitigate this threat, we constructed our search strings by referring to previous review studies. We included all possible keywords to cover abbreviations, synonyms, and morphological root forms (e.g., source code, architecture, trace*). We ran our search on five different databases to cover a broader scope of concerns. To deal with the threats to the relevance of study selection, one author applied the selection criteria and another author validated 10% of the selected studies. The second threat is reliability, concerned with how the authors carried out the data extraction, and interpretation and justified the findings. To eliminate this threat, one author extracted data from all the selected publications and the other authors individually repeated the same process for 20% of the selected studies. Then, the authors discussed the differences and came up with the same conclusions.

The third threat is the external validity concerned with the scope of the review. Our review focused on automating traceability studies to assist the CIA process as the scope is tight. We do not claim that our review applies to other areas of impact analysis (e.g., dependency impact analysis [27]). Thus, the external validity threat is minor. Furthermore, as the review protocol development is presented in detail in Section 2.3 as other researchers can verify the validity of the findings with the search strategy, selection criteria, and applied data extraction. Lastly, internal validity is concerned with the treatment and the outcome. This concern is negligible because we report our findings using a combination of empirical investigations and descriptive statistics.

2.7 Chapter Summary

Recovering traceability links between various software artifacts during the software development process is a challenging task for software practitioners, especially in the safety-critical system. In this chapter, we systematically review the automatic tractability link recovery approaches conjured in previous studies published between 2012-2019. We summarized the findings in terms of approaches, focus trace artifacts, workbench tools, and trace links direction to identify gaps and the challenges of current studies. Our findings demonstrate that one of the fundamental difficulties in creating traceability relationships between various artifacts is a knowledge gap issue, where knowledge is defined as the syntax and semantics of the artifacts. The majority of the previous studies address the problem by taking advantage of advancements in natural language process techniques, as most software artifacts are written in natural language. To further reveal the comprehensive traceability recovery knowledge, we group the investigations based on a CIA. This chapter provides new insight into traceability recovery approaches, limitations, and future work.

CHAPTER 3

INTERACTIVE TRACEABILITY LINKS VISUALIZATION USING A HIERARCHICAL TRACE MAP

3.1 Overview

Traceability has been mentioned in a good deal of software development methodologies as part of software engineering practice. Maintaining the trace links between software artifacts is crucial to performing various software engineering activities including change impact analysis [119], requirements coverage analysis [19], requirements verification and validation [99], test case selection [131], compliance verification [66], issue assignment [117] and bug localization [160]. Over the last few decades, researchers have attempted to recover trace links automatically using Information Retrieval (IR) techniques. The idea behind such techniques is based on the assumption that most software artifacts, such as requirements, use cases, interaction diagrams, source code, and test cases contain textual descriptions and meaningful source code identifiers [11]. IR techniques recover traceability links based on the similarity between the text contained in two related artifacts of different types [98, 109]. Although traceability fields are advancing the results presentations by embedding visualization techniques, bug triage fields still engage with textual representation. Similar textual similarity approaches are applied in bug triage studies [10, 12] to recover the links between historical issue reports and a new issue report to recommend potential developers and issue types. With this motivation in mind, we use the hierarchical trace map visualization technique to identify the links between three primary software artifacts (e.g., requirements, source code, and test case) as the first exploratory study. The main intention of the hierarchical trace map visualization approach is to embed in the bug triage process model. For example, once the bug recommendation model predicts the top-ranking developers, the system intends to automatically map the list of source code change areas recently developed or modified by these developers by constructing the transitive links between bug reports and source code via requirement or test cases to assist in selecting the most relevant developer from the list.

Traditionally, trace links are presented either in two or multidimensional matrix formats, which are commonly known as a Requirements Traceability Matrix (RTM) [167]. Likewise, tabular grid style formatting has been used in automatic trace recovery tools [47, 72] to present result sets for analyst verification. Even though the tabular format is applicable for small dataset verification; it is hard to explore interrelationships between artifacts conveniently. Recent graphical-based approaches like hierarchical leaf node [39], tree view [35, 123], Sunburst [105], and Netmap [105] can only present a coarse-grain view of traceability between two artifacts at a time. The main drawback of these approaches is that they fail to explore relationships between multiple artifacts interactively to comprehend the overall structure of a system. Therefore, it is time-consuming and resource-intensive to explore and interpret the system. Traceability tools like Traceclipse[85] and TraceViz [103] are platform dependent and specific for JAVA development environment. The manual trace link maintenance approach and the cypher query language are used by researchers in [53] to extract the trace links between different requirement artifacts. However, tracing implementation artifacts such as source code is not included in their experiments. In this chapter, we present a hierarchical trace map visualization approach to support system comprehension and change impact analysis interactively.

To analyze the change impact area in various software artifacts, trace links need to be drawn between high and low level artifacts; every object created at a high-level artifact (a source) must flow to a low-level artifact (a target) during any software development process. The analysis involves tracking the flow of values from requirements artifacts to sink through a sequence of downstream software artifacts represented by both coarse-grained and fine-grained views in the trace link visualization approach. To be intuitive and interactive, its underlying trace links visualization technique must also be narrative and animating.

Current change impact analysis techniques include the requirements traceability matrix (RTM) [37, 167], textual reference [127], hyperlinks [126] and graph & chart [50, 167]. There are two ways to perform impact analysis namely requirements analysis and program analysis. The former traces the flow of requirements iteratively at each design specification through stakeholders'objectives, while the latter tracks the flow of system design in the program through static program analysis. Language barriers make it harder to track down the latter because programs are written in machine-translatable language while design requirements are written in natural language. Among all published trace links visualization techniques, TraceME [19, 35], Traceclipse[85], and TraceViz [103] studies used graphs and chart-based approaches to present trace links between design and program artifacts. This is an eclipse-plugin tool designed to recover the trace links between source code and requirements. The tool applies a dependency graph, treemap, hierarchical tree, and coloured and labelled squares to display traceability links for a specific source or target artifact. It allows users to analyze links of a selected source artifact or a chosen target artifact clearly and concisely. However, it is unable to present links for multiple artifacts at the same time. Also, these approaches are not narrative enough to provide the big picture of the overall impact area interactively. The user needs to select the source artifact attentively to identify impact areas in the target artifact. Also, these approaches are only accessible on the development platform as it hinders the impact analysis of other project team members.

This research drew its inspiration from the graph-based visualization techniques works of [19, 85, 103]. Our goal is to build a standalone hierarchical trace map (HTM) visualization approach by using an IR-based traceability link recovery approach. Like previous studies, HTM uses textual information from artifacts to construct trace links between artifacts. HTM is expected to be not only as intuitive as previous works but also much more interactive than traditional approaches by using transition and animation.

Although TraceME [19, 35], Traceclipse [85], and TraceViz [103] are feasible to perform impact analysis between requirements and source code, it requires interaction with other software artifacts to present a narrative view of the overall impact area. Thus, the trace link visualization approach needs to be implemented in an environment where all software artifacts can interact with each other. In addition, the approach must be able to present multiple software artifacts in both coarse-grained and fine-grained views without causing visual cluttering.

To break the cycle of development platform development-dependent challenges, HTM is implemented using a standalone web-based application, which can interact with multiple software artifacts. HTM proceeds in five phrases, as shown in figure 3.3. Their functions are outlined below, and Section 3.3 provides details on the results.

Phase 1: Defined Artifacts In this step, we identified source and target artifacts to construct the links between them. To this end, we provide the example with three standard software artifacts (use cases (source artifact), test case (target artifact), and source code (target artifact)).

Phase 2: Pre-processing This is built for text pre-processing functionalities. To increase model accuracy, we applied common natural language pre-processing techniques to each artifact, such as stopping and stemming. Artifacts written in natural languages such as use case and test case include the stop words that occur commonly across these artifacts and are trivial for similarity calculation. Also, stemming the phrase to its root form by removing a part of a word helps compare the similarities between the two artifacts. The accuracy of recovery trace linkages between the use case and source code was also leveraged in this experiment using manually constructed source code class descriptions.

Phase 3: Trace Links Recovery We applied the standard vector space model (VSM) document classification model from Tracelab [80] component library to recover the trace links between artifacts. Tracelab is an open-source traceability framework component that provides a collection of reusable and configurable natural language processing functionalities. We compute the similarity between the two artifacts using cosine similarity. When the cosine value is equal to 0 means no similarities. Whereas a cosine value closer to 1 imply that there is a similarity between the two artifacts.

Phase 4: Verification This phase involved comparing the trace link results to ground truth values, removing them from the ranking list, and verifying the results.

Phase 5: Visualization Finally, we feed the ranked lists with corresponding parent artifacts to the HTM visualization module. HTM creates nodes for each artifact on the top layer and its target artifacts on the lower layer to demonstrate a hierarchical layer view of the system. As presented in the figure 3.4, we designed HTM in a hierarchical structure where the coarse-grained view of trace artifacts is illustrated in a triangular shape. To visually represents the entire impact area, the source artifact is shown at the top level of the triangle while its decedents' artifacts are shown at the bottom level.

Hierarchical in this context means recovering traceability links between multiple artifacts, which have parent-child relationships between them. HTM uses the colour nodes to present different artifacts (e.g., Orange –Use Case, Red –Source Code, Green –Test Case), and edges convey information about the linkages between artifacts. Each node has the interactive functionality to transition to a fined-grained view, which is presented in a one-layer circular ring format. As illustrated in the figure3.5, HTM supports two types of tracing; direct tracing and indirect tracing between multiple artifacts. The former means recovering traceability links between two artifacts of different kinds. For example, recovering traceability links between the use case and source code. The latter means recovering traceability links between source and target artifacts using an existing artifact of the system. For example, recovering traceability links between the test case and source code using the use case as a transitive artifact.

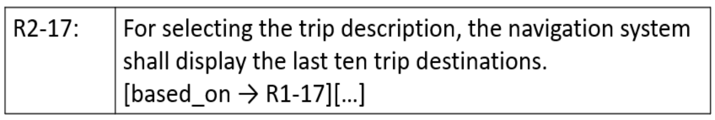
3.2 Background

There are various techniques to visualize traceability links between artifacts. In this section, we present the four common visualization mechanisms namely; requirements traceability matrix (RTM), textual references, hyperlinks, and graphs & charts.

Requirements traceability matrix (RTM) is the most common presentation technique used in the traceability context [37, 167]. RTM supports a multi-dimensional view to present two or more artifacts in table matrix format. In RTM, source and target artifacts are displayed in rows and columns whereas trace links are marked in the cells. In the multi-dimensional view, all trace artifacts are displayed in columns and ordered according to the software development process (e.g., functional requirements, design documents, and test cases).

To convey the traceability results, the relationship between the source and target artifacts is described in textual annotation form for textual reference. Figure 3.1 in the text below illustrates. Requirements 2-17 and 1-17 are interconnected in the picture.

To present the tractability results, the relationship between the source and target



R2-17:	For selecting the trip description, the navigation system shall display the last ten trip destinations. [based_on → R1-17][...]

Figure 3.1: Documenting traceability relationship using textual references [126]

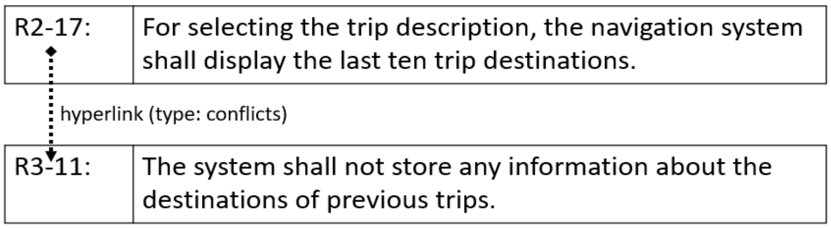

R2-17:
For selecting the trip description, the navigation system shall display the last ten trip destinations.
hyperlink (type: conflicts)
R3-11:
The system shall not store any information about the destinations of previous trips.

Figure 3.2: Documentation of traceability relationships using hyperlinks [126]

artifacts is documented in the Hyperlinks format. A reference example for hyperlinks is shown in Figure 3.2.

In recent years, various visualization techniques have been used to present traceability data. Given that the graph supports hierarchical layout views, hierarchical tree views and tree map techniques are frequently used to link requirements and source code artifacts. Bar graphs, line charts, and objects are used in various research studies [50, 167] to show the trace linkages between two or more items (e.g. people). Sunburst and Netmap are also used to display trace information in radical layers formats [105].

In the next section, we present how our raw hierarchical trace map visualization approach is used to present the artifact relationships interactively. We designed our conceptual model based on the standard components of the IR-based traceability recovery process [46]. Figure 3.3 illustrates the high-level structure of the hierarchical trace map model. Due to the availability of numerous software artifacts, such as use cases, interaction diagrams, test cases, and source code class descriptions, we created the WebTrace [1] application and assessed it using the EasyClinic dataset to experiment with our conceptual model.

[1]`http://www.webtrace.tech/Home/VisualizeTraceLinks`.

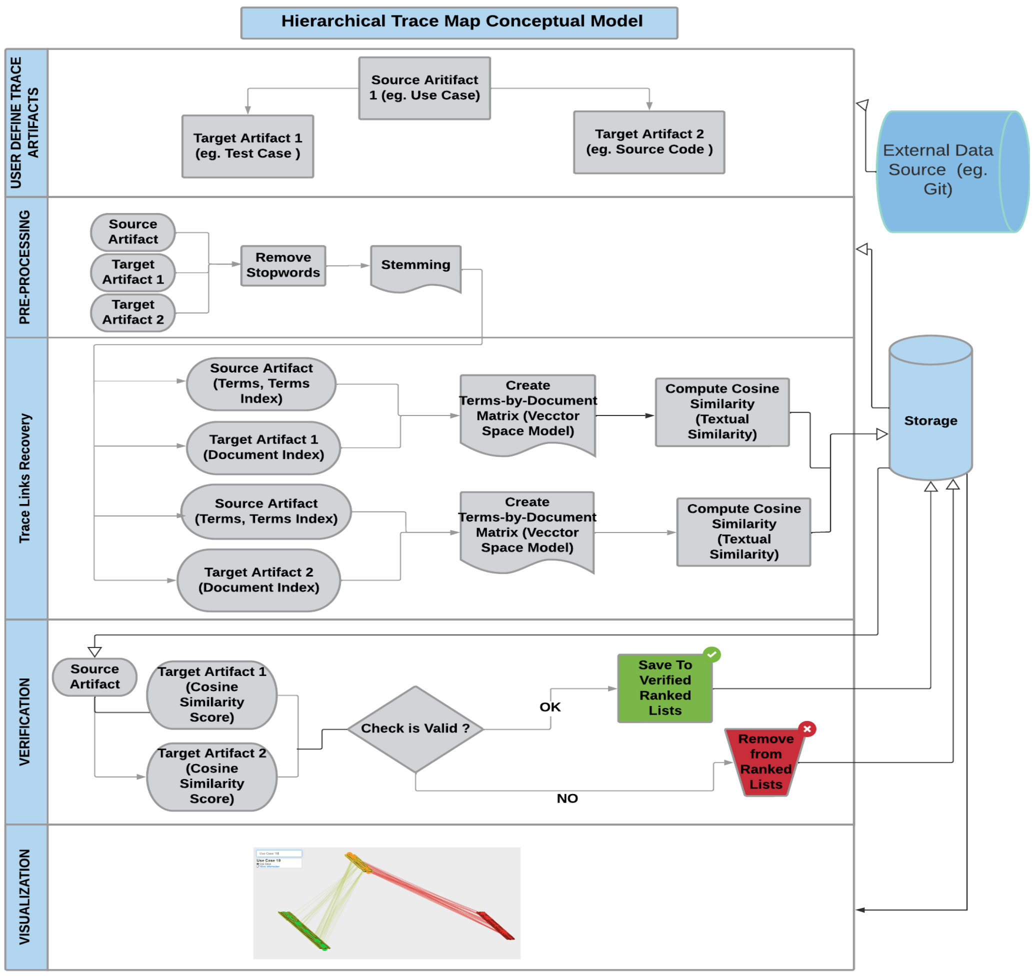

Hierarchical Trace Map Conceptual Model
USER DEFINE TRACE ARTIFACTS
Source Aritifact 1 (eg. Use Case)
Target Artifact 1 (eg. Test Case)
Target Artifact 2 (eg. Source Code)
External Data Source (eg. Git)
PRE-PROCESSING
Source Artifact
Target Artifact 1
Target Artifact 2
Remove Stopwords
Stemming
Trace Links Recovery
Source Artifact (Terms, Terms Index)
Target Artifact 1 (Document Index)
Create Terms-by-Document Matrix (Vecctor Space Model)
Compute Cosine Similarity (Textual Similarity)
Source Artifact (Terms, Terms Index)
Target Artifact 2 (Document Index)
Create Terms-by-Document Matrix (Vecctor Space Model)
Compute Cosine Similarity (Textual Similarity)
Storage
VERIFICATION
Source Artifact
Target Artifact 1 (Cosine Similarity Score)
Target Artifact 2 (Cosine Similarity Score)
Check is Valid ?
OK
NO
Save To Verified Ranked Lists
Remove from Ranked Lists
VISUALIZATION

Figure 3.3: Hierarchical trace map model

3.3 Visual Support for Traceability Links Visualization

The main goal of our Hierarchical Trace Map is to provide visual support in exploring the overall structure of the system in one area to reduce the tasks of cross-referencing and connecting multiple trace link results. It intends to provide an explicit dependency between artifacts of the system. Figure 3.4 presents the unfiltered view of the use case, test case, and source code artifact relations from the EasyClinic dataset. Each artifact is formatted in different colors for easy differentiation between artifacts (e.g., Orange –Use Case, Red –Source Code, Green –Test Case). Each node represents an artifact and supports a browseable filtered view of an artifact by either clicking on each node or

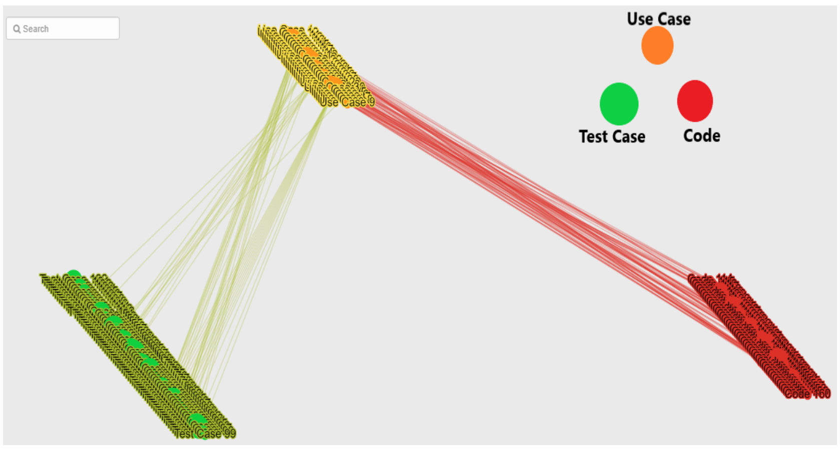


Figure 3.4: Unfiltered view of Hierarchical Trace Map

using free-text search features. The line represents the relationship between artifacts. A high-level trace map view can assist an analyst in various software engineering tasks, including system comprehension and change impact analysis. The similarity scores in the headmap diagram are scaled from dark to light colours, with the darker the hue the more closely the two issue titles resemble one another.

We positioned the source artifact on the top layer and its target artifacts on the lower layer to demonstrate a hierarchical layer view of the system. In this view, the line represents one-to-many relations between the source and target artifact. Therefore, if there is no line between the source and target artifact, it means missing links. In Figure 3.4, some of the last section and middle section of use case artifacts are missing lines in test cases. By using this diagram, we can visually comprehend that some use cases do not have corresponding test cases. To facilitate the interactive visual exploration of the system, we incorporated a node clicking feature and a free-text search feature that allows users to move from an unfiltered view to a filtered view. Free-text search tools allow you to conduct a file name or content search.

Figure 3.5 presents the associations between a source and two target artifacts. We draw the size of the nodes based on either the textual volume of the artifact or the number of source code functions. However, we are still finalizing on size measurement definition at this point. For change impact analysis, a developer can use this diagram to visually inspect the impact areas of source codes and test cases intuitively without cross-referencing multiple trace links results.

Figure 3.6 presents the filtered view between a target and source artifacts. For source code artifacts, we displayed the list of corresponding functions to assist in identifying change impact areas effectively.

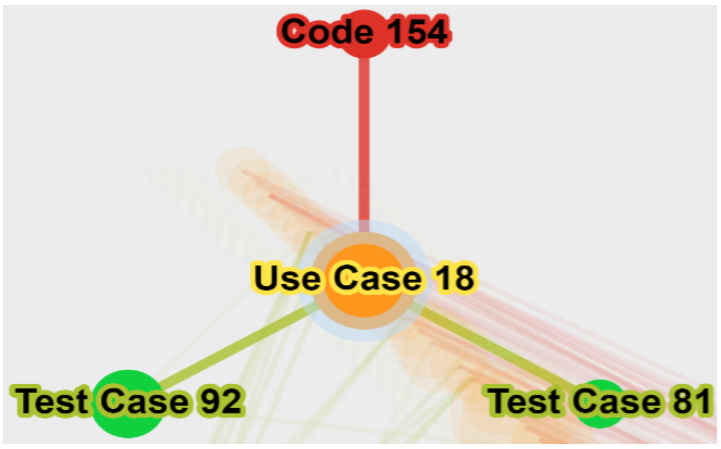


Figure 3.5: Filtered view of a source artifact and two target artifacts relation

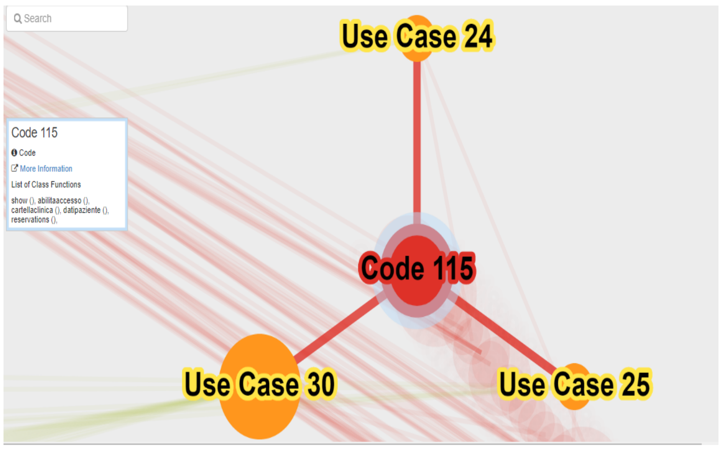


Figure 3.6: Filtered view of a target artifact and source artifacts relation

3.4 Related Work

This section introduces previous studies related to visualizing traceability associations between software artifacts. In addition, we discussed how previous studies address the challenge of integrating the traceability model into various software development projects.

3.4.1 Visualization

In recent years, several studies have addressed the challenges of visualizing trace links between various software artifacts using graphs & charts-based approaches. In [39], a hierarchical leaf node graphical structure is used to present a coarse-grain view of traceability links between software artifacts. A leaf node represents a requirement in a graph, while an internal node represents its associated title and other hierarchical information. The benefit of this graph is that the analyst can scan through the scope of the impact area in a bird-eye view. However, this technique might lead to a visual clutter issue when a large volume of data needs analysis.

The trace linkages between source code and documentation are similarly visualized in [35] using a combination of Treemap and hierarchical tree view graphs. The space-filling method, in which the child nodes are positioned along the parent boundaries and each sibling group is encircled by a margin, was adopted by the TreeMap graph to illustrate the trace linkages. In this combined approach, Treemap is used to convey the high-level structure, whereas a hierarchical tree view is used to present the fine-grain view of the links between source code and sections of the feature or bug report. Therefore, it is helpful to pinpoint the impact area of the source code in detail. Although their approach addresses the challenges of visualizing high and low-level trace information in one workspace, it can only visualize two-dimensional trace link information. In [105], the authors proposed Sunburst and Netmap visualization mechanisms to traceability links in the requirements engineering knowledge domain. In Sunburst, the relationships between artifacts are depicted in the form of the radical layout where nodes are presented in the adjacent rings. In Netmap, nodes are arranged in a circular structure with one layer ring where the trace links are drawn in the inner circle and categorized with different colouring. Both methods can be used to display a coarse-grained view of traceability relationships, but when big data sets are presented, the layouts become congested.

In terms of evaluation, the majoring of the previous studies conjured graph-based visualization techniques in standalone workbench system [71, 121] where the user needs to import trace data to visualize the linkage between artifacts. In [120], the authors proposed the trace links visualization process model to integrate with the requirement management system to identify the links between high and low-level requirements. Likewise, in [35], the researchers presented the benefits of visualizing trace links in an integrated development environment, where the linkages between documentation and source code can visualize dynamically. Although the current graph-based visualization techniques are useful in identifying the relationships between artifacts, little textual information is available. Therefore, it cannot provide narrative accounts of linkages between multiple artifacts (e.g.., trace links between requirements, test cases, and source code) in one workplace. Therefore, it is time-consuming to identify the overall impact areas to estimate the development efforts. To address the challenge of interacting with multiple software artifacts, we proposed the hierarchical trace map visualization approach to explore the system's overall structure in the same workbench where it can cross-reference and connect multiple trace link results interactively.

3.4.2 Traceability model

Recent progress in the study of requirements traceability models in a broad range of software development projects is presented in this section. The model can generally be grouped into two categories: generic and domain-specific models.

In the work of [67, 68], the authors presented the generic traceability model to transform the system modeling language (SysML) requirements diagram into text-based formats by using a trace generation algorithm to enable traceability between stakeholders'requirements and various subsystem components. The model is evaluated with a vehicle anti-lock braking system. Similarly, in [50], the authors proposed the model to recover the links between subsystems and components from interdisciplinary fields such as mechanics, electronics, and Information Technology (IT). The model recovered the links between five main artifacts of the system, namely, products, functions, product structures (modules and components), requirements, and organizational structure. Likewise, [93] presented a trace deviation algorithm based on the traceability reference model of [132]. In their approach, three types of trace links, namely evolution link, dependency link, and satisfaction link, are formulated to trace links between high-level and low-level artifacts. Evolution and satisfaction links are used for forward and backward tracing, whereas dependency links are used to trace between the same level artifacts.

Also, the study of [51] presented a traceability framework called TORUS (Traceability of Requirements using splices). TORUS is a domain-specific model where the framework is intended for recovering trace links in cyber-physical systems (e.g., aerospace, automotive, healthcare, and transportation). The framework involves tracing distributed software components and processes to manage key project activities such as change impact analysis and requirements coverage. In [151], the study conducted the traceability experiment in the car sharing system project where it requires managing various artifacts including requirements artifacts, specification artifacts, management artifacts, solution artifacts, development artifacts, and stakeholders'artifacts.

In general, the traceability process involves creating and recovering the links and accessing the quality of the links. Quality assessment and validation activities are often missing in the traceability reference model as the automatically generated trace links are vulnerable to bias. Thus, the authors proposed the quality assessment model to systematically verify the quality of requirement traceability data [135]. In [135], the authors define traceability quality as “the degree to which existing artifacts of a software development project are traceable as mandated by the project’s traceability stakeholders”. The two assessment quality criteria are used in their approach to identifying unfulfilled and incomplete trace links. First, trace information created by the trace planner does not conform to trace information mandated by stakeholders marked as an unfulfilled link. Second, trace data created by the trace creator does not conform to trace information provided by the trace planner flagged as an incomplete link.

Similarly, in [3], the authors proposed the two generic traceability templates called SRS trace item (SRSTI) and SDD trace item (SDDTI) to recover the links between software requirements specification (SRS) artifact and software design document (SDD) artifacts. By constructing the links using the reference templates, the approach identifies four types of traceability problems; 1) missed in SD, 2) logical discrepancy, 3) missed in SRS, and 4) extra in SDD. Thus, their template-based model addresses logical discrepancy errors between two artifacts, which is non-trivial in system verification.

3.5 Chapter Summary

We developed an interactive hierarchical trace map visualization method to examine the impact area across many software artifacts in a workspace. It can scale to a large volume of data, making it generic to reusable across different projects. We conjure our approach in a standalone web-based application, developed with modern C-sharp programming language. We have validated it in the application and evaluated it with the EasyClinic public dataset due to the availability of the most commonly used software artifacts (e.g., use case, test case, and source code). By visualizing the coarse-grained view and fine-grained view of linkages between diverse artifacts using a hierarchical trace map interactively, we analyze the effectiveness and practicality of our approach in this chapter.

CHAPTER 4

MULTI-TRIAGE : BUG TRIAGE BASED ON DEEP MULTI-TASK LEARNING

4.1 Overview

The bug triage procedure involves two crucial steps: assigning to developers and classifying issue types. Existing approaches tackle these two tasks separately, which is time-consuming due to the repetition of effort and negating the values of correlated information between tasks. In this chapter, we presented our multi-triage model that predicts both tasks simultaneously via multi-task learning (MTL). Multi-triage model is faster in training than two single-task learning models and more precise than (SVM + BOW) [12] and DeepTriage [102] models. A multi-triage model is constructed iteratively by removing unnecessary model parameters through ablation analysis. To balance a class label and alleviate the class-imbalanced problem, we developed the contextual data augmentation approach.

In past years, previous studies have addressed issue assignment and labeling problems using traditional statistical machine learning approaches, such as support vector machine (SVM) [12, 116, 158], and Naive Bayes [111]. Recently, deep-learning techniques have been successfully applied in various issue-related link recovery problems, including missing links between issue reports and commits [136], automated pull request classification [161], and detecting duplicate issue reports [157]. In [69, 92], convolutional neural networks (CNN) and word embedding feature engineering approaches are used in predicting potential developers and bug severity.

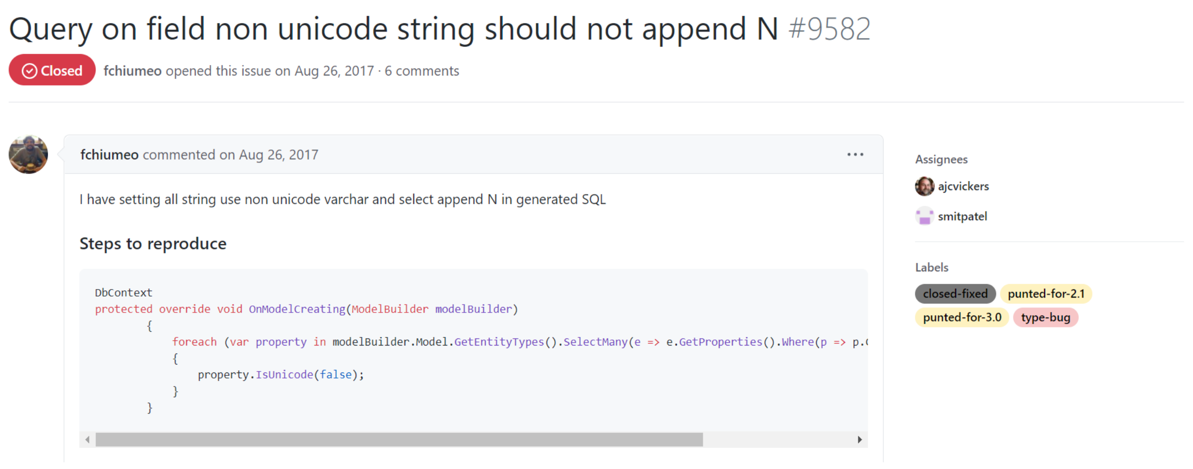
In these studies, bug triage tasks such as developer assignment and issue labeling are regarded as classification problems, and conclusions are drawn based on the context of issue reports. Therefore, these task classification model performances can be improved by jointly learning the representations of the issue report information in the multi-task learning model. In recent years, the multi-task learning approach has been primarily used in the computer vision field, which requires solving multiple tasks simultaneously [97, 139]. However, no prior work has experimented on learning numerous bug-triage tasks in a single model and conjured the effects.

Our key observation is whether bug report developers and bug types prediction models are likely to be trained together with jointed learning representation layers or not. As previously mentioned, prior works of [97, 139] presented the possibility of training multiple tasks in a single model in the computer vision field. To learn the issue report representation in a joint layer, our multi-triage model integrates the two prediction models into a single multitasking learning model. In previous studies, the code snippet is filtered out from the issue report to reduce the noise in learning representation.

To eliminate the basis of noise caused by code snippets, we designed our representation learning layer using two deep learning models and separately learning textual and AST representation. In contrast, we included the code snippet information in our model by transforming the abstract syntax tree (AST) token and conjuring the attribute's effects in the model training. Lastly, we applied the contextual data augmentation approach to generate syntactic bug reports addressing the class-imbalanced issue in the dataset. To sum up, the novelty of the multi-triage approach lies in exploiting the multi-task learning approach in the bug triage automation field and conjuring the effects of synthetic bug reports using the data augmentation approach. Finally, we evaluated our model on eleven open-source projects to demonstrate the effectiveness of this model compared with state-of-the-art methods.

4.2 Background

This section discusses background information about the correlation between developers and issue types recommendation tasks as well as their usages in the issue report and pull-based development projects. Then, we present our motivating example.



Query on field non unicode string should not append N #9582

Closed fchiumeo opened this issue on Aug 26, 2017 · 6 comments

fchiumeo commented on Aug 26, 2017

I have setting all string use non unicode varchar and select append N in generated SQL

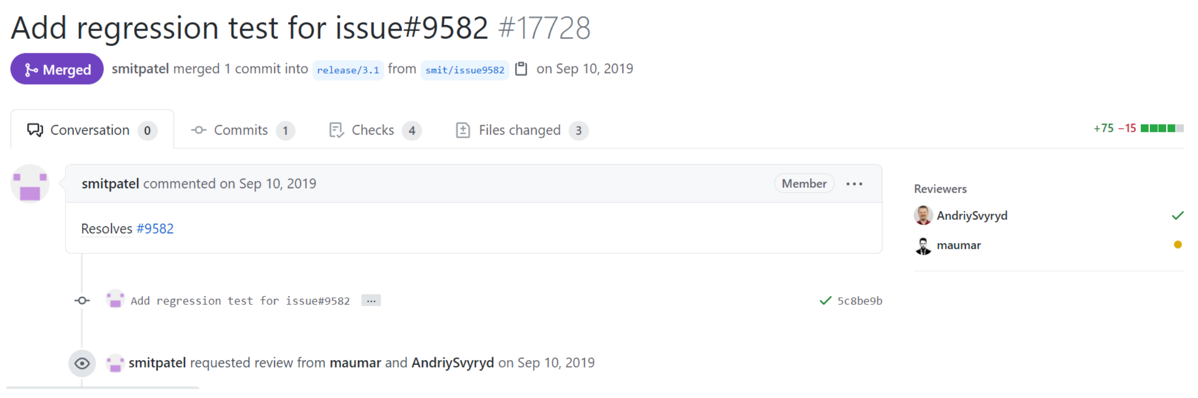
Steps to reproduce

```
DbContext
protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            foreach (var property in modelBuilder.Model.GetEntityTypes().SelectMany(e => e.GetProperties().Where(p => p.
            {
                property.IsUnicode(false);
            }
        }
```

Assignees
ajcvickers
smitpatel

Labels
closed-fixed punted-for-2.1 punted-for-3.0 type-bug

((a)) Issue report



Add regression test for issue#9582 #17728

Merged smitpatel merged 1 commit into `release/3.1` from `smit/issue9582` on Sep 10, 2019

Conversation 0 · Commits 1 · Checks 4 · Files changed 3 · +75 −15

smitpatel commented on Sep 10, 2019 · Member

Resolves #9582

`Add regression test for issue#9582` ✓ 5c8be9b

smitpatel requested review from maumar and AndriySvyryd on Sep 10, 2019

Reviewers
AndriySvyryd ✓
maumar ●

((b)) Pull request

Figure 4.1: An example of an issue report and the corresponding pull request

4.2.1 Developers and Issue Types Recommendation Tasks in Bug Triage

Assigning developers and allocating issue types are two essential tasks in the bug triage process. In the issue tracking system, an issue tracker normally performs these two tasks as the first step in the bug triage process. Our multi-triage recommendation model predicts relevant developers and issue types for a new issue report to leverage the bug triage process. In this context, issue reports include both bug and enhancement-related issues. Our recommendation model performs two tasks, as below.

Developer recommendation task This task involves predicting a list of potential developers to fix a new issue report. Sometimes, the issue report is fixed by more than one developer, due to its complexity.

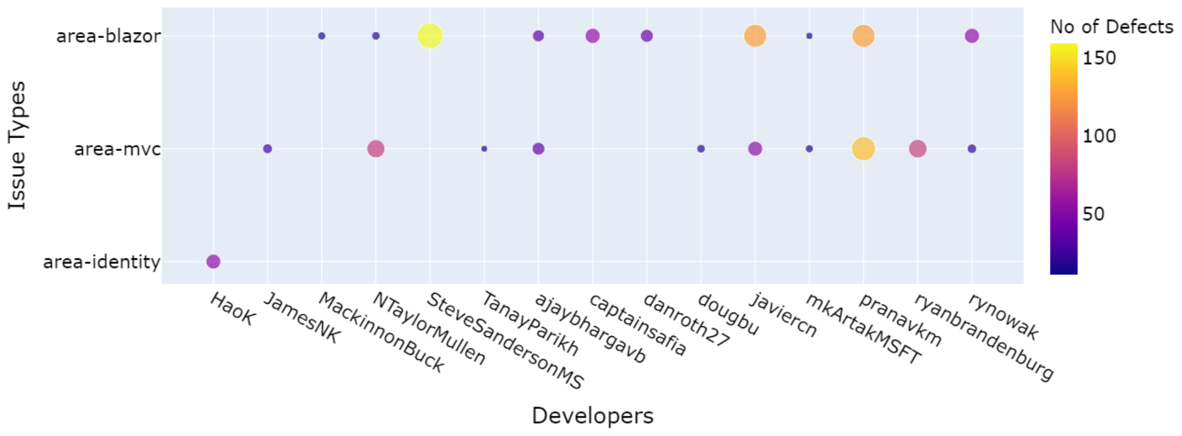


Figure 4.2: Developers and issue type correlation example

Issue type recommendation task This task involves predicting a list of issue types to categorise a new issue report. For example, GitHub's issue tracking system provides seven generic labels (i.e. bug, duplicate, enhancement, help wanted, invalid, question, and won't fix), but can add a new custom label as needed [30]. Interestingly, most projects create custom labels to track issue priority (e.g. high, low), product version (e.g. 2.1), workflow (e.g. backlog, review), and product components (e.g. area-identity, area-mvc, area-blazer).

Issue report and corresponding pull request Fig. 4.1 presents an example of the GitHub issue report 4.1(a) and its corresponding pull request 4.1(b). Recent years have seen a growing interest in pull-based development in open-source software projects [63, 76, 162]. In a pull-based model, a developer uses a pull request form to submit code for request code review. The reviewers are usually project owners or contributors who make the final decisions on the requested changes (i.e. reject, merge, or reopen). For the GitHub project, the fields contained in the pull request form are similar to those in the issue request form but also include additional sections, such as reviewers, a commits tab, a checks tab, and files changed tab. In the description field, most projects reference the fixed issue IDs for traceability. The reviewer's field contains the list of reviewers who review the changes, while the commits tab contains the commits hierarchy, and the checks tab presents the detailed build outputs. Last but not least, the files changed tab shows a list of the files that have been modified across all commits. During initial observations, it was learned that the developer allocated to the issue report may be different from the developer who created the pull request to fix the issue. Therefore, this study considered that the developer information from the pull request is non-trivial in the label construction process.

Developer and issue type correlation In existing projects, both developers and issue types recommendation tasks use historical issue reports training the prediction model. Therefore, there is a common learning representation layer between these two tasks, which can be learned together. Also, as a software project involves various components (e.g. user interface, database, application programming interface), an issue report can relate to any part of the system. Consequently, certain issue types are usually assigned to a group of developers with expertise in certain system areas. The recent work of [33] highlighted that not all bugs are the same, and the structure of project teams is based on the components of a system. Fig. 4.2 presents a simple example in which developers focus on fixing particular system areas. This example was extracted from aspnetcore[1].

4.2.2 Multi-task learning

MTL has been successfully used in numerous fields recently, including computer vision [26, 52, 86, 166], and natural language processing [96]. Computer vision involves a host of tasks, such as image classification, object detection, semantic segmentation, instance segmentation, and salience estimation. Successful joint learning of these different visual task studies shows that the MTL approach can optimize the training strategy and improve performance without compromising accuracy. Similarly, recent studies in natural language processing demonstrate significant gains from using the MLT approach in training query classification and ranking tasks. Motivated by the success of multi-task learning in various fields, the MTL model is used in this paper to enhance the efficiency of the bug triage procedure. MTL tackles developer and issue type recommendation tasks simultaneously by sharing learning parameters to enable these tasks to interact with each other. Joint learning of these two tasks significantly improves the performance of each task, compared to learning independently. The multi-task learning model can share parameters between multiple tasks with either hard or soft parameter sharing of hidden layers. The hard parameter sharing model explicitly shares the common learning layers

[1] `https://github.com/dotnet/aspnetcore` the GitHub project. The x-axis represents the developers, whereas the y-axis represents the system areas. The size of the bubble indicates the total issues fixed by developers in the corresponding areas. Referring to the example, a handful of potential developers can fix area-blaze and areas-mvc issues. However, there is one developer (Haok) who is capable of resolving area-identity issues. Based on this observation, we are motivated to seek the effect of the learning developer and issue types jointly in the multi-task learning model

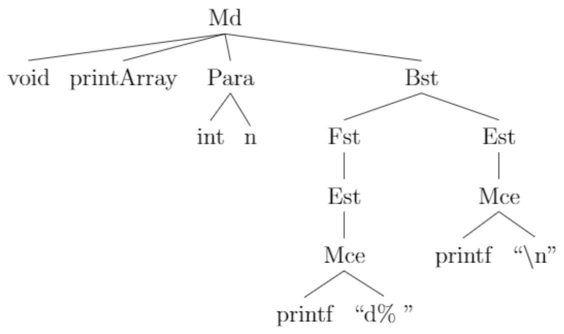
between all tasks while branching the task-specific output layer [32]. The soft parameter sharing model, meanwhile, implicitly shares the parameters by regularizing the distance between the parameters of each task. Although both approaches can be viewed as the underlying architecture of the multi-task learning model, hard-parameter sharing is commonly applied in the context of the neural network.

This multitask learning model uses a hard-parameter sharing approach to learn the issue report representation in the common layer and then branches the two task-specific output layers to predict developers and issue types. In the common layer, the individual issue report is further subdivided into two categories, namely 1) natural language and 2) structural language, to learn the representation effectively. An issue title and description, excluding code snippets, are grouped under natural language, whereas code snippets are placed under structural language. Then, two encoders are used, namely 1) context encoder and 2) AST encoder, to extract the essential features of these two contexts. Next, these two features are combined and fed into the task-specific output layers to perform co-responding classification tasks. The detailed implementation of this approach is explained in Section 4.4.

4.3 A motivating example

```
//Utilizing function to print the Array
void printArray(int arr[], int n)
{
  for(int i = 0; i < n; i++)
    printf("%d  ", arr[i]);
  printf("\n");
}
```

((a)) Code snippet



((b)) AST

Figure 4.3: A code snippet and corresponding AST example

As mentioned earlier, the previous bug triage approach has considered developers and issue types prediction tasks as independent tasks and trained separately for each. Therefore, it is time-consuming to train the model. In addition, in existing approaches, code snippets are either excluded [102, 152] to reduce noise, or treated as natural language sequence tokens [92, 155]. Thus, these approaches cannot learn the code snippets

Table 4.1: AST nodes terms and abbreviation

Terms	Abbreviation
Method Declaration	Md
Parameter	Para
Block statement	Bst
For statement	Fst
Expression statement	Est
Method call expression	Mce

or representations precisely. In initial observations, the issue report characteristics of eleven open-source projects from various domains were investigated, including a web application, unit testing, entity development, programming interface, compiler, mobile app, augmented reality, gaming, and search engine configuration. Further information is presented in Table 4.2.

These GitHub projects were chosen based on their level of activity and popularity (i.e. ranking and recent commits). Projects with a high number of contributors and issue types labels were also considered to identify the gap in the existing approaches to leverage the bug triage process. Eclipse issue reports, which are used in baseline studies to compare this study's approach and the state-of-the-art approach, were also included. Eclipse issue reports were extracted from the Bugzilla issue management system. However, Bugzilla[2] does not keep developers,Äô tossing sequences as this study did not present the average tossing sequence, value for the eclipse project in Table 4.2.

Code snippet The percentage of issue reports including method-level code snippets was analyzed to reproduce the problem. Interestingly, as presented in Table 4.2, 12 to 20 percent of issue reports contain code snippets. Recent studies [9, 82] have found that learning representations of AST tokens are more effective than simple code-based tokens in various code prediction tasks (i.e. code translation, code captioning, code documentation). These code snippets are converted into AST routes following this method, which was inspired by earlier investigations, and a separate token is produced. Fig. 4.3 shows an example of a java code snippet 4.3(a) and its corresponding AST 4.3(b), where a node (i.e. Para, Bst, Fst, Est, and Mce) is non-terminal node, and the rest are terminal nodes. In this approach, the code snippets are complied using Eclipse IDE[3] for java code snippets and Microsoft visual studio IDE[4] for C# code snippets. Table 4.1 presents the abbreviation for each of the AST node terms. Then, the code snippets are parsed into AST

[2]https://bugs.eclipse.org/bugs/
[3]https://projects.eclipse.org/projects/eclipse.platform/
[4]https://visualstudio.microsoft.com/

Table 4.2: Raw datasets information

Name	Period	#No	#Code	#Dev	#Types	#Tossing	#Days
aspnetcore	10/2014 - 10/2020	7151	2520	60	131	62	45
azure-powershell	01/2015 09/2020	2540	312	386	204	128	82
eclipse	10/2001 05/2021	50806	6320	21	621	-	60
efcore	01/2015 - 09/2020	6612	1650	24	57	2293	76
elasticsearch	01/2015 - 10/2020	5190	1504	104	238	178	92
mixedreality toolkit-unity	03/2016 - 09/2020	2294	70	55	124	53	71
monogame	01/2015 - 09/2020	1008	110	4	28	22	21
nunit	10/2013 - 09/2020	656	70	27	24	130	63
realm-java	05/2012 -10/2020	1160	340	15	23	400	69
roslyn	02/2015 - -09/2020	5093	1300	79	123	300	100
rxjava	01/2013 - -09/2020	2076	610	5	32	121	44
						Avg (368)	**Avg (67)**

using Java and C# extractor from the code to sequence the representation approach [9]. Implementation details are presented in Section 4.5.

Issue reassignment In the GitHub project, it is noted that a single pull request can include fixes for multiple issue reports, and a developer who fixes the issue may be different from the assigned developers recorded in these issue reports. In the context of bug triage, this process is normally referred to as tossing [152]. The label construction procedure includes the information about these pull request developers to incorporate the issue report tossing sequence.

4.4 Multi-triage

This section first explains the high-level structure of the multi-triage framework. Next, it presents the integral components of the multi-triage model.

4.4.1 General

Fig. 4.4 presents the overall structure of the multi-triage framework. This framework includes three main components: (1) data extraction, (2) a contextual data augmenter,

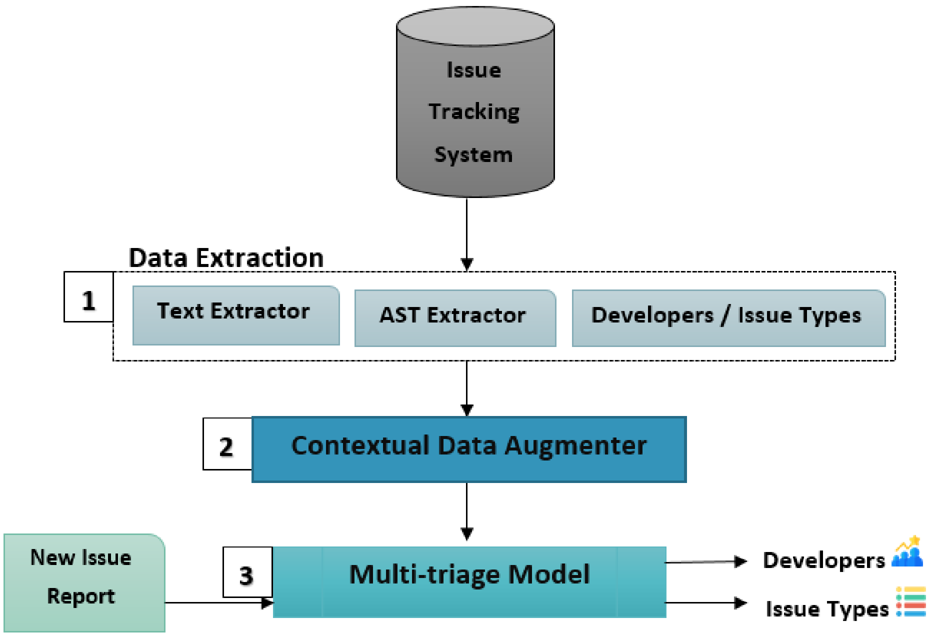
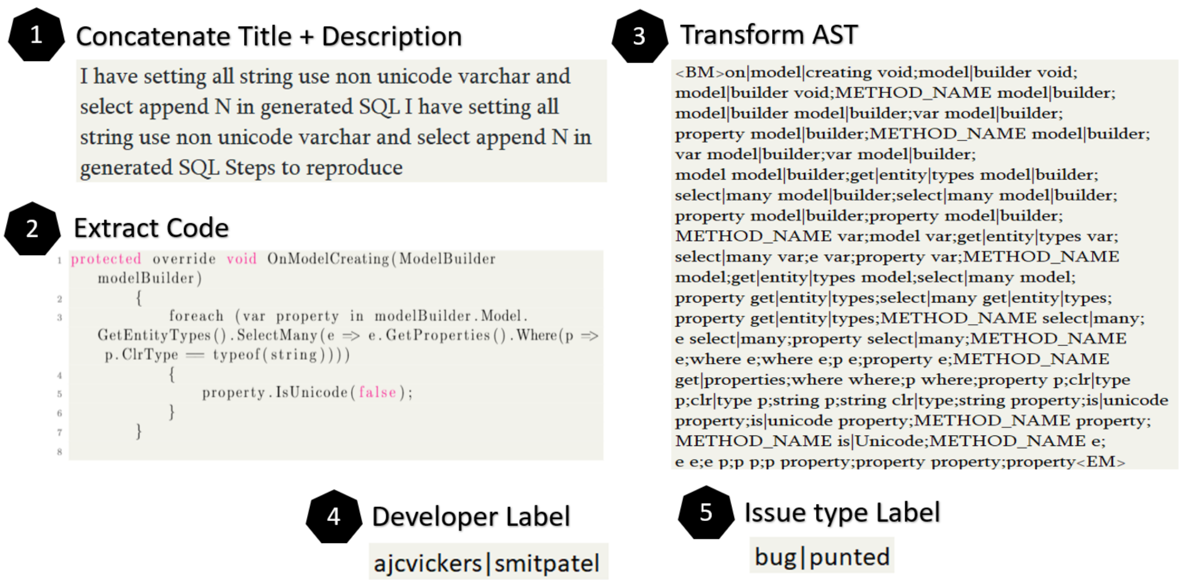


Figure 4.4: The multi-triage framework

and (3) the multi-triage model. In the data extraction component, ground truth links are constructed between issue reports and multi-labels (i.e. developers and issue types).

4.4.2 Data Extraction



I have setting all string use non unicode varchar and select append N in generated SQL I have setting all string use non unicode varchar and select append N in generated SQL Steps to reproduce

```
protected override void OnModelCreating(ModelBuilder
   modelBuilder)
        {
            foreach (var property in modelBuilder.Model.
   GetEntityTypes().SelectMany(e => e.GetProperties().Where(p =>
    p.ClrType == typeof(string))))
            {
                property.IsUnicode(false);
            }
        }
```

```
<BM>on|model|creating void;model|builder void;
model|builder void;METHOD_NAME model|builder;
model|builder model|builder;var model|builder;
property model|builder;METHOD_NAME model|builder;
var model|builder;var model|builder;
model model|builder;get|entity|types model|builder;
select|many model|builder;select|many model|builder;
property model|builder;property model|builder;
METHOD_NAME var;model var;get|entity|types var;
select|many var;e var;property var;METHOD_NAME
model;get|entity|types model;select|many model;
property get|entity|types;select|many get|entity|types;
property get|entity|types;METHOD_NAME select|many;
e select|many;property select|many;METHOD_NAME
e;where e;where e;p e;property e;METHOD_NAME
get|properties;where where;p where;property p;clr|type
p;clr|type p;string p;string clr|type;string property;is|unicode
property;is|unicode property;METHOD_NAME property;
METHOD_NAME is|Unicode;METHOD_NAME e;
e e;e p;p p;p property;property property;property<EM>
```

ajcvickers|smitpatel

bug|punted

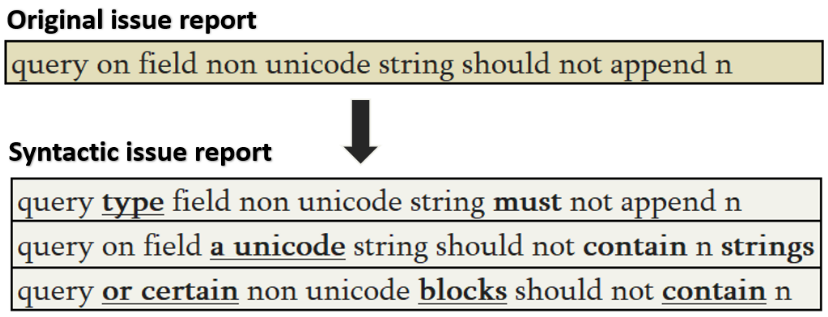
Figure 4.5: An example of data extraction steps for an issue report

The data extraction component includes two sub-components: the text extractor and the AST extractor. The *text extractor* component concatenates each issue report's title and description into one text token, excluding the code snippet information. The *AST extractor* parses each code snippet and constructs the AST paths. An AST or syntax tree has two types of nodes: terminal and non-terminal. The terminal node represents user-defined values (e.g. identifiers), whereas the non-terminal node represents syntactic structures (e.g. variable declarations, a for loop) [9]. The AST path is the sequence of the terminal and non-terminal nodes.

In this paper, Eclipse, and Microsoft visual studio IDE was used to compile the code snippet before passing it to the AST extractor. The AST generator tool from [9] is used to construct AST paths, using the default parameters settings (max child node = 10, max path length = 1000, and max code length = 1000). In any issue report, a single code snippet can contain multiple methods as the generator is modified [9] by adding '$\langle BM \rangle$' and '$\langle EM \rangle$' separator tags between each method for model learning purposes.

Fig. 4.5 presents the data extraction steps for a single issue report seen in Fig. 4.1(a). First, the issue report's title and description are concatenated. Next, the code snippet is compiled and parsed into AST paths. The AST paths are generated by pairing all the dependent nodes and using the ';' separator between a pair to indicate a path. Next, multiple developer labels are created by using the '|' separator. In the developer labelling process, a pull request creator account is included if the developers allocated in the issue report do not include a pull request creator account. Finally, the issue type label is constructed by using bug or enhancement and system components format using the same '|' separator.

4.4.3 Contextual Data Augmenter


Original issue report

query on field non unicode string should not append n

Syntactic issue report

query **type** field non unicode string **must** not append n

query on field **a unicode** string should not **contain** n **strings**

query **or certain** non unicode **blocks** should not **contain** n

Figure 4.6: A synthetic issue report example

Synthetic issue reports are generated for each project using the method described

Algorithm 1: An algorithm with which to generate a synthetic issue report with the contextual data augmentation approach

```
   input  : list of training issue reports $TB$, augmentation threshold $Threshold$
   output : list of synthetic issue reports $TS$
1  $MajC \leftarrow$ a majority class samples count;
2  $MinC \leftarrow$ total no of minority classes;
3  $MinClist \leftarrow$ list of minority classes;
4  $EstimateDataAugAmount \leftarrow MinC * MajC$;
5  if $EstimateDataAugAmount < Threshold$ then
6      $MajC \leftarrow (Threshold / EstimateDataAugAmount) * MajC$
7  end
8  for $minclass \in MinClist$ do
9      $BC \leftarrow$ retrieve total no of issue reports fixed by $minclass$ from $TB$;
10     while $BC < MajC$ do
11         $RC \leftarrow$ retrieve one random record of $minclass$ from $TB$;
12         $NC \leftarrow$ generate a new synthetic record based on $RC$ with contextual data augmentation approach;
13         Append $NC$ to $TS$;
14         $BC \leftarrow BC + 1$;
15     end
16 end
18 return $TS$
```

in algorithm 1 for contextual data augmentation. The algorithm's input is the list of training issue reports and the *Threshold* to generate synthetic records. In this approach, a new record is created based on the training datasets, and the generation of synthetic records is limited by using the *Threshold* parameter. In this experiment, a *Threshold* value of 30,000 is used to control the total number of data augmentation records. The Threshold is calculated based on the approximate total number of issue reports from target projects. However, it is a hyperparameter value and can change as needed. First, it initializes the values with the majority and minority class details (lines 1 to 3). It creates clusters by grouping developer and issue type labels. After initialisation, *MinC* ∗ *MajC* is multiplied to calculate the estimated number of synthetic records to compare with the *Threshold* amount (line 4). If the estimated value is larger than the *Threshold*, then it calculates the new majority class count value for an adjustment (lines 5 to 7). Next, it iterates through each minority class to generate a synthetic record (lines 8 to 16).

In each iteration, it randomly retrieves an issue report description (excluding code snippets) of the current minority class. Then, it substitutes 15% of the words in the description with the new words using the contextual data augmentation approach proposed by [77] and creates a new issue report. In this experiment, the BERT-base-uncased pre-trained model [5] was used, which was trained with a large corpus of English data to predict the substitute words. However, this approach can be generalized to other

[5] `https://huggingface.co/bert-base-uncased/`

pre-trained models as well. Lastly, the output of the algorithm is the training datasets, including syntactic records.

Fig. 4.6 presents an example of synthetic issue reports generated with the contextual data augmenter via comparison with the original issue report. As shown in Fig. 4.6, all the syntactic context generated by the data augmenter is underlined. In general, the data augmenter generates synthetic reports by substituting the main keywords from original issue reports while maintaining the original context. In the next section, the final component of the framework, the multi-triage model, is explained in detail.

4.4.4 Multi-Triage Model

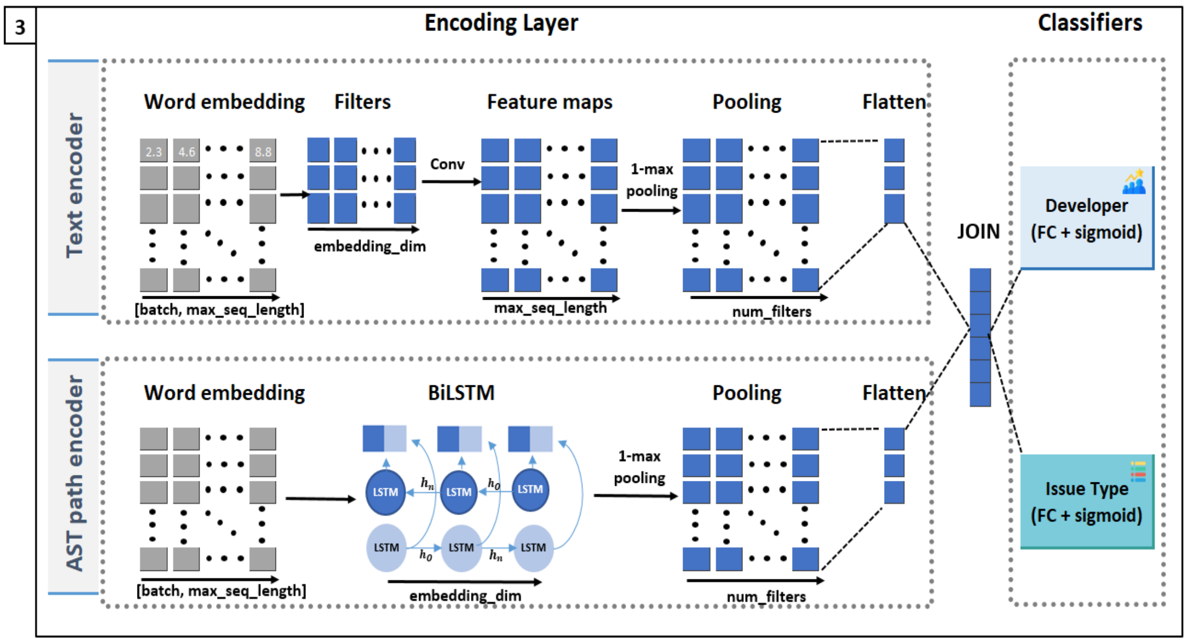


Figure 4.7: The multi-triage model

As shown in Fig. 4.7, the multi-triage model has three main components: the context encoder, the AST encoder, and classifiers. The two encoders are used to generate the natural language and structural (code) representation based on the input issue reports. The shared layer between the encoders concatenates the outputs of the encoders to construct the overall feature representations of issue reports. Finally, the classifiers analyze these feature representations and recommend the potential developers and issue types as outputs. The main hyper-parameters of the model are $batch = 32$, $max\_seq\_length = 300$, $embedding\_dim = 100$, and $num\_filters = 100$. The batch size can be set between

1 and a few hundred; however, a standard batch size (32) was selected to train this model [21].

4.4.5 Code Representation

Context encoder extracting representation features from issue reports is non-trivial in the bug-triage process. In this model, a context encoder is used to extract the natural language representations of the issue report. Convolutional neural networks (CNN) are used to generate these representations. In recent years, CNN has been successfully applied in various modelling tasks, including textural classification [18, 83, 92] and image classification [91, 112]). The input of this encoder is the concatenated values of the issue title and description. The raw input is normalized by removing stop words, stemming, lower-casing, and padding equally to the right with the max_seq_length range. First, each issue report is transformed into a vector by turning each issue report into a sequence of integers (each integer value being the index of a token in a dictionary). Second, these inputs are fed into a word embedding layer with input dimension ($vocab\_size$ + 1). A dynamic $vocab\_size$ value equal to the size of the vocabulary of each project is used. The next layer filters are the core of CNN's architecture. 1D convolution is applied via $filters$. The standard kernel size of 4×4 is used to extract the important features [40]. Then, the max-over-time pooling operation is applied to extract the most relevant information from each feature map. The output from the pooling process is then sent to the joining layer for concatenation.

For example, given an issue report with n words $[b_1, b_2, \ldots, b_n]$, the word vectors corresponding to each word are presented as $[x_1, x_2, \ldots, x_n]$ (i.e. x_i is the word vector representation of word b_i). Let $x_i \epsilon \mathbb{R}$ be k-dimensional (k=1). The inputs of a convolution layer are the concatenation of each word vector, represented as:

$$x_{1:n} = x_1 \oplus x_2 \oplus \cdots \oplus x_n, \tag{4.1}$$

where $\oplus$ denotes the concatenation operator. In a convolution layer, a filter $w \; \epsilon \; \mathbb{R}$, slides across inputs by applying a window of h=4 (words) to capture the relevant features. In general, a feature c_i is processed by sliding a window of words $x_{i:i+h-1}$ by

$$c_i = f(w \cdot x_{i:i+h-1} + b), \tag{4.2}$$

where b denotes bias and f is a non-linear function (i.e. the hyperbolic tangent function)

$$c = [c_1, c_2, \ldots, c_{n-h+1}]. \tag{4.3}$$

Next, a max-over-time pooling operation [4] is applied to extract the maximum value $\hat{c} = max\{c\}$ to capture the most important feature for each feature map. In general, one feature is extracted from one filter. In this model, 100 filters are used to obtain multiple features from the issue report. Next, the output is flattened to one dimension and fed into the joining layer.

AST encoder In this approach, each code snippet in an issue report is parsed to construct an AST path using the AST extractor and which is used as input to the AST encoder. In the pre-processing phase, all inputs are first prepared to the same size by padding equally to the right with the max_seq_length range. Second, an AST path is transformed into a vector by turning each word into a sequence of integers. Next, these inputs are fed into the word-embedding layer with input dimension ($vocab\_size$ + 1).

To learn AST representations, bidirectional recurrent neural networks with long short-term memory (BiLSTM) neurons [31, 64] are used. In general, BiLSTM models combine two separate LSTM layers which operate in opposite directions (i.e. forward and backward) to utilize information from both preceding and succeeding states. In LSTM networks, each memory cell c contains three gates: input gate i, forget gate f, and output gate o. Formally, an input AST sequence vector $[a_1, a_2, \ldots, a_n]$ is given, where n denotes the length of the sequence. The input gate i controls how much of the input a_t is saved to the current cell state c_t. Next, the forget gate f controls how much of the previous cell state c_{t-1} is retrained in the current cell state c_t. Lastly, the output gate controls how much of the current cell state c_t is submitted to the current output h_t. The formal representation of the LSTM network is as follows:

$$\begin{aligned}
i_t &= \sigma(W_{ia}a_t + W_{ih}h_{t-1} + b_i),\\
f_t &= \sigma(W_{fa}a_t + W_{fh}h_{t-1} + b_f),\\
o_t &= \sigma(W_{oa}a_t + W_{oh}h_{t-1} + b_o),\\
c_t &= f_t * c_{t-1} + i_t * \tanh.(W_{ca}a_t + W_{ch}h_{t-1} + b_c),\\
h_t &= o_t * \tanh(c_t).
\end{aligned} \tag{4.4}$$

In Eq. 4.4, a_t indicates the input word vector of the AST path, h_t indicates the hidden state, W indicates the weight matrix, b indicates the bias vector, and σ indicates the logistic sigmoid function. A BiLSTM network calculates the input AST sequence vector a in a forward direction sequence $\overrightarrow{h_t} = [\overrightarrow{h_1}, \overrightarrow{h_2}, \ldots, \overrightarrow{h_n}]$ and a backward direction sequence $\overleftarrow{h_t} = [\overleftarrow{h_1}, \overleftarrow{h_2}, \ldots, \overleftarrow{h_n}]$, then concatenates the outputs $y_t = [\overrightarrow{h_t}, \overleftarrow{h_t}]$. The formal representation of the BiLSTM network is as follows:

$$\begin{aligned}\overrightarrow{h}_t &= \sigma(W_{\overrightarrow{h}a} a_t + W_{\overrightarrow{h}} \overrightarrow{h}_{t-1} + b_{\overrightarrow{h}}),\\ \overleftarrow{h}_t &= \sigma(W_{\overleftarrow{h}a} a_t + W_{\overleftarrow{h}} \overleftarrow{h}_{t+1} + b_{\overleftarrow{h}}),\\ y_t &= W_{y\overrightarrow{h}} \overrightarrow{h}_t + W_{y\overleftarrow{h}} \overleftarrow{h}_t + b_y\end{aligned} \tag{4.5}$$

In Eq. 4.5, y_t is the output sequence of the hidden layer h_t at a time step t. Next, a max-over-time pooling operation [4] is applied over BiLSTM outputs to extract the important information. Finally, the output is flattened and fed into the joining layer. In the joining layer, the two encoders are concatenated, output, and fed into the classification layer.

4.4.6 Task-Specific Classifiers

The sigmoid function was used to classify the relevant developers and issue types for a new issue report. As illustrated in Fig. 4.7, both developer and issue type classifiers share the same structure but differ in their input labels (i.e. developer and issue type). Therefore, only illustrate one classification layer is illustrated in this section. The classification layer is composed of two layers: a fully-connected FFN with ReLUs as well as a sigmoid layer.

Label classifier In the FFN layer, the ReLU is an activation function that outputs the input directly if the input is positive; otherwise, it will output zero [114]. In Eq. 4.6, x denotes the concatenated embedding vector with 150 dimensions, W denotes weights, and b denotes bias. Next, the output vectors are fed into the sigmoid layer to predict the appropriate developers or issue types for the input issue report.

$$\mathrm{FFN}(x) = \max(0, xW_i + b_i). \tag{4.6}$$

The sigmoid exponential activation function is then used to calculate the probability distribution of the output vectors from the FFN layer for each possible class (i.e. developers or issue types):

$$\mathrm{P}(c_j|x_i) = \frac{1}{1+\exp(-z_j)}. \tag{4.7}$$

Eq. 4.7 presents the formal representation of the sigmoid activation function at the final neural network layer to calculate the probability of a class c_j, where x_i is an input issue report and z_j is the output of the FFN layer.

4.5 Evaluation

In this section, the research questions and detailed information on the experimental implementation are presented. The code, data, and trained models are available at [17].

Datasets The issue reports of ten GitHub projects were collected as described in Table 4.2. In addition, eclipse issue reports were also collected effectively to compare the present approach against baseline studies. Following previous studies, only retrieve the issue reports with ‘closed’ status [12, 92, 102, 152] are retrieved. The issue reports with unassigned developers or issue types are also removed, as the model cannot be trained and validated with unlabeled records. Furthermore, issue reports assigned to ‘software bots’, which are frequently used in automatic issue assignment processes [61], are excluded. As no actual developer is used, these reports are not applicable to users in the developer prediction process. Statistics of datasets such as labels (i.e. developers, issue types) and code snippets are presented in Table 4.2. In terms of issue report metadata, an issue report title, description, creation date, assignee, and labels are presented, as well as the corresponding pull request’s assignee information, to create a tossing sequence.

Single task learning model The two single-task learning models shown, below are constructed to evaluate the effectiveness of this multi-task learning model.

- **BiLSTM-based triage model -** Two single-task BiLSTM networks are constructed: one for the developers’ prediction task and the other one for the issue types prediction task. In these models, architecture similar to the multi-triage model is replicated and used to create the two-word embedding layers to contract textual information and AST paths embedding tokens. Next, these two embedding tokens are concatenated and fed into the BiLSTM network to learn the issue report’s representation. Finally, these learned vectors are passed onto the classifier to predict labels (i.e. developers or issue types).
- **CNN-based triage model -** Similar to the BiLSTM model, the two single-task networks are constructed using CNN networks to learn the representations of issue reports.

As noted in Section 4.4, the multi-triage model combines BiLSTM and CNN networks to learn the representations of issue reports. Therefore, single networks are built using these two networks to effectively compare the time and accuracy trade-offs of the model.

Baselines The below two baseline approaches were used to evaluate the effective of the present approach.

- SVM+BOW [12]: This uses a Tf-IDF weighting matrix to transform textual features of issue reports into vector representations, and applies a support vector machine (SVM) machine learning classifier to automate the bug triage process.
- DeepTriage [102] - This uses a recurrent neural network (RNN) to learn the representations of issue reports and a softmax layer to recommend the potential developers and issue types as outputs.

Both of these approaches focus on predicting labels for a new issue report by learning the representation of existing issue reports. The first approach uses a support vector machine, whereas the second utilizes a recurrent neural network to automate the bug triage process. As the present approach uses BiLSTM and CNN to learn the representations of issue reports, these approaches have been selected for evaluation. For SVM+BOW, scikit-learn libraries are used to set up SVM+BOW because the source code is not accessible. In addition, scikit-learn is widely used in various studies [50, 2] to set up machine learning algorithms, including SVM.

Ablation analysis Parameter analysis plays a crucial role in the supervised learning model since tuning a single parameter can affect the model's performance. Ablation analysis is a procedure investigating configuration paths to ascertain which model's parameters contribute most to optimizing model performance [25, 56]. An ablation analysis procedure is adopted to determine which components of the multi-triage model contribute most to leveraging model performance. In the ablation analysis approach, developers identify a set of candidate parameters, evaluate the training data by running with these parameters, and take the candidate parameter which outperforms at least one other configuration. In this study's ablation analysis experiments, encoder decoupling and parameter tuning are performed to determine which encoder and parameters contribute most to improving model performance.

Evaluation settings The time-series-based 5-fold cross-validation procedure is followed to split the training (train), development (dev), and test sets [22, 24, 75, 142, 155]. This is a commonly used validation approach to measure the generalisability of a learning model. Fig. 4.8 presents the validation approach used in the data evaluation process. In this approach, the dev set makes up 10 per cent of the train set, and the test set

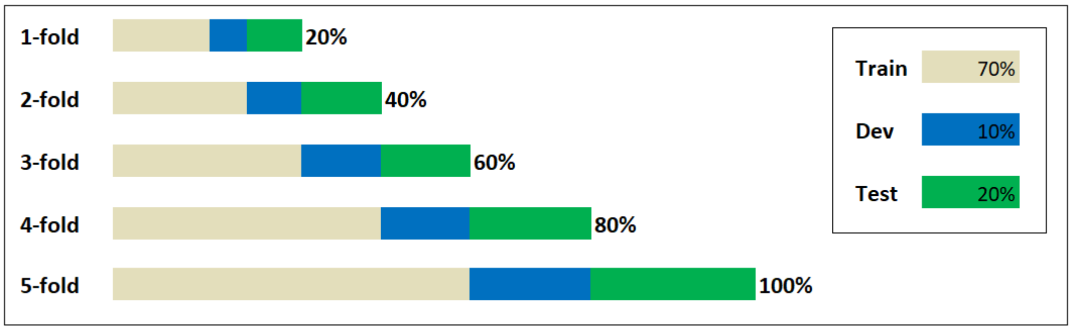


Figure 4.8: Time-series-based 5-fold cross-validation

assigns 20 per cent of the subset of the allocated data sample. The data set is folded on a rolling basis, based on the issue report creation date in ascending order.

All experiments are run in the google-colab[6] cloud-based platform on tesla v100-sxm2 GPU with 32 GB RAM. Python source code provided by the authors is used to set up the baseline models (i.e. SVM+BOW [12] and DeepTriage [102]). Also, the deep learning model is implemented using the TensorFlow Keras[7] deep learning library. In the multi-triage approach, both text input and AST path input are truncated to the length of 300. Each word is embedded into 100 dimensions. The output sizes of the text encoder and the AST encoder are 100 and 50, respectively. After joining the two encoder outputs, batch normalization is performed on the concatenated output and the drop (rate 0.5) is employed to reduce overfitting [2]. For the classifier, binary-cross-entropy and the Adam optimizer from the Keras library are used with a learning rate of 0.001. The model is tuned with different dimension sizes and learning rates, and the results are presented in Section 4.6. Finally, the vocabulary size is set based on individual project vocab size, and the default batch size (32) is used to train the model.

Evaluation metrics In these experiments, F-scores are used to measure the model's accuracy. In the following equations, TP denotes true positives, TN denotes true negatives, FP denotes false positives, and FN denotes false negatives.

- Precision - This is the ratio of the predicted correct labels to the total number of actual labels averaged over all instances. Eq. 4.8 presents the precision formula:

$$\text{Precision} = \frac{TP}{TP+FP} \tag{4.8}$$

[6]https://github.com/dotnet/aspnetcore/
[7]https://www.tensorflow.org/

- Recall - This is the ratio of the predicted correct labels to the total number of predicted labels averaged over all instances. Eq. 4.9 presents the recalled formula:

$$\text{Recall} = \frac{TP}{TP+FN} \tag{4.9}$$

- F-scores - This is a commonly used metric for the bug triage process. It is calculated from the precision and recall scores. The F1 score is calculated by assigning equal weights to precision and recall, while the F2 score adds more weight to recall. Even though both precision and recall are important, the F2 score is usually preferred in bug triage studies, where measuring the recall is more non-trivial than precision. Eq. 4.10 presents the F2 score formula:

$$\text{F}_\beta = \frac{(1+\beta^2)\times precision \times recall}{\beta^2 \times precision \times recall} \tag{4.10}$$

- Accuracy —Calculated by the average across all instances, where the accuracy of each instance is the ratio of the predicted correct labels to the total number of (predicted and actual) labels for that instance. Eq. 4.11 presents the accuracy formula:

$$\text{Accuracy} = \frac{TN+TP}{TN+TP+FN+FP} \tag{4.11}$$

4.6 Results

In this section, evaluation results are presented for the three research questions.

4.6.1 RQ1: How does the multi-triage model compare to other approaches?

The performance of the multi-triage model is compared to that of (SVM + BOW) [12] and DeepTriage [102] in the eleven open-source projects. The comparison results are presented in Table 4.3. The time-series-based 5-fold validation is performed on all approaches, and the average accuracy is presented for both developers and issue types prediction results. Since the Deeptriage [102] source code is publicly available, its environment can be replicated. However, the source code of (SVM + BOW) [12] is not accessible, and thus it was manually implemented using sklearn[8] libraries. Both approaches filter out code snippets and stack trace as these features are excluded in these

[8]`https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html`

Table 4.3: Multi-triage v.s. baselines (Base1 - SVM + BOW [12], Base2 - DeepTriage [102]) Accuracy (%)

Project	Developer			Issue type		
	Base1	Base 2	Multi-triage	Base1	Base 2	Multi-triage
aspnetcore	58%	51%	63%	25%	27%	47%
azure-powershell	35%	39%	48%	24%	29%	44%
ecplise	31%	35%	54%	23%	24%	26%
efcore	52%	55%	59%	30%	34%	40%
elasticsearch	46%	53%	58%	13%	21%	31%
mixedreality toolkitunity	41%	50%	62%	30%	33%	47%
monogame	62%	65%	69%	53%	55%	57%
nunit	36%	38%	41%	19%	23%	27%
realmjava	59%	60%	62%	24%	25%	50%
roslyn	33%	35%	39%	22%	25%	27%
rxjava	64%	66%	68%	31%	40%	49%
AVG	47%	50%	57%	27%	31%	42%
MAX	64%	66%	69%	53%	55%	57%

models. Conversely, this approach generates a separate token for each code snippet by parsing it to AST paths and including it in the model,Äôs training.

As shown in Table 4.3, this approach outperforms (SVM + BOW) [12] and DeepTriage [102] by an average increase of 10 and 7 percentage points for developers, and 15 and 11 percentage points for issue types, respectively. At its highest, this approach achieves 69% and 57% for developers and issue types, respectively. It was observed that, in both prediction tasks, an accuracy lower than 40% on the projects (i.e. eclipse, elasticsearch, nunit, and Roslyn) has either the higher number of potential issue types or developers' labels, or low sample data compared to the rest of the projects. In summary, this approach achieves the best performance, with DeepTriage [102] second by comparison. In the following section, the qualitative analysis test is performed to determine how many bug and enhancement records were correctly predicted with this approach compared to the state-of-the-art approaches.

As in qualitative analytical evaluation, the sample data is sorted into two issue groups, namely 1) the bugs and 2) the enhancements, and the performance is examined in light of the forecasted outcomes. Table 4.4 presents the statistics of the prediction results in terms of numbers, whereas the Venn diagram in Fig. 4.9 illustrates the total numbers of bugs and enhancements found by base1, base2, and the present approach. Notably, the present approach can predict all issue types which are predicted correctly in base1 and base2. In addition, this approach predicts 546 bugs and 46 enhancement records missed by baseline approaches. After inspecting these records, it became clear that these reports provide trivial descriptive text with code snippets to reproduce the issue. Previous studies neglected the code snippets in their approach, as the feature

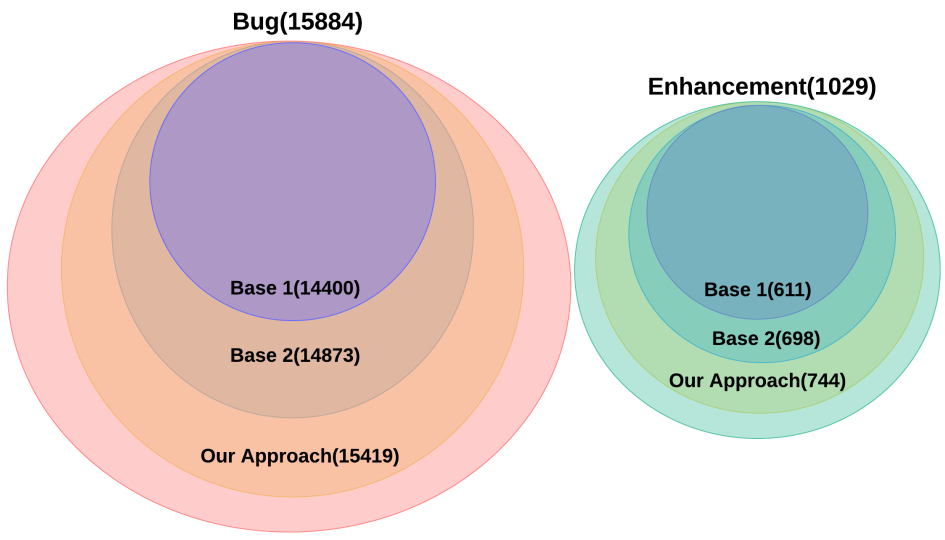


Figure 4.9: Qualitative analysis venn diagram

Table 4.4: Qualitative analysis (bug and enhancement)

Project	**Bug**			**Enhancement**			**Total Bug/ Enhancement**
	Base1	**Base2**	**Our Approach**	**Base 1**	**Base 2**	**Our Approach**	
aspnetcore	1124	1148	1382	21	25	29	1391/39
azure-powershell	434	448	455	3	4	5	499/8
ecplise	9670	9950	9980	78	91	96	10035/126
efcore	1038	1069	1216	51	65	68	1245/77
elasticsearch	811	820	857	45	66	69	939/99
mixedreality toolkitunity	386	413	435	9	13	14	435/23
monogame	122	144	151	10	14	16	180/21
nunit	65	79	97	8	13	16	109/22
realmjava	148	171	190	4	6	7	220/12
roslyn	291	299	303	360	370	390	458/560
rxjava	311	332	353	22	31	34	373/42
AVG	1309	1352	1406	56	63	68	
MAX	9670	9950	9980	360	370	390	
Total	14402	14873	15419	611	698	744	15884/744

representation of these records cannot provide valuable features for the model to perform the prediction. It was also observed that most of the descriptive information provided in the bug and enhancement reports used similar terms. For example, terms such as ‘add’, ‘improve’, ‘enhance’, ‘upgrade’ and ‘include’ are frequently used in both bug and enhancement reports. Thus, the baseline approach that relied on the issue reports’ textual features might be wrongly mislabelled as enhancement in some scenarios. The present approach uses textual and AST representation of the issue reports to eliminate the mislabelling case by using the additional context from code snippet metadata.

In addition, it was further observed that the reports failed to predict from all three approaches. Interestingly, these records do not include either non-trivial descriptive text or code snippets. These issue reports include screenshot images, stack trace information, and hyperlinks, which are ignored in all three approaches. Stack trace information was neglected by this approach to reduce noise in the model training. Screenshots were not covered due to limitations of the model, which supports either natural language or structural context.

4.6.2 RQ2: Which component contributes more to the multi-triage model?

Ablation analysis is performed on the multi-triage model to ascertain which component contributes more to model performance. To answer this question, the ablation analysis is divided into two sections: 1) system component level ablation analysis, and (2) embedding parameter level ablation analysis.

4.6.2.1 System Component Level Ablation Analysis

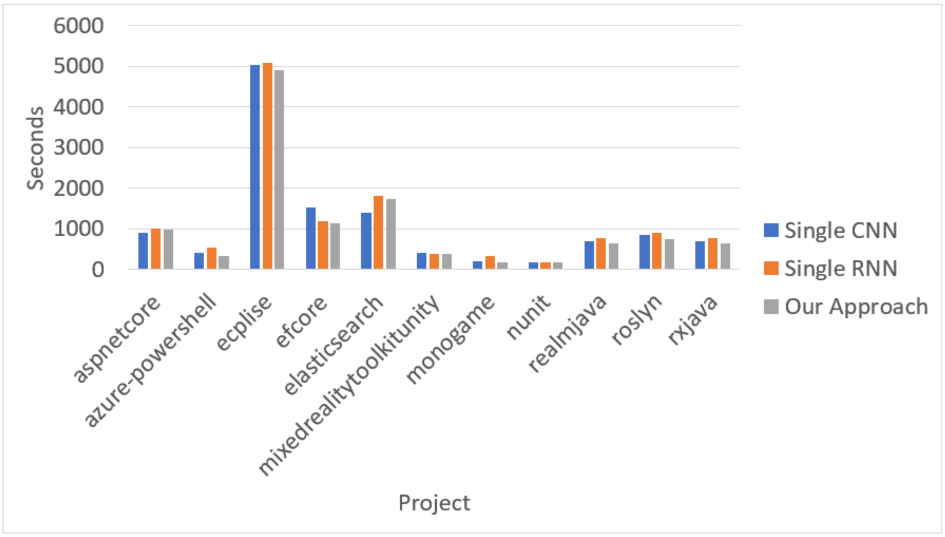


Figure 4.10: Training time

This section compares the multi-task learning model with the conventional single-task learning model to analyze which model performs better. The two single-task learning models, one with CNN and the other with BiLSTM networks, are implemented by referring to the present approach's encoder architecture. In a single model, the text and the AST path's are concatenated into one token and fed into the CNN, or BiLSTMs layer,

Table 4.5: Single task prediction model v.s. our approach for developer predictions (precision(P), recall(R), and accuracy(Acc))

Project	**Single CNN**		**Single BiLSTM**		**Multi-triage**	
	P	**R**	**P**	**R**	**P**	**R**
aspnetcore	57%	52%	57%	51%	66%	61%
azure-powershell	52%	40%	52%	40%	55%	42%
ecplise	43%	24%	50%	30%	52%	38%
efcore	53%	47%	53%	47%	62%	56%
elasticsearch	54%	44%	54%	44%	62%	53%
mixedrealitytoolkitunity	56%	51%	55%	51%	64%	60%
monogame	59%	59%	59%	59%	69%	69%
nunit	49%	42%	51%	45%	59%	52%
realmjava	55%	52%	55%	52%	64%	61%
roslyn	49%	33%	49%	33%	52%	35%
rxjava	58%	58%	58%	58%	68%	68%
AVG	53%	46%	54%	46%	61%	54%
MAX	59%	59%	59%	59%	69%	69%

((a)) Developers precision and recall results

Project	**Single CNN**		**Single BiLSTM**		**Multi-triage**	
	Acc	**F2**	**Acc**	**F2**	**Acc**	**F2**
aspnetcore	54%	52%	53%	51%	63%	61%
azure-powershell	44%	41%	44%	41%	48%	42%
ecplise	32%	24%	46%	33%	54%	38%
efcore	50%	48%	50%	47%	59%	56%
elasticsearch	47%	44%	48%	45%	58%	53%
mixedrealitytoolkitunity	53%	51%	53%	52%	62%	60%
monogame	59%	59%	59%	59%	69%	69%
nunit	34%	43%	37%	46%	41%	52%
realmjava	53%	52%	53%	53%	62%	61%
roslyn	35%	37%	34%	37%	39%	38%
rxjava	58%	58%	58%	58%	68%	68%
AVG	47%	46%	49%	47%	57%	55%
MAX	59%	59%	59%	59%	69%	69%

((b)) Developers accuracy and F2 results

respectively. The same classifier components are used in a single model. The output of the single-task learning model is either developers or issue labels. The comparison results are presented in Table 4.5 for developers and Table 4.6 for issue types predictions. Tables 4.5(a) and 4.6(a) present the precision and recall, whereas Tables 4.5(b) and 4.6(b) describe the accuracy and F2 scores. Out of the three models, the present model achieves the best performance in precision, recall, accuracy, and F2 score.

In terms of developer precision and recall, the present model outperforms the others by an average increase of 8 percentage points compared to single CNN, and 7 percentage points compared to single BiLSTM. It increases recollection by an average of 8 percentage points over both a single CNN and a single BiLTSM. In accuracy, on average, it exceeds the others by 10 percentage points compared to single CNN, and by 8 percentage points compared to single BiLSTM. In F2 scores, this model performs better than the single CNN by 9 percentage points, and the single BiLSTM by 8 percentage points. Therefore,

Table 4.6: Single task prediction model v.s. our approach for issue type predictions (precision(P), recall(R), and accuracy(Acc))

Project	**Single CNN**		**Single BiLSTM**		**Multi-triage**	
	P	**R**	**P**	**R**	**P**	**R**
aspnetcore	52%	38%	52%	37%	58%	39%
azure-powershell	49%	38%	51%	42%	59%	47%
ecplise	28%	22%	48%	34%	52%	33%
efcore	48%	36%	48%	34%	52%	33%
elasticsearch	44%	21%	43%	20%	48%	25%
mixedrealitytoolkitunity	49%	34%	50%	40%	55%	41%
monogame	53%	46%	55%	49%	60%	53%
nunit	48%	38%	51%	44%	58%	49%
realmjava	46%	35%	50%	43%	56%	45%
roslyn	46%	30%	47%	33%	54%	38%
rxjava	49%	41%	50%	43%	53%	44%
AVG	47%	34%	48%	38%	53%	40%
MAX	53%	46%	55%	49%	60%	53%

((a)) Issue types precision and recall results

Project	**Single CNN**		**Single BiLSTM**		**Multi-triage**	
	Acc	**F2**	**Acc**	**F2**	**Acc**	**F2**
aspnetcore	43%	43%	43%	43%	47%	43%
azure-powershell	39%	40%	39%	40%	44%	49%
ecplise	21%	20%	38%	37%	26%	39%
efcore	38%	38%	38%	37%	40%	38%
elasticsearch	29%	27%	29%	26%	31%	27%
mixedrealitytoolkitunity	40%	38%	45%	43%	47%	38%
monogame	49%	49%	51%	51%	57%	49%
nunit	28%	38%	24%	43%	27%	38%
realmjava	40%	37%	46%	45%	50%	37%
roslyn	22%	32%	25%	34%	27%	32%
rxjava	43%	43%	45%	45%	49%	43%
AVG	36%	37%	37%	39%	42%	39%
MAX	49%	49%	51%	51%	57%	49%

((b)) Issue types accuracy and F2 results

it can be concluded that developers and issue types prediction tasks are compatible with learning in one large network.

Interestingly, similar improvements were found for issue type prediction results. In issue types precision, the present model outperforms the others on average by 6 percentage points compared to single CNN, and by 5 percentage points compared to single BiLSTM. In recall, it improves on a single CNN by 6 percentage points and on a single BiLTSM by 2 percentage points, on average. When compared to a single CNN and a single BiLSTM, it outperforms the others in accuracy on average by 8 percentage points and 5 percentage points, respectively. In F2 scores, the model performs slightly better than single CNN by 1 percentage point, and the same for a single BiLSTM. Therefore, it is possible to conclude that developers and issue types prediction tasks are compatible with learning in one large network.

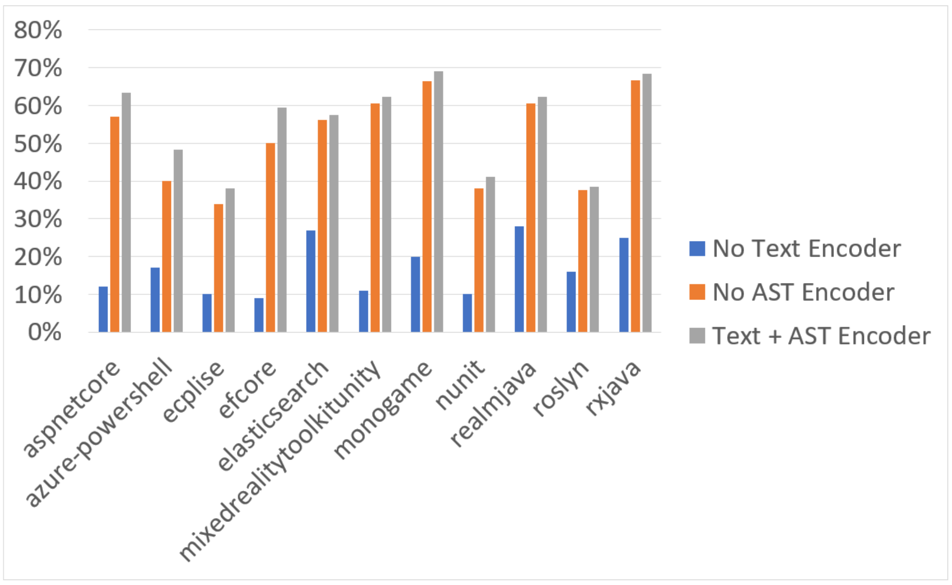
Training times for each model are also presented in Fig. 4.10. On average, the multi-triage model accelerates the training process with a drop of 476 sec and 1175 sec compared to the single CNN and single BiLSTM models, respectively. Although the accelerated training times are not obvious in the present scenario, imagine a project with N issue reports; the training time complexity of the single model is $(N^2 * t)$, where t is the time consumed by the model to learn feature representations of each issue report. However, the multi-triage model only needs $(N * t)$ times to learn the feature representation; therefore, the present model is more capable of scaling to train to projects with large amounts of training data. In summary, the multi-triage model outperforms the single-task learning model in terms of accuracy and training time.

4.6.2.2 Embedding Parameter Level Ablation Analysis

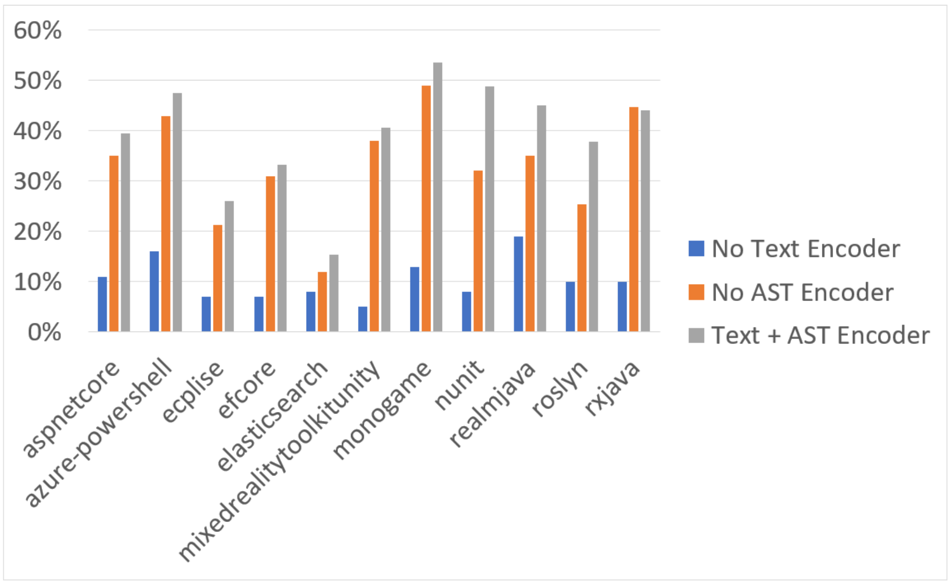
Two types of ablation analysis are performed to evaluate the embedding parameters: encoder decoupling and parameter tuning.

In the *encoder decoupling* experiment, the two encoders, text, and AST are decoupled, and the model‚Äôs performance is evaluated with three experimental settings: (1) no text encoder, (2) no AST encoder, and (3) both. In the *no text encoder* experiment, the negation effect of the textual input is studied. Similarly, AST path input is excluded in the *no AST encoder* experiment. The comparison results for prediction accuracy of developers and issue types in Fig. 4.11(a) and Fig. 4.11(b), respectively. In both predictions, the combination of textual and AST path inputs achieves the highest results in all eleven projects, with an average increase of 35 and 3 percentage points for developers and 23 and 6 percentage points for issue types in comparison with no text encoder and no AST encoder, respectively. Therefore, it can be concluded that both the textual encoder and AST encoder are important components of the multi-triage model.

In the *parameters tuning* experiment, the effects of embedding dimension and learning rate on the accuracy of our model were analyzed. The model was tuned with embedding dimensions (100 and 200) and learning rates (0.1, 0.01, and 0.001), which are the most commonly used hyperparameters in deep learning models. As previously mentioned, a time-series-based cross-validation approach was adopted, and the model was trained with various learning rates and embedding dimension size incrementally. Fig. 4.12 presents the accuracy results for the six experiments with developer prediction accuracy in Fig. 4.12(a) and issue types prediction accuracy in Fig. 4.12(b). In both prediction tasks, embedding dimension 100 with a learning rate of 0.01 provides the highest average, with an accuracy of 55 percentage points for developers and 41 percentage points for issue types. With an average accuracy of 53 percentage points for developers and 40

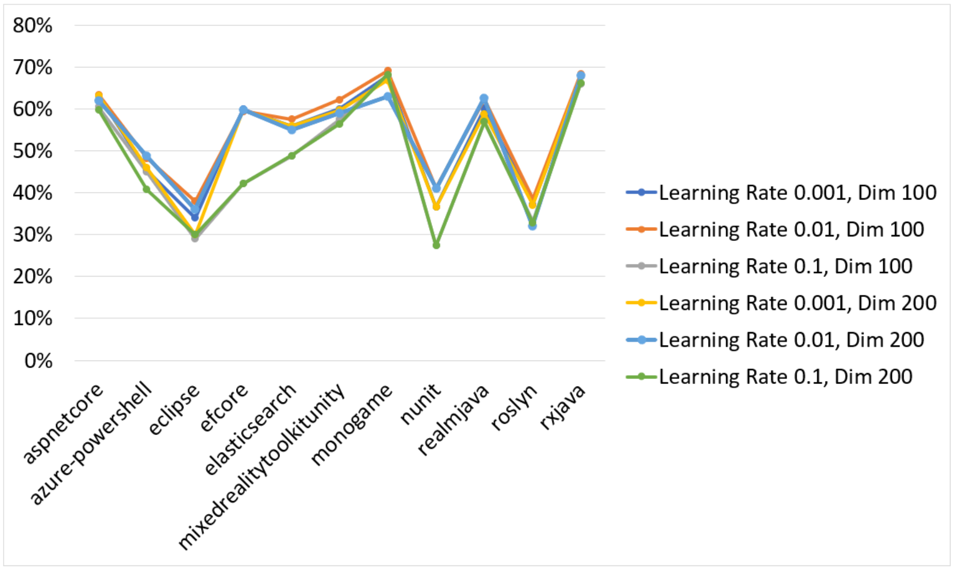


((a)) Developer

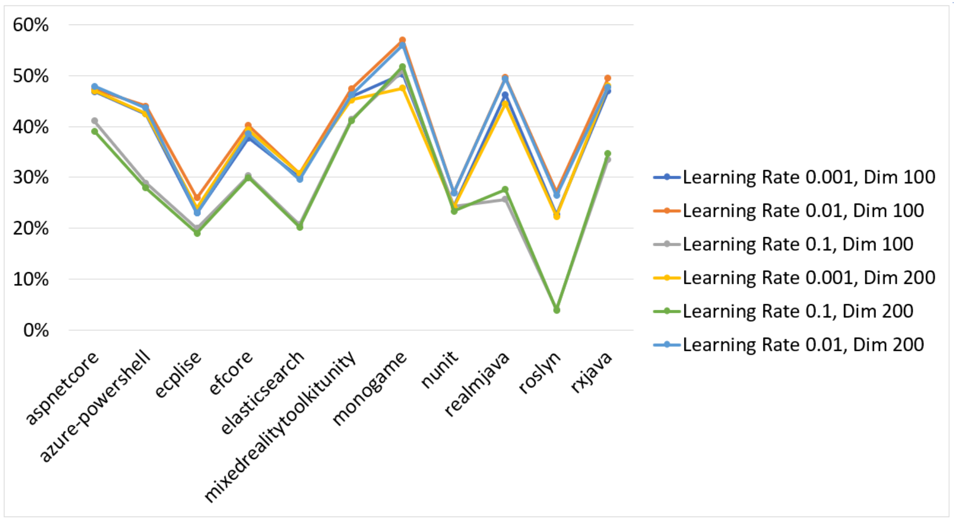


((b)) Issue Type

Figure 4.11: Multi-triage: ablation analysis



((a)) Developer



((b)) Issue type

Figure 4.12: Multi-triage parameter analysis

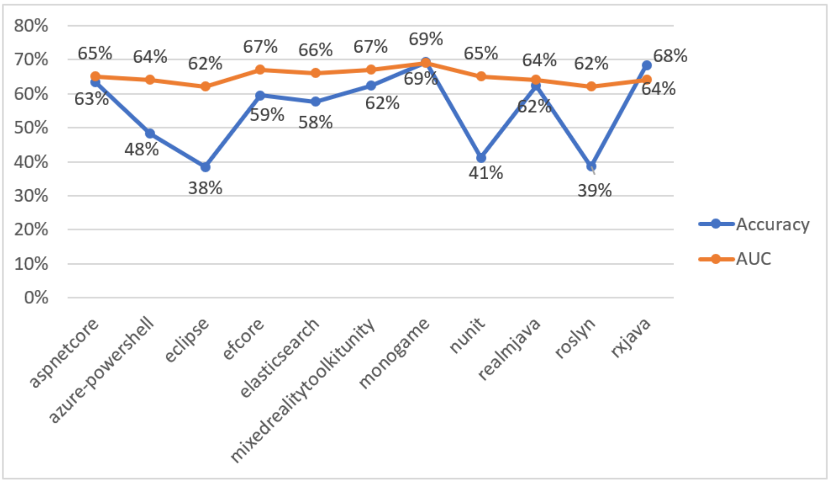
Table 4.7: Unique word count for Text and AST

Project	Text	AST
aspnetcore	32959	29559
azure-powershell	20005	5200
ecplise	342103	1234
efcore	24627	51115
elasticsearch	28116	223942
mixedrealitytoolkitunity	11749	3570
monogame	8839	6351
nunit	5231	3197
realmjava	10950	40145
roslyn	21265	25372
rxjava	11225	93517
AVG	47006	43927
MAX	342103	223942

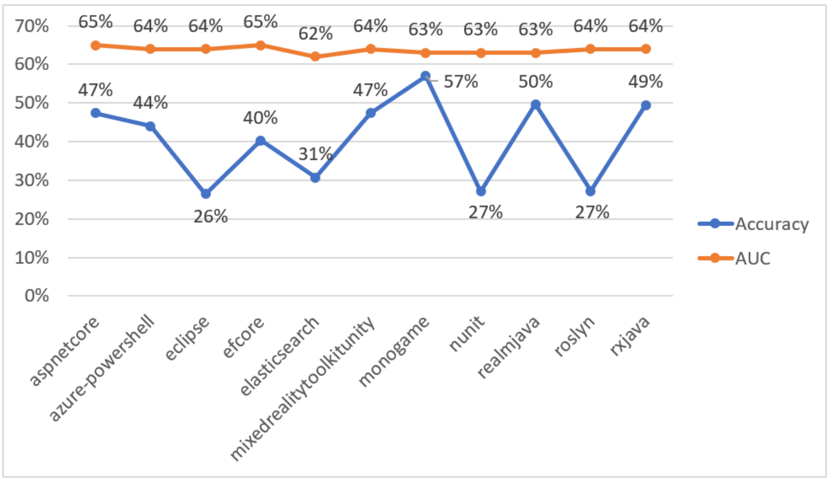
percentage points for issue categories, the embedding dimension 200 with a learning rate of 0.01 comes next. The internal validity of the embedding parameter results is further analyzed by validating the total number of unique word counts for both text encoder and AST encoder input for each project. Table 4.7 presents the word count results for all projects. Stop words and special characters were filtered out before the number of unique words was counted. As shown in Table 4.7, the average word counts for text encoder input is 47006, whereas the AST encoder input is 43927. The highest word count is 342103 for the text encoder and 223942 for the AST encoder, respectively. By following previous studies, a word corpus of around 2 million is trained with embedding size 300 or higher [81, 125, 129]. The maximum corpus size of the projects is lower than 35k, as it is reasonable that both 100 and 200 embeddings provide comparable results in these experiments. However, 100 embedding size was selected as the optimal hyper-parameter to eliminate complex processing. In summary, a learning rate of 0.01 with an embedding dimension of 100 hyperparameters was used as optimal parameters to train the model.

4.6.3 RQ3: Does increasing the size of training datasets (based on the contextual data augmentation approach) improve our model's accuracy?

In this section, the data-imbalanced problem is addressed with the contextual data augmentation approach presented in algorithm 1. First, an Area under the ROC Curve



((a))

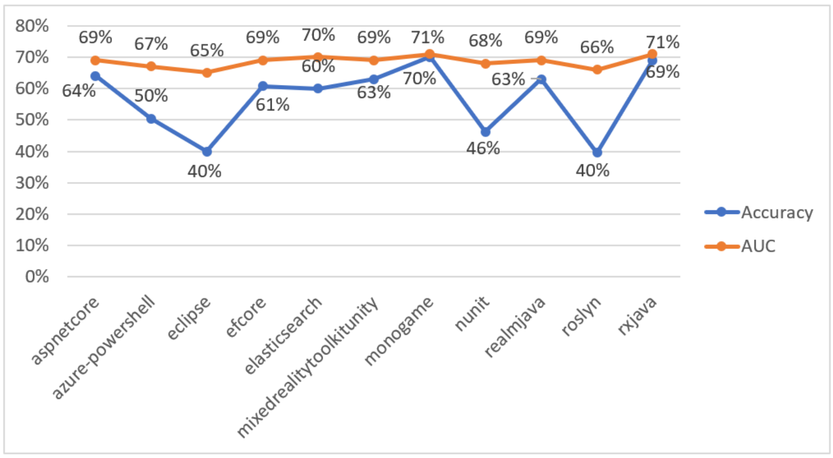


((b))

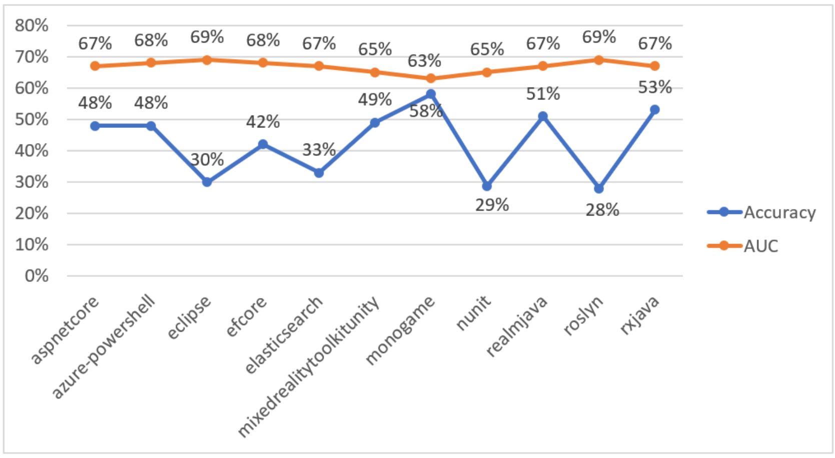
Figure 4.13: Multi-triage: AUC v.s. Accuracy

Table 4.8: No data augmentation v.s. data augmentation (accuracy(%))

	Multi-triage		**Multi-triage A+**	
Project	**Dev**	**Issue Type**	**Dev**	**Issue Type**
aspnetcore	63%	47%	64%	48%
azure-powershell	48%	44%	50%	48%
ecplise	38%	26%	40%	30%
efcore	59%	40%	61%	42%
elasticsearch	58%	31%	60%	33%
mixedrealitytoolkitunity	62%	47%	63%	49%
monogame	69%	57%	70%	58%
nunit	41%	27%	46%	29%
realmjava	62%	50%	63%	51%
roslyn	39%	27%	40%	28%
rxjava	68%	49%	69%	53%
AVG	55%	41%	57%	43%
MAX	69%	57%	70%	58%



((a))



((b))

Figure 4.14: Multi-triage with Data Augmentation: AUC v.s. Accuracy

(AUC) analysis is performed to measure classifier performance. Fig. 4.13 presents the average AUC and accuracy results for the multi-triage model. The line graph in 4.13(a) illustrates the developers' AUC and accuracy results, whereas the line graph in 4.13(b) shows the issue types AUC and accuracy results. In both tasks, AUC fluctuates around 62% and 69%, which indicates that the classifiers perform fairly well.

Therefore, further analysis was performed on the impact of the size of the training data on model accuracy. The training data size was increased by using algorithm 1. Table 4.8 presents the comparison results. For ease of reference, the model that uses augmented data was named as multi-triage (A). As mentioned earlier, the training data augmentation size was incrementally increased in each cross-fold validation as the average accuracy from the 5-fold validation result was reported. As shown in Table 4.8, the model accuracy slightly improved in the multi-triage (A) model, with an average increase of 2 percentage points on both developers and issue types. The performance of the prediction model was further analysed using the AUC test. Fig. 4.14 presents the AUC represented for the multi-triage (A) model. The line graphs in 4.14(a) and 4.14(b) illustrate the developers and issue types of AUC and accuracy results. Notably, AUC performance increased an average of 4 percentage points for developers and 3 percentage points for issue types in comparison to the multi-triage model AUC performance, as shown in Fig. 4.13. The data augmentation approach leveraged the base multi-triage model in both accuracy and AUC performance measures. Therefore, it is concluded that the contextual data augmentation approach effectively increases the issue reports training data.

4.7 Threats to Validity

Threats to external validity This is associated with the calibre of the datasets we utilized to assess our model. To generalize our work, we used problem reports from eleven open-source Java and C# applications. All the datasets'programs were collected from GitHub repositories; each dataset contains over 600 training issue reports. However, further studies are needed to validate and generalize our findings to other structural languages. Furthermore, more case studies are needed to confirm and improve the usefulness of our multi-triage recommendation model.

Threats to internal validity This includes the influence of hyperparameters settings. Our model's performance would be affected by different learning rates and embedding dimensions, which were set manually in our experiments. Another threat to internal validity relates to the errors in the implementation of the benchmark methods. For Deep-Triage [102], we directly used their published GitHub repository. For SVM+BOW [12], we implemented it ourselves using scikit-learn libraries, because the source code is not accessible. Nonetheless, scikit-learn is widely used in various studies [94, 159] to set up machine learning algorithms, including SVM. As a result, the implementation of the baseline faces few risks. In terms of the contextual data augmentation approach, we calculated the threshold amount (30,000) using the approximate total number of issue reports from targeted projects, based on the assumption that synthetic records should not be larger than the total. Thus, the threshold value can change based on the targeted project.

Threats to construct validity This relates to the applicability of our evaluation measurement. We use accuracy and the F2 score as the evaluation metrics that evaluate the performance of the model. They represent standard evaluation metrics for bug triage models used in previous studies [12, 102].

4.8 Discussion

This section discusses the implications of the accuracy, precision, and recall rates we achieved on our eleven experimental projects. We also report various alternatives we have considered in implementing our model and in choosing a time-series-based cross-validation approach. Then, we further discuss the decision to use a contextual data augmentation approach in generating synthetic issue reports. Lastly, we also review the lessons we have learned in applying a deep learning approach to an issue report contextual and structural information.

4.8.1 Accessing the Significance of Our Approach

Our approach achieves an average accuracy of 57% and 47% for developers and issue types, respectively. Also, our approach compromises precision and recall for both developers and issue types prediction results, with an average of 61—54% and 53—40% respectively. The only way to ensure these prediction rates are good enough for the bug triage process is by either performing a direct observation with human triages or by statistical analysis of the qualitative data. Our study performs qualitative analysis by categorizing the results into two generic issue report types (i.e. bug and enhancements) and observing the prediction results in terms of numbers. However, we envision our approach will be evaluated with human triages in the future. Notably, all the issue reports predicted correctly in baseline approaches are covered by our approach. In addition, our approach can correctly predict issue reports, which are missed by state-of-the-art approaches, due to our model capability to comprehend the structural context of code snippets. Therefore, we believe that the prediction rates we report in this paper for the eleven open-source projects are sufficient to assist human triages in assigning a developer and an issue type for a new issue report. As previously mentioned, there is an average of 67 days to fix a new issue report in these projects, due to the delay in triage becoming acquainted with the problem and finding the relevant developers. Our approach can reduce the time spent on issue report allocation tasks and regain the time to resolve the issues.

Furthermore, our multi-triage learning model takes advantage of the multitask learning approach to train the developers and issue types of classification tasks in one model. It reduces the training time substantially, compared to a single-task learning model. However, the multi-task learning model is pruned to encounter a negative transfer learning problem if prediction tasks are not compatible for learning together. We eliminate the problem by comparing our approach with two single-task learning models. To evaluate the two single models effectively, we designed these models in the same manner as the two neural networks used in our encoder layers (i.e. BiLSTM and CNN). Notably, our model surpasses single models in terms of precision, recall, and accuracy for both issue type and developer prediction. Because it is possible to learn in a single prediction network, we, therefore, evaluated a relationship between developers and issuer types of prediction tasks.

4.8.2 Evaluation using Time-Series Based Cross Validation

The standard method for evaluating the machine learning model is the K-fold validation approach. The original sample is divided into k equal-sized subsamples at random, and the model is trained repeatedly k times in the K-fold validation procedure. However, the standard K-fold validation approach is inappropriate in a time-ordered dataset, where future issue reports will be used to predict past bug reports. Therefore, we followed a time-series 5-fold validation approach and trained all our models, including the baselines approach. When we used the time-series approach, we noticed that the first one-or two-fold accuracy results are relatively lower than the later folds, due to smaller data size. The neural networks-based approach generally produces better results when there is more data available to learn. However, our time-series approach statically generalized the results based on how the issue report information flows and alters an issue tracking system.

4.8.3 Alternative Considerations on Model Building

We choose to use CNN in-text encoder and BiLSTM for AST encoder by referring to previous studies in similar areas [9, 95, 102]. Both of the networks are commonly used in natural language and structural language processing. Alternatively, we could incorporate the BiLSTM model for the text encoder or CNN for the AST encoder. However, in our preliminary test, the CNN model performs better than the BiLSTM in the text encoder layer, whereas the BiLSTM model performs better than the CNN in the AST encoder. Therefore, we choose the combination, which produces the best results.

4.8.4 Applicability of Contextual Data Augmentation Approach

We adopted a supervised machine-learning approach, as our triage model required a ground truth label for each report to training the model. Therefore, we faced an imbalanced class problem in our model training. When we evaluated our model with the AUC test, we observed that our model performance is slightly low, with an average of 65% for developers and 64% for issue types. Thus, we adopted the contextual augmentation approach to generate synthetic issue reports to balance developers and issue type label distribution on training samples. In general, there are two ways to develop synthetic reports with the contextual augmentation approach: 1) random word substitution, and 2) random word removal [77]. We have selected the substitution approach as we do not want to lose the important information of the issue report. We incrementally generated the synthetic reports using a time-series cross-validation approach and trained model. Since we are interested in the performance of our model, we statistically evaluated the improvement of the data augmentation approach using the AUC test. Notably, our model performance rose on average 69% for developers and 67% for issue types. Therefore, we considered that the contextual augmentation approach is reasonable for smoothing label distributions in the supervised learning approach.

4.8.5 Lessons Learned

The information we use comes from issue reports as text and code snippets. It's possible that including more data will increase the accuracy of our technique.

A screenshot image is a valuable asset for issue reports, providing additional information about user requirements. The execution stack trace from issue reports can be used as the pointer to identify a code area in recommending issue types. As mentioned in Section 4.2, the GitHub projects issue types label includes project areas or components information. Identifying the project areas or components can assist in finding potential developers by looking into the list of developers who are actively working on these areas, either using the code ownership information or previous issue assignment history. However, as explained in Section 4.4, stack trace introduces noise into the model training, as we neglected this information. Also, correlating code ownership information to issue reports is challenging, especially for large projects evolving throughout time.

4.9 Related Work

This section introduces previous studies related to the semi-automatic bug triage process and multi-task learning model. Moreover, other studies related to bug resolution (e.g. bug localization) are discussed.

4.9.1 Semi-Automatic Bug Triage

In an early work of [111], the authors proposed an automatic bug triage approach that used a native Bayes (NB) classifier to recommend candidate developers to fix a new bug. Later, [12] extended this by comparing the work of [111] with three machine learning classifiers: NB, SVM, and C4.5. Their preliminary results found that SVM outperforms the other classifiers. In [104], the authors proposed an approach to modeling developers' profiles using the vocabularies from their changed source code files, compared with terms from issue reports to rank the relevant developers.

A comparison of different machine learning algorithms (i.e. NB, SVM, EM, conjunction rules, and nearest neighbors) to recommend potential developers can be found in [13]. In general, the authors used project-specific heuristics to construct a label for each issue report rather than using the assigned-to field, to eliminate default assignee assignment and duplicate reports with unchanged assigned-to field problems. In our approach, we alternatively address these problems by filtering out issues assigned to *software bots* and including the corresponding pull request's developer information as the tossing sequence in our labeling process.

Similarly, [164] proposed a concept profile and social network-based bug triage model to rank expert developers to fix a bug. In their work, concept profiling first defines the topic terms to cluster the issue reports. Then, the social network feature captures a set of developers' collaborative relationships, extracted from the concept profiles, to rank the candidate developers based on the level of expertise (i.e. a fixer of a bug, a contributor of a bug). In [124], the authors proposed CosTriage to assist triage in finding candidate developers who can fix the bug in the shortest time frame. CosTriage adopts content-boosted collaborative filtering (CBCF), which combines issue report similarity scores with each developer's bug fixing time to recommend relevant developers for a new issue report.

Aside from developer recommendation studies, other studies have focused on automating issue type prediction in the bug triage process. In [156], the authors proposed TagCombine, an automatic tag recommendation method, which is based on a composite ranking approach to analyzing information on software forum sites (i.e. Stack Overflow, free-code). TagCombine consists of three ranking components: multi-label ranking, similarity-based ranking, and tag-terms-based ranking. In their approach, multi-nominal NB classifier, Euclidean distance algorithm, and latent semantic indexing (LSI) are used to calculate three component scores separately. The linear combination score of these three components is then used to recommend a list of relevant tags for a new issue report. In [154]], the authors proposed MLL-GA, a composite method to classify crash reports and failures. MLL-GA adopts various multi-label learning algorithms and genetic algorithms to identify faults from crash reports automatically.

In the work of [163], the author adopted a BM25-based textual similarity algorithm and KNN to predict severity levels and developers for a new issue report. In [10], the authors adopted machine learning classifiers, such as NB and SVM, to predict an issue label (e.g. bug, enhancement) for a new issue report. Their approach uses the bag-of-words model to represent issue reports in text classification. In this representation, every word in the training corpus is considered a feature; therefore, each issue report presents a sparse representation with a high number of features. These features are used by machine learning classifiers to predict issue labels for new issue reports. Recently, in [102], the author used the BiLSTM model to recommend potential developers.

Our work is closely related to that of [102]. However, our multi-triage model adopts a multi-task learning approach and recommends both developers and issue types from one learning model. As such, it reduces a considerable amount of training time in comparison to the single-task learning model. In addition, our model uses both textual and structural information (i.e. code snippets) to learn the representation of issue reports, as doing so provides a more accurate representation. In comparison, previous studies have neglected the code snippet information in order to reduce noise in the model training. Our system eliminates the possibility of adding noise to the model by converting the code snippet to AST routes and learning the representation in a separate encoder.

There are several techniques to parse AST from partial programs. Some of the well-known approaches are fuzzy parsers [87], island grammars [110], partial program analysis (PPA) [44] and pairwise paths [9]. Fuzzy parsers scan the code keywords and extract the coarse-grained structure out of code snippets [87]. Similarly, island grammars extract part of the code snippets that describes some details of the function (island) and ignores the rest of the trivial lexical information (water). In contrast, PPA parsers trace the defined type of a class or method and extract a typed AST [44]. PPA recovers the declared type of expression by resolving declaration ambiguities in partial java programs.

Undeclared fields or unqualified external references are referred to as having ambiguous declarations. These approaches are more suitable for situations where a sound analysis is required, such as code cloning, code representation, and code summarization.

Lastly, in the pairwise paths parser [9], the AST paths are extracted using modern integrated development environments (IDE) (e.g, Eclipse), which generate the pairwise paths between terminal nodes (e.g. variable declaration) by neglecting the non-terminal nodes (e.g. do-while loop). In the pairwise path approach, two programs that have similar terminal nodes are likely to parse in a similar format. As we intend to compare similar code snippets between issue reports, we have adopted these pairwise paths approach in our study. Next, we discuss the related work of the multitask learning model.

4.9.2 Multi-Task Learning

The multi-task learning model has been successfully applied in computer vision applications as well as in many natural language problems which require solving multiple tasks simultaneously [97, 139]. In the recent work of [86], the author's used hard parameter sharing to address seven computer vision tasks. Similar works are presented in [26, 52, 166]. In the work of [140], the authors proposed a framework by which to evaluate which tasks are compatible with learning jointly in the multi-task learning network. Their preliminary results revealed that multi-task learning networks' prediction quality depends on the relationship between the jointly trained tasks. Their framework incrementally increases the number of tasks assigned to the model by starting with three or fewer networks. They used predefined inference time, and the lowest total lost value to identify the compatible pairing tasks.

In [48], the authors used a multi-task learning approach to tackle two types of question-answering tasks: answer selection and knowledge-based question answering. In their approach, the CNN network is used to model the shared learning layer to learn the contextual information of historical questions and answer data to predict answers to a new question automatically. Similar research was conducted by the authors of [165], who identified face landmarks and characteristics by using the CNN network (i.e. emotions). To learn query classification tasks and ranking of online search tasks simultaneously, the authors of [48] employed a multitask learning network. Our work is similar to that of [48], but we tackle a problem in a different domain. We adopted multi-task learning with a hard-parameter sharing approach to recommend potential developers and issue types for a new issue report.

4.9.3 Other Tasks in the Bug Resolution Process

In [30], the authors reported the usage of GitHub's label in over 3 million GitHub projects. Their preliminary results revealed that most projects use four generic types of labeling strategies: priority labels, versioning labels, workflow labels, and architecture labels to categorize the issue reports. In [128], the authors proposed a CNN-based bug localization model to assist developers in identifying code smell areas. In the work of [49], the researchers leverage deep neural networks to detect duplicate issue reports automatically.

Likewise, [144] relied on recurrent neural networks (RNN) and graph embedding to detect similarities in source code components. The work proposed in [150] used deep learning neural networks to identify similar code components in generating bug-fixing patches for program repair. In [147], an LSTM encoder-decoder was used to generate a code summary that provided a high-level description of code functionality changes. Despite different strategies, these approaches use AST tokens as embedding input to learn the representation of source code components. Instead, the work in [36, 141] used the control flow graph (CFG) representation of a program to embed the code to support a variety of program analysis tasks (e.g. code summarization and semantic labeling).

4.10 Chapter Summary

Developer assignment and allocating relevant categories to a new issue report are common tasks in the bug triage process needed in any software project. In this chapter, we have introduced Multi-triage, developers, and issue types recommendation models for integrating into the bug triage process workflow. Multi-triage used deep learning techniques to predict potential developers and issue types for a new issue report. Our goal in this work has been twofold. First, we have combined CNN and BiLSTM neural networks to learn a representation of the issue report text description and code snippets to construct the precise representation of the issue report. Second, we have presented the benefits of using the contextual data augmentation approach to address the data-imbalanced problem. The multi-triage mode is effective in finding a relevant developer and issue type in eleven open-source projects from diverse domains. In training-time, Multi-triage is significantly faster than a single task learning model by a total drop of 476 sec and 1175 sec compared to the single CNN and single BiLSTM models. In accuracy, on average, it outperforms the others by 10 percentage points compared to a single CNN, and by 8 percentage points compared to a single BiLSTM.

Chapter 5

Case Study of Automatic Bug Triage Process Model at Software Industry

5.1 Overview

In this chapter, we briefly discuss the automatic bug triage process model project performed at Dialog Information Technology (IT). One of the most prestigious privately-owned IT companies is Dialog IT, which was founded in 1979. The company delivers innovative IT solutions ranging from full life-cycle application development and managed application services to long-term operational support to clients across all industries, including all levels of government. This industrial case study fulfills parts of the requirement for pursuing a Ph.D. (Software Engineering) research degree at the University of Technology, Sydney.

Prior studies assess the recommendation model's validity using open-source projects as it cannot enable a holistic review to ensure the approach applies to different aspects of software projects. Therefore, we perform the case study in this research to size up real-life business problems in designing the recommendation model while considering the broader organizational requirements. The findings of this case study contribute to the body of knowledge in the bug triage community, where there is a need to obtain an in-depth understanding of putting theories into practice.

As part of the ongoing maintenance and application support processes, the company uses diverse issue tracking systems to manage issue reports. Thus, leveraging the bug triage process is an integral component in efficiently managing the application. The scope of the project involves analyzing the company's in-house issue tracking systems, interacting with technical consultants from different departments, collecting historical issue reports, and developing artificial intelligence (AI) based centralized issue tracking system.

The chapter starts with a brief overview of issue-tracking applications that are currently used in the company and the business sector of Dialog IT. Subsequently, I proposed a new AI-based centralized issue tracking system and developed the proof of concept (POC) prototype application to provide an in-depth presentation of the design mode. Thereafter, I evaluated the prototype application with the company's historical issue reports and discussed the findings. In the end, I provided suggestions and recommendations for the company's future.

5.1.1 Contributions

The significant contributions of this research are summarized as follows:

- To leverage the company's existing bug triage process model, I proposed a centralized automatic bug triage process model to recommend potential developer issue types automatically, as well as similar issue, reports relevant to a new issue report.
- To effectively observe the impact of the process model, I developed a web-based prototype tool using a common development platform used in the company
- To evaluate the feasibility of the recommendation model in industrial settings, I extensively evaluated the model with the existing issue reports of three company projects and presented the results.

5.2 Background

Software issue reports generally contain technical problems, steps to reproduce the errors, and requests for enhancement. These reports are typically stored in issue-tracking systems. The process of managing an issue tracking system involves reviewing new issue reports to ensure they are valid (for example, duplicate reports), finding appropriate developers for assignment, and classifying them into the relevant issue types (for example, bug, feature, and product components). However, manually performing these processes is time-consuming and error-prone. Currently, the Dialog IT company uses a manual bug triage process in assigning developers and classifying issue types. Consequently, it delays the process of triaging issues. In this case study research, we proposed the AI-based bug triage to leverage the developer and issue type classification process to facilitate the company's issue resolution process.

5.2.1 Dialog IT Issue Tracking Systems Background

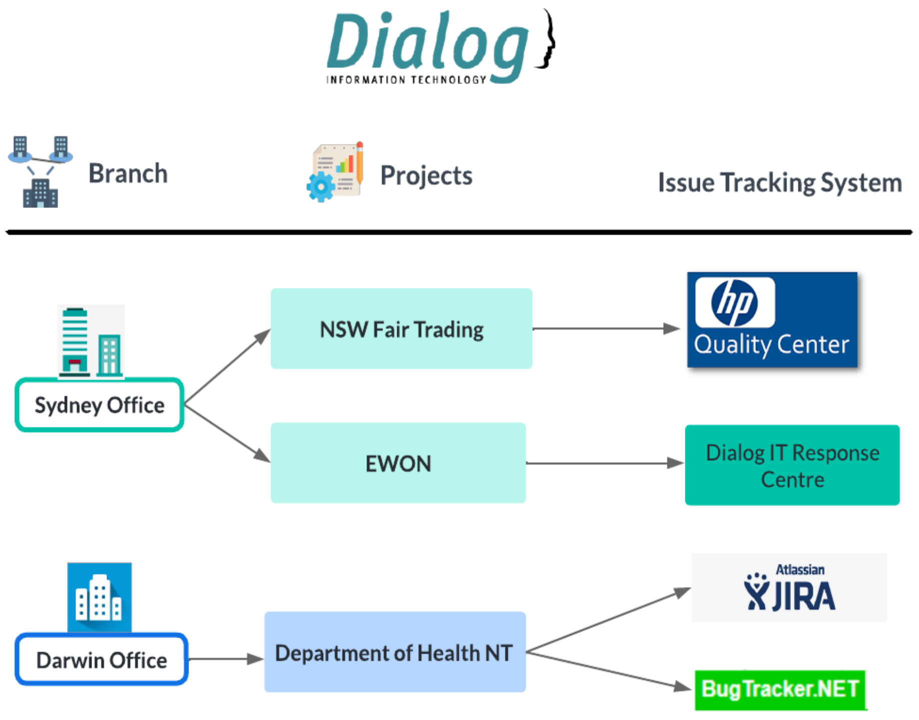


Figure 5.1: Issue Tracking Systems used in Sydney and Darwin Offices

During the feasibility study, we discovered that the company utilized a multitude of cloud-based platforms for tracking issues, which are managed and kept up to date by several departments spread across Brisbane, Sydney, Melbourne, Canberra, Perth, Darwin, and Adelaide. The main branch office is located in Brisbane and is traded nationally. With the agreement and arrangement of the intern supervisor of the company, we analyzed the issue tracking systems that are currently used in the Sydney and Darwin branches and collected the historical issue reports of three active projects for the purpose of this research. Figure5.1 presents the issue tracking platforms used in these offices.

- **NSW Fair Trading Project:** It is a rental bond online management application used by tenants, agents, and self-managing landlords to securely and easily lodge and refund bond money. The company has been providing end-to-end application management services since 2013. Currently, the company uses the HP Quality Center management system to log the issue raised by the clients.

- **Energy & Water Ombudsman NSW (EWON) Project:** EWON is the government-approved dispute resolution scheme for NSW electricity and gas customers and some water customers. The company provides ongoing support and maintenance services to manage the case and membership management system. EWON issue requests are managed through the Dialog in-house response center application.

- **Dialog NT Project:** In this project, the company provided the Department of Health divisions with related projects to conduct a feasibility study. Currently, the company in Darwin provides ongoing end-to-end application support to various Department of Health NT projects. The Department of Health is related to the Ministerial responsibilities of Health, Families, and Children and Child protection. The agency is responsible for several services in the Health Sector of the Northern Territory. These sectors include Aged & Disability Services, Cancer Services, Community Health, Dental Services, Hospitals, Palliative Care, Remote Health Centers, Renal Services, and Aboriginal Health. All issue reports are registered and maintained in JIRA and Bugtracker.NET issue tracking systems.

5.3 Proposed Solution

As presented in Figure 5.1, the company used multiple issue tracking applications to manage various projects which are developed and maintained by technical consultants from different geographical locations. The NSW Fair Trading and Dialog NT projects allow the company to use third-party issue tracking systems due to the client's needs. Thus, similar issues that could arise throughout the multiple project development lifecycle cannot be easily searchable, as the company cannot easily share the bug resolution knowledge between different departments. In addition, bug triage tasks, such as finding appropriate developers and categorizing the issue types, are performed manually as they are time-consuming and error-prone. Therefore, I propose the AI-based issue tracking system, which can recommend potential developers and issue types for a new issue report.

Figure 5.3 presents the proposed AI-based automatic bug triage process model to leverage the existing issue tracking process in the company. To reduce the time and

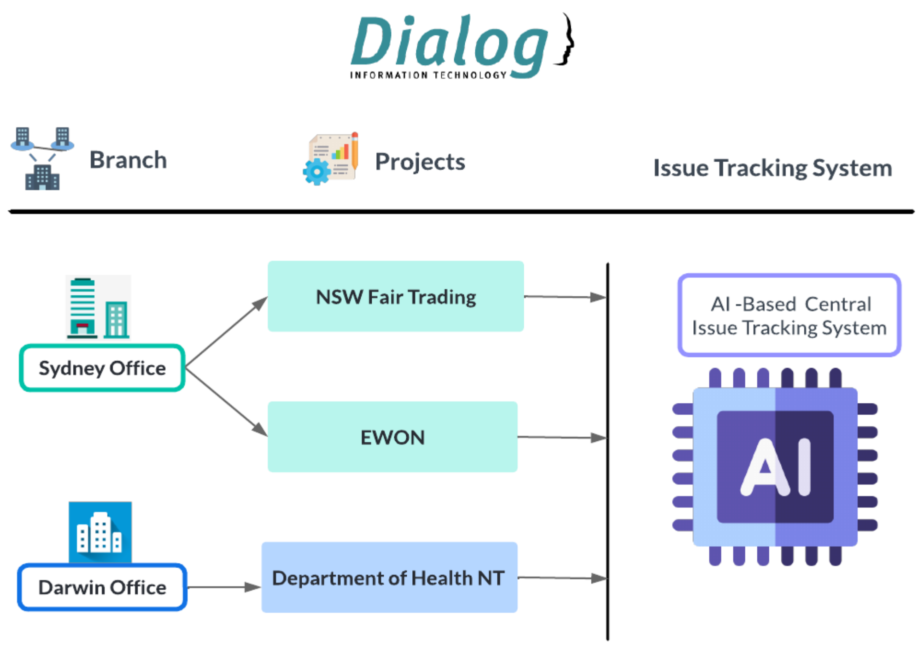


Figure 5.2: Automatic Bug Triage Process Model

effort required to manage multiple issue tracking platforms, I introduced a centralized AI-based bug triage process model to predict potential developers and issue types for new issue reports. The main purpose of a centralized issue tracking system is that technical consultants from different geographical locations can communicate and share their expertise in the bug-fixing process. In addition, training the developers and issue types recommendation models using data from the centralized data management system can empower the model performance and provide a better result. In the following section, I have described the proposed process model implementation in detail. I utilize a prototype application in this project to visualize and assess the process model.

5.4 Research Method

In this study, we applied the design science research method to experiment with the bug triage process model. Design science is "a problem-solving paradigm" [73], that is widely adopted in most Information System (IS) research. Design science research is suitable for creating new artifacts for a specified problem domain to resolve "the unsolved problem or solving a known problem in a more effective or efficient manner" [73]. In this case study, we intend to implement an AI-based bug triage artifact to resolve the challenges of finding potential developers and issue types and building a centralized bug triage model.

Therefore, we followed the design science method and performed the steps outlined in the following Table 5.1.

Table 5.1: Research Method

Design Science Guidelines	Project Process Mapping
G1 - Design as an artifact	Design a centralized AI-based bug triage artifact that supports the developer and issue type recommendation model.
G2 - Problem relevance	Develop an AI-based bug triage prototype application for the company to evaluate the effectiveness of the artifact. Figure 3 presents the high-level design diagram of the artifact.
G3 - Design evaluation	To effectively evaluate the prototype model, I collected the historical issue reports from various ongoing projects. The primary researcher frequently communicated with technical consultants involved in these projects to gain insightful knowledge about the projects.
G4 - Research contributions	The significant contributions of the research are an AI-based bug triage tool, which includes a developer and issue type recommendation model.
G5 - Research Rigor	We used the iterative design development approach to design the bug triage model and evaluate existing issue reports from various projects with the company. Furthermore, we applied a statistical data analysis approach to evaluate the AI recommendation model performance.
G6 - Design as a search process	We iteratively designed the application based on the structure of individual issue reports, which are used in various projects, and the feasibility of AI recommendation model implementation.
G7 - Communication of Research	The primary researcher shared the project progress with an intern supervisor and principal superior on a monthly basis throughout the duration of the project development.

5.5 Design Solution

Figure 5.3 presents the high-level AI-based bug triage process model proposed in this project. In general, the process model consists of three main components, namely 1) bug triage app, 2) bug triage hosting components, and 3) bug triage application service.

1. **Bug triage application** The dialogue consultants (technical) can manage the issues for many projects in one place using this front-end application. We used a model-driven Microsoft Power Application platform to design the application so that the users can access the application either through the mobile application or a web browser.

2. **Bug triage hosting components** The purpose of the hosting layer is to integrate the bug triage application with application services layers. We used the Microsoft Azure platform and published the bug triage application and application services.

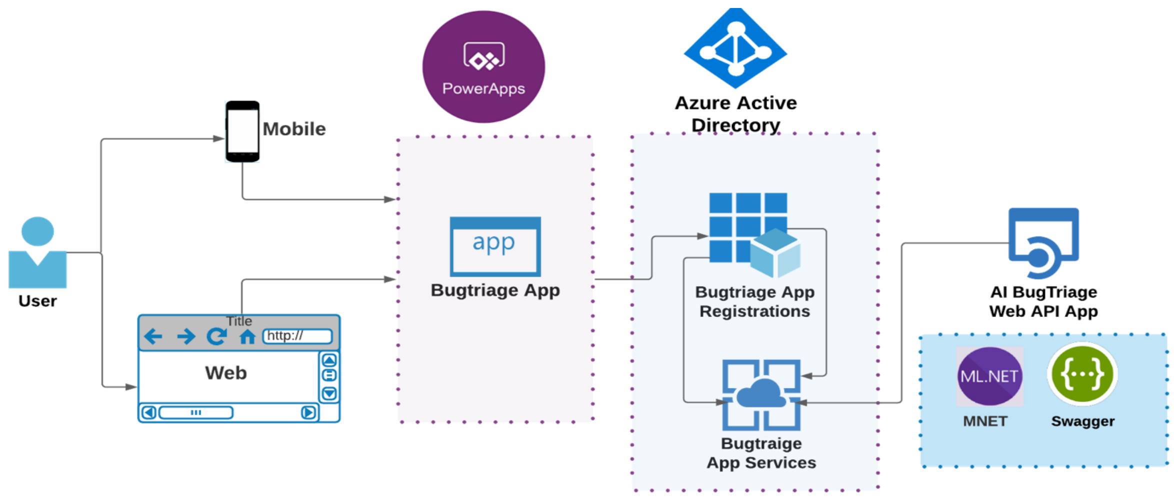


Figure 5.3: High-Level AI-based Bug Triage Model

3. **Bug triage application services** The purpose of building the application services is two-fold. First, we used the application service to retrieve the list of issue reports from the bug triage front-end application and train the developer and issue type recommendation models periodically or based on user demand. Thereafter, we created an application service to recommend potential developers and issue types for a new issue report. We used ML.Net and swagger application programming interface (API) services generator to design the API services.

We incrementally designed the three main components above. Second, we collected the issue reports from three projects mentioned in Section 5.2.1 and imported these reports into the bug triage application. Lastly, we built the developer and issue types recommended model in bug triage application services and used these existing issue reports to train the model. Finally, we created the front-end components in the bug triage application to display the prediction results. We detailed our findings in the next section.

5.6 Results

As mentioned in Section 5.5, the proposed process model has three main components, namely 1) bug triage application, 2) bug triage hosting components, and 3) bug triage application service. In this section, we reported the data analysis results of historical issue reports from the three company projects, the AI recommendation performance, and the outcomes of each component design.

5.6.1 Data analysis of historical issue reports

This section outlines the data statistics for each project to visualize the data-set clearly. In this study, we explored the three ongoing projects mentioned in Section 5.2 and presented the data analysis results in figures 5.4 and 5.4.

A total of 11,253 problem reports were utilized to train the AI recommendation model, as illustrated in figure 5.4, to anticipate the pertinent developers and issue types for a new issue report. Figure 5.5 describes the total no of developers and issue types currently used in these projects. Among the three projects, the Dialog NT project has the highest developers and lowest issue types. In the Dialog IT project, the defect-related issues are categorized as a bug, and the rest of the issue reports are logged as an improvement. Exploring the class label distributions among datasets makes it easier to identify the individual model requirements and specify the needs accordingly. For example, the binary classification model is suited for dialog NT issue type prediction model, whereas the multi-label classification model is applicable for others.

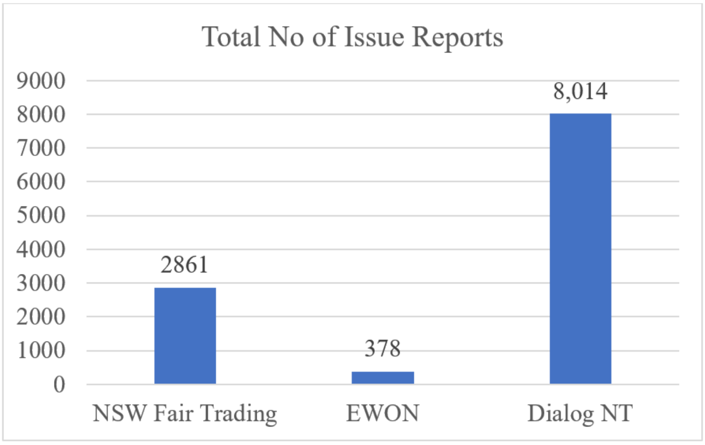


Figure 5.4: Total no of Issue Reports Summary

5.6.2 AI recommendation model performance

To evaluate the performance of the AI recommendation, we used the three standard machine learning evaluation techniques described in equations 5.1, 5.2, and 5.3. Precision is a fraction of relevant instances among the retrieved instances, while recall is a fraction of relevant instances that were retrieved. The accuracy is the fraction of the number of correct predictions to the total number of input samples.

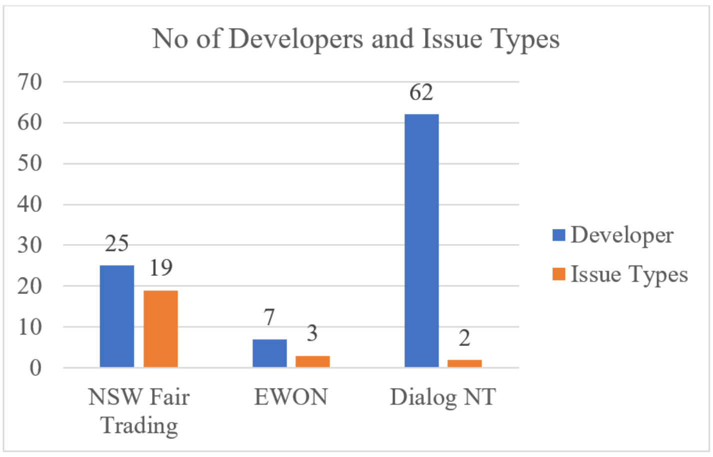


Figure 5.5: Total no of Developers and Issue Types Summary

$$\text{Precision} = \frac{TP}{TP+FP} \tag{5.1}$$

$$\text{Recall} = \frac{TP}{TP+FN} \tag{5.2}$$

$$\text{Accuracy} = \frac{TN+TP}{TN+TP+FN+FP} \tag{5.3}$$

Table 5.2 presents the performance of the two models in three projects. In the developer recommendation model, the NSW Fair Trading project outperforms the accuracy of the others by 75%. The developer class distributions in the NSW Fair Trading dataset are more balanced than other projects. In the issue-type recommendation model, the Dialog NT project outperforms the accuracy of the others by 100% because Dialog NT has lesser issue-type classes than the other projects.

Table 5.2: Evaluation Results

Project	**Developer**			**Issue Type**		
	Precision	**Recall**	**Acc**	**Precision**	**Recall**	**Acc**
NSW Fair Trading	22	25	75	94	95	98
EWON	22	18	60	95	96	98
Dialog IT	28	21	60	100	100	100
AVG	24	21	65	96	97	99

5.6.3 Bug triage application

Figure 5.6 presents the bug triage front-end application, which is - built with the Microsoft Power Apps Platform. The prototype application can manage issue reports for three different projects such as NSW Fair Trading, EWON, and Dialog NT project. We imported the existing issue reports, which are dispersed across other issue tracking applications, to one centralized location to train the developer and issue recommendation models. In the bug triage application, we added the Train AI Model functionality to train the recommendation model using the front-end application to improve the performance of the recommendation model though the system evolved. Figure 5.7 presents the results of the training model for EWON project. Currently, the recommendation model can conduct training based on the project. Once the model is trained, the users can retrieve the prediction results by triggering the “Recommend” button presented in Figure 5.8. The recommendation results are shown in figure 5.9. Currently, the prototype application provides three recommendations 1) relevant issue type, 2) relevant developer, and 3) similar existing issue reports assisting the bug triages in resolving the issue.

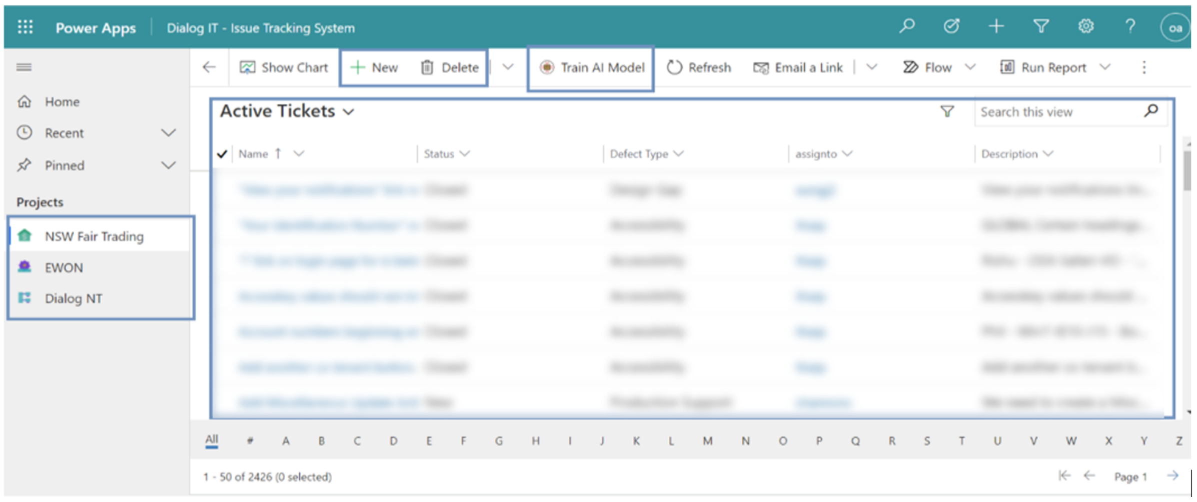
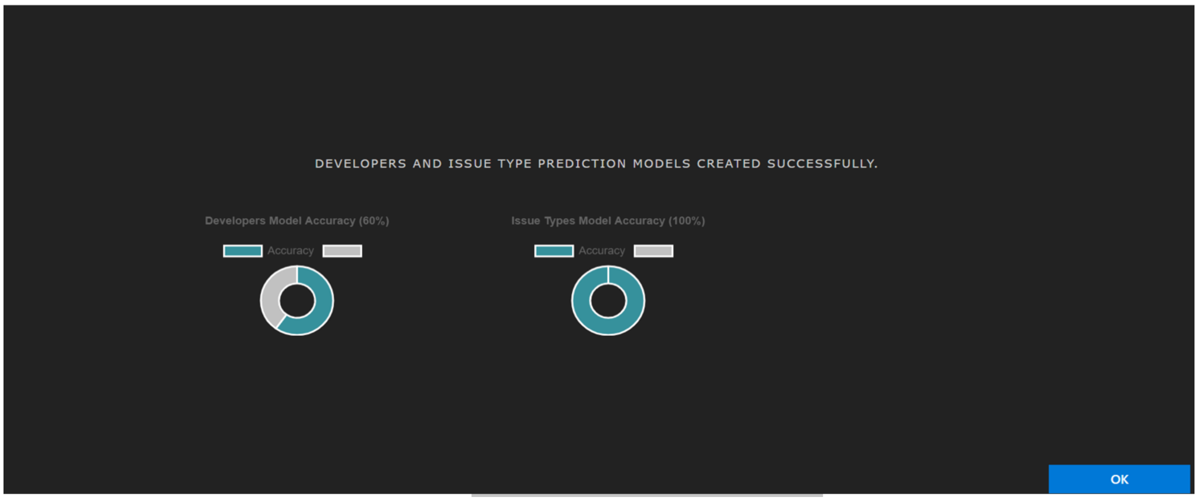


Figure 5.6: Bug Triage App Main Page

5.6.4 Bug triage hosting components and application services

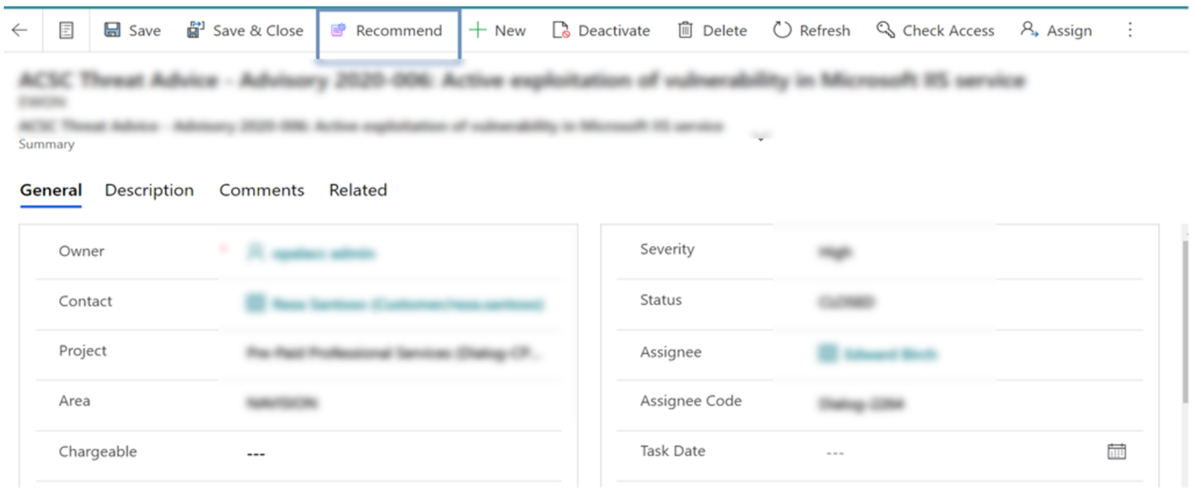
In terms of hosting the components, we used the Azure cloud-based platform to integrate application components on a central platform. In this study, we implemented six API services for each project to retrieve existing issue reports, train recommendation models, and return prediction results for a new issue report. The summary of each service is provided below.



DEVELOPERS AND ISSUE TYPE PREDICTION MODELS CREATED SUCCESSFULLY.

Developers Model Accuracy (60%)
Accuracy
Issue Types Model Accuracy (100%)
Accuracy
OK

Figure 5.7: On-Demand Train AI Model Example



[redacted]

Summary

General Description Comments Related

Owner	[redacted]	Severity	[redacted]
Contact	[redacted]	Status	[redacted]
Project	[redacted]	Assignee	[redacted]
Area	[redacted]	Assignee Code	[redacted]
Chargeable	---	Task Date	---

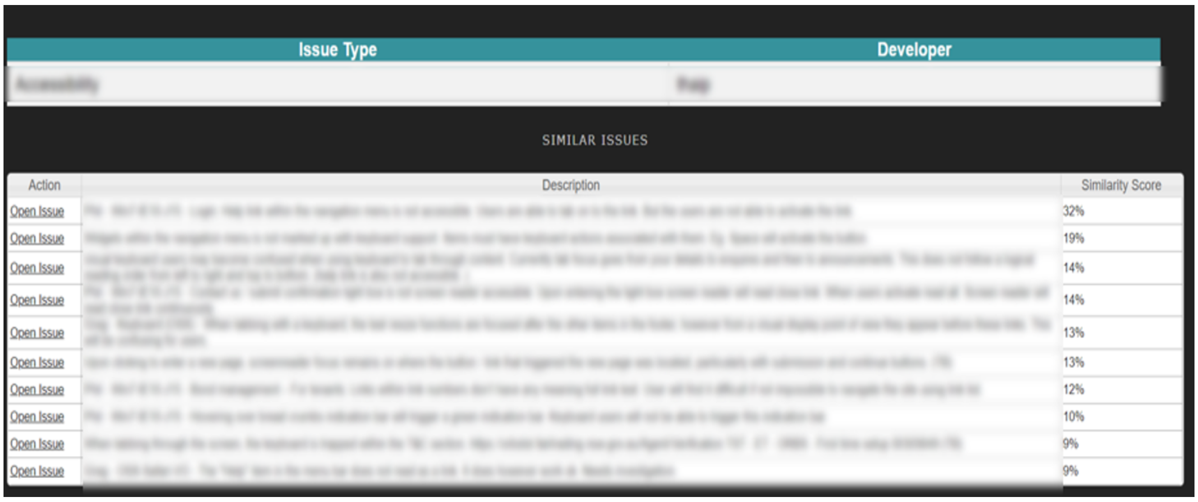
Figure 5.8: Retrieve Prediction Result for a New Issue Report

1. **GetTickets** This function is used to retrieve a list of defect tickets from the bug triage application to train the developer and issue type prediction model.

2. **TrainBugTriageModelToPredictDevelopers** This function is used to train the developers prediction model with existing defect tickets.

3. **TrainBugTriageModelToPredictIssueTypes** This function is used to train the issue types prediction model with existing defect tickets.

4. **PredictDevelopers** This function predicts potential developers for new defect tickets.



Issue Type	Developer
[illegible]	[illegible]

SIMILAR ISSUES

Action	Description	Similarity Score
Open Issue	[illegible]	32%
Open Issue	[illegible]	19%
Open Issue	[illegible]	14%
Open Issue	[illegible]	14%
Open Issue	[illegible]	13%
Open Issue	[illegible]	13%
Open Issue	[illegible]	12%
Open Issue	[illegible]	10%
Open Issue	[illegible]	9%
Open Issue	[illegible]	9%

Figure 5.9: Prediction Model Results Example

5. **PredictIssueTypes** This function predicts potential issue types for new defect tickets.

5.7 Chapter Summary

In this study, we performed the case study research in Dialog IT company to leverage the performance of the legacy bug triage system. One of Australia's top technology service providers, Dialog IT Company, operates out of locations in Brisbane, Sydney, Canberra, Melbourne, Adelaide, Perth, and Darwin. The company provides a wide range of innovative IT solutions and long-term operational support to clients across all industries, including all levels of government. Thus, the bug triage process plays an integral role in managing client requests timely. However, the company is currently using a manual bug triage process as it is time-consuming and error-prone. In addition, the company uses diverse issue tracking platforms to manage the issue request for various projects as it hinders the communication between technical consultants from different projects to share bug resolution knowledge. This study conjured the artificial intelligence (AI) based centralized bug triage process model to leverage the bug triage process. The main objectives of this research are to design and develop the automatic bug triage prototype tool to evaluate the impact of the new process model design effectively. Based on the findings of this research, the application of the AI-based bug triage model performance is promising for company projects. In addition, the prototype tool has room to expand to meet more business requirements in the future.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we have presented a set of empirical studies to leverage the automation of software maintenance activities such as change impact analysis (CIA) and bug triage process. First and foremost, to extract and analyze methodologies offered in earlier studies and identify the research gap, we conducted a systematic review study on the automatic trace link recovery strategy in the field of software change effect analysis. Second, we proposed an interactive hierarchical trace map (HTM) visualization technique to perform change impact analysis for multiple software artifacts at the same time. Lastly, we presented a multi-triage bug triage model to leverage developer assignment and issue type allocation to a new issue report. We evaluated our approaches with a wide range of open source projects from various domains to replicate real-world applications.

In a systematic review study, we reviewed the peer-review articles which are published between 2012 and 2019 and found 33 relevant studies in this review. We reviewed the investigations based on CIA-related characteristics, outlining traceability strategies, CIA coverage levels, trace directions, and techniques for creating trace links between various types of artifacts. There has been less research work on recovering the links between design and test case artifacts, presumably due to the scarcity of data sets. Furthermore, most works focused on leveraging the automating process of constructing the links between artifacts, but few studies are interested in improving the presentation parts, which hinders the usage of traceability links. Visualizing the trace links intuitively and narratively plays an important role in increasing the performance of the change impact analysis process. Thus, in the next phase of the thesis, we proposed the traceability visualization technique to leverage the change impact process.

In the HTM study, we conjured the hierarchical trace map visualization technique, which applies the vector space model (VSM) approach, to construct interrelationships between multiple software artifacts. The HTM visualization approach aims to assist software comprehension and change impact analysis activities by providing a visual trace links exploration space. In our approach, we presented a standalone application, which includes the functionality to automatically consume data from external data sources to recover the trace links between high and low-level artifacts. We evaluated our approach with the common trace links experimental project and presented the results.

Finally, in the multi-triage study, we presented the novel multitask learning-based bug triage model to classify relevant developers and issue types for a new issue report by analyzing the contextual and structural information of historical issue reports. Multi-triage is the first triage model to explore the effects of synthetic bug reports generated with a contextual data augmentation approach. We designed and trained our model incrementally by performing analysis and a time-series-based cross-fold validation approach. By using the two encoders, the contextual and abstract syntax tree (AST) structure of an issue report'is learned separately and concatenated at the end of the learning representation phase as the model can detect the full representation of the issue report precisely. Also, the multi-triage model can train faster than two single-task learning as well as the accuracy is noticeably higher than the state-of-the-art approaches.

6.2 Threats to Validity

We briefly discuss the main threats to the validity of our study with respect to internal validity and external validity.

Internal validity refers to the possibility of error in creating the data set used for the evaluation within the context of a particular study. In both our experimental and case studies, the class labels were extracted from the historical bug reports to generate the ground truth dataset of developer and issue type labels. However, there is a risk that these labels can either be incorrectly labeled by bug triagers or outdated as the project evolves. For example, developers A and B have fixed product X-related issues, but developer B resolved most of the recent issues. Therefore, assigning the new product-X-related case to developer B is more applicable based on the order of bug resolution history. Although we took the risk of incorrect labels, we minimized the chance of assigning outdated labels using a time-series-based 5-fold cross-validation approach. Performing the model training based on time can increase the likelihood of predicting the most relevant class labels based on the data sequence.

External validity refers to how our study's results can be generalizable to the variability of the bug-tracking system used in industrial and open-source projects. In our experimental study, we solely focused on open-source projects from various domains, which provided all the necessary information to conduct this study. However, the exploratory analysis of our approach to various open-source projects has led to several rich observations; the risk of external validity arises when we generalize our findings to other situations, including industrial software projects. Therefore, we perform a case study in one commercial software company to justify our assumption. Although the field study was conducted in a company with real-world data, replicating our approach to more commercial projects is required to raise empirical evidence.

6.3 Future Work

Automating the bug triage process using deep learning techniques has become emerging research due to the growing demand for managing rapidly evolving software requirements in fast-paced industries. In general, existing bug triage approaches mainly fall into two categories: the algebraic model-based approach [137, 155] and the statistical language model-based approach [12, 102, 157]. Both approaches train both developers and issue-type prediction tasks with a single-task learning model. Studies have used the terms frequency (TF) and inverse document frequency (IDF) as the term's weighting factor in algebraic models. Various distance calculation algorithms (e.g., Euclidean distance) are used to calculate the distance between two issue reports and construct links between a new issue report and relevant developers or issue types by matching existing issue reports.

Compared to a vast number of single-task learning model-based bug triage automation works, only a handful of research exploited the multi-task learning model, and continuous distributed vector representations of code [9, 16]. Also, there are a few empirical studies that focus on exploring the effects of visual bug information in a bug triage context. The majority of users who report software bugs are non-technical users, as it is tedious and error-prone to describe the bug description in detail to reproduce the errors. Even for developers, it is time-consuming and unclear to reproduce the errors by following a reproduction step. Therefore, more and more users are beginning to provide a screenshot or animated images to supplement the information in a bug report to visualize and replicate the issue.

In reality, a human triage used all the available information to comprehend the bug report as the learning representation of the bug report taking into account all the information, including supplement information, to precisely comprehend the requirements. More importantly, our multitask learning model-based bug triage approach opens up the concepts of training multiple tasks in a single model [16]. We have enabled our model's joint representation process layer to take advantage of the training time since both developers and issue-type recommendation models have relied on issue report representation. Furthermore, there is emerging research in the area of extracting textual information from images, and the outcomes of this research are promising as well [42, 79, 88]. It is hoped that the ideas proposed in this thesis (e.g., multi-triage model, visualizing trace links) could provide some inspiration to leverage software maintenance activities such as bug triage and change impact analysis.

Bibliography

[1] H. Abukwaik, A. Burger, B. K. Andam, and T. Berger, *Semi-automated feature traceability with embedded annotations*, in 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2018, pp. 529–533.

[2] A. Achille and S. Soatto, *Information dropout: Learning optimal representations through noisy computation*, IEEE transactions on pattern analysis and machine intelligence, 40 (2018), pp. 2897–2905.

[3] R. Agarwal, R. R. Chetwani, M. Ravindra, and K. M. Bharadwaj, *Novel methodology for requirements to design traceability of onboard software*, in 2014 International Conference on Advances in Electronics Computers and Communications, 2014, pp. 1–6.

[4] H. Alaeddine and M. Jihene, *Deep network in network*, Neural Computing and Applications, 33 (2021), pp. 1453–1465.

[5] N. Ali, H. Cai, A. Hamou-Lhadj, and J. Hassine, *Exploiting parts-of-speech for effective automated requirements traceability*, Information and Software Technology, 106 (2019), pp. 126–141.

[6] N. Ali, Y. Gueheneuc, and G. Antoniol, *Trustrace: Mining software repositories to improve the accuracy of requirement traceability links*, IEEE Transactions on Software Engineering, 39 (2013), pp. 725–741.

[7] N. Ali, F. Jaafar, and A. E. Hassan, *Leveraging historical co-change information for requirements traceability*, in 2013 20th Working Conference on Reverse Engineering (WCRE), 2013, pp. 361–370.

[8] N. Ali, Z. Sharafl, Y.-G. Gueheneuc, and G. Antoniol, *An empirical study on requirements traceability using eye-tracking*, in 2012 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 191–200.

[9] U. ALON, M. ZILBERSTEIN, O. LEVY, AND E. YAHAV, *Code2vec: Learning distributed representations of code*, Proc. ACM Program. Lang., 3 (2019).

[10] J. M. ALONSO-ABAD, C. LÓPEZ-NOZAL, J. M. MAUDES-RAEDO, AND R. MARTICORENA-SÁNCHEZ, *Label prediction on issue tracking systems using text mining*, Progress in Artificial Intelligence, 8 (2019), pp. 325–342.

[11] G. ANTONIOL, G. CANFORA, G. CASAZZA, A. DE LUCIA, AND E. MERLO, *Recovering traceability links between code and documentation*, IEEE transactions on software engineering, 28 (2002), pp. 970–983.

[12] J. ANVIK, L. HIEW, AND G. C. MURPHY, *Who should fix this bug?*, in Proceedings of the 28th international conference on Software engineering, 2006, pp. 361–370.

[13] J. ANVIK AND G. C. MURPHY, *Reducing the effort of bug report triage: Recommenders for development-oriented decisions*, ACM Transactions on Software Engineering and Methodology (TOSEM), 20 (2011), pp. 1–35.

[14] T. W. W. AUNG, H. HUO, AND Y. SUI, *Interactive traceability links visualization using hierarchical trace map*, in 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2019, pp. 367–369.

[15] ——, *A Literature Review of Automatic Traceability Links Recovery for Software Change Impact Analysis*, Association for Computing Machinery, New York, NY, USA, 2020, p. 14,Äì24.

[16] T. W. W. AUNG, Y. WAN, H. HUO, AND Y. SUI, *Multi-triage: A multi-task learning framework for bug triage*, Journal of Systems and Software, 184 (2022), p. 111133.

[17] T. W. W. A. AUNG, *Multitriage*, 2021.

[18] I. BANERJEE, Y. LING, M. C. CHEN, S. A. HASAN, C. P. LANGLOTZ, N. MORADZADEH, B. CHAPMAN, T. AMRHEIN, D. MONG, D. L. RUBIN, O. FARRI, AND M. P. LUNGREN, *Comparative effectiveness of convolutional neural network (cnn) and recurrent neural network (rnn) architectures for radiology text report classification*, Artificial Intelligence in Medicine, 97 (2019), pp. 79 – 88.

[19] G. BAVOTA, L. COLANGELO, A. DE LUCIA, S. FUSCO, R. OLIVETO, AND A. PANICHELLA, *Traceme: Traceability management in eclipse*, in 2012 28th IEEE International Conference on Software Maintenance (ICSM), Sep. 2012, pp. 642–645.

[20] E. BEN CHARRADA, A. KOZIOLEK, AND M. GLINZ, *Identifying outdated requirements based on source code changes*, in 2012 20th IEEE International Requirements Engineering Conference (RE), Sep. 2012, pp. 61–70.

[21] Y. BENGIO, *Practical recommendations for gradient-based training of deep architectures*, Arxiv, (2012).

[22] C. BERGMEIR AND J. M. BENITEZ, *On the use of cross-validation for time series predictor evaluation*, Information Sciences, 191 (2012), pp. 192–213.
Data Mining for Software Trustworthiness.

[23] P. BERTA, M. BYSTRICKỲ, M. KREMPASKỲ, AND V. VRANIĆ, *Employing issues and commits for in-code sentence based use case identification and remodularization*, in Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems, ACM, 2017, p. 1.

[24] P. BHATTACHARYA AND I. NEAMTIU, *Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging*, in 2010 IEEE International Conference on Software Maintenance, IEEE, 2010, pp. 1–10.

[25] A. BIEDENKAPP, M. LINDAUER, K. EGGENSPERGER, F. HUTTER, C. FAWCETT, AND H. HOOS, *Efficient parameter importance analysis via ablation with surrogates*, in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 31, 2017.

[26] H. BILEN AND A. VEDALDI, *Integrated perception with recurrent multi-task neural networks*, in Advances in neural information processing systems, 2016, pp. 235–243.

[27] BOHNER, *Impact analysis in the software change process: a year 2000 perspective*, in 1996 Proceedings of International Conference on Software Maintenance, Nov 1996, pp. 42–51.

[28] M. BORG, *Embrace your issues: compassing the software engineering landscape using bug reports*, in Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, ACM, 2014, pp. 891–894.

[29] M. BORG, P. RUNESON, AND A. ARDÖ, *Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability*, Empirical Software Engineering, 19 (2014), pp. 1565–1616.

[30] J. CABOT, J. L. C. IZQUIERDO, V. COSENTINO, AND B. ROLANDI, *Exploring the use of labels to categorize issues in open-source software projects*, in 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, 2015, pp. 550–554.

[31] L. CAI, S. ZHOU, X. YAN, AND R. YUAN, *A stacked bilstm neural network based on coattention mechanism for question answering*, Computational intelligence and neuroscience, 2019 (2019).

[32] R. CARUANA, *Multitask learning: A knowledge-based source of inductive bias*, in Proceedings of the Tenth International Conference on Machine Learning, Morgan Kaufmann, 1993, pp. 41–48.

[33] G. CATOLINO, F. PALOMBA, A. ZAIDMAN, AND F. FERRUCCI, *Not all bugs are the same: Understanding, characterizing, and classifying bug types*, Journal of Systems and Software, 152 (2019), pp. 165–181.

[34] O. CHAPARRO, J. M. FLOREZ, AND A. MARCUS, *Using observed behavior to reformulate queries during text retrieval-based bug localization*, in 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2017, pp. 376–387.

[35] X. CHEN, J. HOSKING, AND J. GRUNDY, *Visualizing traceability links between source code and documentation*, in 2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Sep. 2012, pp. 119–126.

[36] X. CHENG, H. WANG, J. HUA, G. XU, AND Y. SUI, *Deepwukong: Statically detecting software vulnerabilities using deep graph neural network*, 30 (2021).

[37] J. CLELAND-HUANG, O. GOTEL, A. ZISMAN, ET AL., *Software and systems traceability*, vol. 2, Springer, 2012.

[38] J. CLELAND-HUANG AND J. GUO, *Towards more intelligent trace retrieval algorithms*, in Proceedings of the 3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE 2014, New York, NY, USA, 2014, Association for Computing Machinery, p. 1‚Äì6.

[39] J. Cleland-Huang and R. Habrat, *Visual support in automated tracing*, in Second International Workshop on Requirements Engineering Visualization (REV 2007), Oct 2007, pp. 4–4.

[40] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, *Natural language processing (almost) from scratch*, Journal of machine learning research, 12 (2011), pp. 2493–2537.

[41] O. contributors, *Documentation and bug triage day*, 2014.

[42] 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 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, 2021, pp. 957–969.

[43] V. Csuvik, A. Kicsi, and L. Vidacs, *Source code level word embeddings in aiding semantic test-to-code traceability*, in 2019 IEEE/ACM 10th International Symposium on Software and Systems Traceability (SST), May 2019, pp. 29–36.

[44] B. Dagenais and L. Hendren, *Enabling static analysis for partial java programs*, SIGPLAN Not., 43 (2008), p. 313,Äì328.

[45] A. De Lucia, F. Fasano, and R. Oliveto, *Traceability management for impact analysis*, in 2008 Frontiers of Software Maintenance, Sep. 2008, pp. 21–30.

[46] A. De Lucia, A. Marcus, R. Oliveto, and D. Poshyvanyk, *Information retrieval methods for automated traceability recovery*, in Software and systems traceability, Springer, 2012, pp. 71–98.

[47] A. De Lucia, R. Oliveto, and G. Tortora, *Adams re-trace: Traceability link recovery via latent semantic indexing*, in Proceedings of the 30th International Conference on Software Engineering, ICSE '08, New York, NY, USA, 2008, ACM, pp. 839–842.

[48] Y. Deng, Y. Xie, Y. Li, M. Yang, N. Du, W. Fan, K. Lei, and Y. Shen, *Multitask learning with multi-view attention for answer selection and knowledge base question answering*, in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, 2019, pp. 6318–6325.

[49] J. Deshmukh, S. Podder, S. Sengupta, N. Dubash, et al., *Towards accurate duplicate bug retrieval using deep learning techniques*, in 2017 IEEE International conference on software maintenance and evolution (ICSME), IEEE, 2017, pp. 115–124.

[50] W. Dominik and L. Udo, *Manage interdisciplinarity based on requirements traceability: A graph-based tool support for requirements traceability*, in 2017 Portland International Conference on Management of Engineering and Technology (PICMET), IEEE, 2017, pp. 1–8.

[51] B. Dowdeswell, R. Sinha, and E. Haemmerle, *Torus: Tracing complex requirements for large cyber-physical systems*, in 2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS), 2016, pp. 23–32.

[52] N. Dvornik, K. Shmelkov, J. Mairal, and C. Schmid, *Blitznet: A real-time deep network for scene understanding*, in Proceedings of the IEEE international conference on computer vision, 2017, pp. 4154–4162.

[53] R. Elamin and R. Osman, *Implementing traceability repositories as graph databases for software quality improvement*, in 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), IEEE, 2018, pp. 269–276.

[54] D. Falessi, M. Di Penta, G. Canfora, and G. Cantone, *Estimating the number of remaining links in traceability recovery*, Empirical Software Engineering, 22 (2017), pp. 996–1027.

[55] D. Falessi, J. Roll, J. L. Guo, and J. Cleland-Huang, *Leveraging historical associations between requirements and source code to identify impacted classes*, IEEE Transactions on Software Engineering, (2018).

[56] C. Fawcett and H. H. Hoos, *Analysing differences between algorithm configurations through ablation*, Journal of Heuristics, 22 (2016), pp. 431–458.

[57] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, *Integrated impact analysis for managing software changes*, in 2012 34th International Conference on Software Engineering (ICSE), June 2012, pp. 430–440.

[58] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, *On integrating orthogonal information retrieval methods to improve traceability recovery*, in

2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 133–142.

[59] A. GHANNEM, M. S. HAMDI, M. KESSENTINI, AND H. H. AMMAR, *Search-based requirements traceability recovery: A multi-objective approach*, in 2017 IEEE Congress on Evolutionary Computation (CEC), IEEE, 2017, pp. 1183–1190.

[60] R. GHARIBI, A. H. RASEKH, M. H. SADREDDINI, AND S. M. FAKHRAHMAD, *Leveraging textual properties of bug reports to localize relevant source files*, Information Processing & Management, 54 (2018), pp. 1058–1076.

[61] M. GOLZADEH, D. LEGAY, A. DECAN, AND T. MENS, *Bot or not? detecting bots in github pull request activity based on comment similarity*, in Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, 2020, pp. 31–35.

[62] O. C. GOTEL AND C. FINKELSTEIN, *An analysis of the requirements traceability problem*, in Proceedings of IEEE International Conference on Requirements Engineering, IEEE, 1994, pp. 94–101.

[63] G. GOUSIOS, M. PINZGER, AND A. V. DEURSEN, *An exploratory study of the pull-based software development model*, in Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 345–355.

[64] A. GRAVES AND J. SCHMIDHUBER, *Framewise phoneme classification with bidirectional lstm and other neural network architectures*, Neural Networks, 18 (2005), pp. 602 – 610.
IJCNN 2005.

[65] J. GUO, J. CHENG, AND J. CLELAND-HUANG, *Semantically enhanced software traceability using deep learning techniques*, in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 3–14.

[66] J. GUO, M. GIBIEC, AND J. CLELAND-HUANG, *Tackling the term-mismatch problem in automated trace retrieval*, Empirical Software Engineering, 22 (2017), pp. 1103–1142.

[67] S. HAIDRAR, A. ANWAR, AND O. ROUDIES, *Towards a generic framework for requirements traceability management for sysml language*, in 2016 4th IEEE

International Colloquium on Information Science and Technology (CiSt), 2016, pp. 210–215.

[68] ——, *A sysml-based approach to manage stakeholder requirements traceability*, in 2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA), 2017, pp. 202–207.

[69] Z. Han, X. Li, Z. Xing, H. Liu, and Z. Feng, *Learning to predict severity of software vulnerability using only vulnerability description*, in 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2017, pp. 125–136.

[70] J. Hayes, A. Dekhtyar, S. Sundaram, and S. Howard, *Helping analysts trace requirements: an objective look*, in Proceedings. 12th IEEE International Requirements Engineering Conference, 2004., 2004, pp. 249–259.

[71] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, *Advancing candidate link generation for requirements tracing: The study of methods*, IEEE Transactions on Software Engineering, 32 (2006), pp. 4–19.

[72] J. H. Hayes, A. Dekhtyar, S. K. Sundaram, E. A. Holbrook, S. Vadlamudi, and A. April, *Requirements tracing on target (retro): improving software maintenance through traceability recovery*, Innovations in Systems and Software Engineering, 3 (2007), pp. 193–202.

[73] A. R. Hevner, S. T. March, J. Park, and S. Ram, *Design science in information systems research*, MIS quarterly, (2004), pp. 75–105.

[74] M. A. Javed and U. Zdun, *A systematic literature review of traceability approaches between software architecture and source code*, in Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, ACM, 2014, p. 16.

[75] G. Jiang and W. Wang, *Error estimation based on variance analysis of k-fold cross-validation*, Pattern Recognition, 69 (2017), pp. 94–106.

[76] J. Jiang, Q. Wu, J. Cao, X. Xia, and L. Zhang, *Recommending tags for pull requests in github*, Information and Software Technology, (2020), p. 106394.

[77] K. Kafle, M. Yousefhussien, and C. Kanan, *Data augmentation for visual question answering*, in Proceedings of the 10th International Conference on Natural Language Generation, 2017, pp. 198–202.

[78] M. R. Karim, G. Ruhe, M. M. Rahman, V. Garousi, and T. Zimmermann, *An empirical investigation of single‚Äêobjective and multiobjective evolutionary algorithms for developer's assignment to bugs*, Journal of Software: Evolution and Process, 28 (2016), pp. 1025 – 1060.

[79] C. Kaundilya, D. Chawla, and Y. Chopra, *Automated text extraction from images using ocr system*, in 2019 6th International Conference on Computing for Sustainable Global Development (INDIACom), 2019, pp. 145–150.

[80] E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. Maletic, J. H. Hayes, A. Dekhtyar, D. Manukian, S. Hossein, and D. Hearn, *Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions*, in 2012 34th International Conference on Software Engineering (ICSE), 2012, pp. 1375–1378.

[81] J. D. M.-W. C. Kenton and L. K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, in Proceedings of NAACL-HLT, 2019, pp. 4171–4186.

[82] S. Kim, J. Zhao, Y. Tian, and S. Chandra, *Code prediction by feeding trees to transformers*, in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 150–162.

[83] Y. Kim, *Convolutional neural networks for sentence classification*, in Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, Oct. 2014, Association for Computational Linguistics, pp. 1746–1751.

[84] B. Kitchenham and S. Charters, *Guidelines for performing systematic literature reviews in software engineering*, 2007.

[85] S. Klock, M. Gethers, B. Dit, and D. Poshyvanyk, *Traceclipse: an eclipse plug-in for traceability link recovery and management*, in Proceedings of the 6th international workshop on traceability in emerging forms of software engineering, ACM, 2011, pp. 24–30.

[86] I. KOKKINOS, *Ubernet: Training a universal convolutional neural network for low-, mid-, and high-level vision using diverse datasets and limited memory*, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 6129–6138.

[87] R. KOPPLER, *A systematic approach to fuzzy parsing*, Software: Practice and Experience, 27 (1997), pp. 637–649.

[88] D. KOTTACHCHI AND T. GINIGE, *Slide hatch: Smart slide generator*, in 2021 2nd Global Conference for Advancement in Technology (GCAT), 2021, pp. 1–5.

[89] H. KUANG, H. GAO, H. HU, X. MA, J. LÜ, P. MÄDER, AND A. EGYED, *Using frugal user feedback with closeness analysis on code to improve ir-based traceability recovery*, in Proceedings of the 27th International Conference on Program Comprehension, ICPC ,Äô19, IEEE Press, 2019, p. 369,Äì379.

[90] T.-D. B. LE, M. LINARES-VASQUEZ, D. LO, AND D. POSHYVANYK, *Rclinker: Automated linking of issue reports and commits leveraging rich contextual information*, in 2015 IEEE 23rd International Conference on Program Comprehension, 2015, pp. 36–47.

[91] H. LEE AND H. KWON, *Going deeper with contextual cnn for hyperspectral image classification*, IEEE Transactions on Image Processing, 26 (2017), pp. 4843–4855.

[92] S.-R. LEE, M.-J. HEO, C.-G. LEE, M. KIM, AND G. JEONG, *Applying deep learning based automatic bug triager to industrial projects*, in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 926–931.

[93] W.-T. LEE, *Dependency link derivation process and theorems of requirements traceability matrix*, in 2016 International Computer Symposium (ICS), 2016, pp. 561–566.

[94] X. LIANG, A. JIANG, T. LI, Y. XUE, AND G. WANG, *Lr-smote–an improved unbalanced data set oversampling based on k-means and svm*, Knowledge-Based Systems, (2020), p. 105845.

[95] G. LIU AND J. GUO, *Bidirectional lstm with attention mechanism and convolutional layer for text classification*, Neurocomputing, 337 (2019), pp. 325–338.

[96] X. Liu, J. Gao, X. He, L. Deng, K. Duh, and Y.-Y. Wang, *Representation learning using multi-task deep neural networks for semantic classification and information retrieval*, (2015).

[97] Y. Lu, A. Kumar, S. Zhai, Y. Cheng, T. Javidi, and R. Feris, *Fully-adaptive feature sharing in multi-task networks with applications in person attribute classification*, in Proceedings of the IEEE conference on computer vision and pattern recognition, 2017, pp. 5334–5343.

[98] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, *Recovering traceability links in software artifact management systems using information retrieval methods*, ACM Transactions on Software Engineering and Methodology (TOSEM), 16 (2007), p. 13.

[99] K. Mahmood, H. Takahashi, and M. Alobaidi, *A semantic approach for traceability link recovery in aerospace requirements management system*, in 2015 IEEE Twelfth International Symposium on Autonomous Decentralized Systems, March 2015, pp. 217–222.

[100] A. Mahmoud and N. Niu, *Supporting requirements traceability through refactoring*, in 2013 21st IEEE International Requirements Engineering Conference (RE), 2013, pp. 32–41.

[101] A. Mahmoud and G. Williams, *Detecting, classifying, and tracing non-functional software requirements*, Requirements Engineering, 21 (2016), pp. 357–381.

[102] S. Mani, A. Sankaran, and R. Aralikatte, *Deeptriage: Exploring the effectiveness of deep learning for bug triaging*, in Proceedings of the ACM India Joint International Conference on Data Science and Management of Data, 2019, pp. 171–179.

[103] A. Marcus, X. Xie, and D. Poshyvanyk, *When and how to visualize traceability links?*, in Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering, 2005, pp. 56–61.

[104] D. Matter, A. Kuhn, and O. Nierstrasz, *Assigning bug reports using a vocabulary-based expertise model of developers*, in 2009 6th IEEE international working conference on mining software repositories, IEEE, 2009, pp. 131–140.

[105] T. MERTEN, D. JUPPNER, AND A. DELATER, *Improved representation of traceability links in requirements engineering knowledge using sunburst and netmap visualizations*, in 2011 4th International Workshop on Managing Requirements Knowledge, Aug 2011, pp. 17–21.

[106] T. MERTEN, D. KRÄMER, B. MAGER, P. SCHELL, S. BÜRSNER, AND B. PAECH, *Do information retrieval algorithms for automated traceability perform effectively on issue tracking system data?*, in International Working Conference on Requirements Engineering: Foundation for Software Quality, Springer, 2016, pp. 45–62.

[107] C. MILLS, J. ESCOBAR-AVILA, AND S. HAIDUC, *Automatic traceability maintenance via machine learning classification*, in 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Sep. 2018, pp. 369–380.

[108] C. MILLS AND S. HAIDUC, *The impact of retrieval direction on ir-based traceability link recovery*, in 2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER), IEEE, 2017, pp. 51–54.

[109] ——, *A machine learning approach for determining the validity of traceability links*, in 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), 2017, pp. 121–123.

[110] L. MOONEN, *Generating robust parsers using island grammars*, in Proceedings Eighth Working Conference on Reverse Engineering, IEEE, 2001, pp. 13–22.

[111] G. MURPHY AND D. CUBRANIC, *Automatic bug triage using text categorization*, in Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering, Citeseer, 2004, pp. 1–6.

[112] H. T. MUSTAFA, J. YANG, AND M. ZAREAPOOR, *Multi-scale convolutional neural network for multi-focus image fusion*, Image and Vision Computing, 85 (2019), pp. 26 – 35.

[113] N. MUSTAFA AND Y. LABICHE, *The need for traceability in heterogeneous systems: A systematic literature review*, in 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), vol. 1, IEEE, 2017, pp. 305–310.

[114] V. NAIR AND G. E. HINTON, *Rectified linear units improve restricted boltzmann machines*, in Proceedings of the 27th International Conference on International Conference on Machine Learning, ICML'10, USA, 2010, Omnipress, pp. 807–814.

[115] M. D. NETWORK AND INDIVIDUAL CONTRIBUTORS, *Bugdays / bug-triage*, 2018.

[116] A. NIGAM, B. NIGAM, C. BHAISARE, AND N. ARYA, *Classifying the bugs using multi-class semi supervised support vector machine*, in International Conference on Pattern Recognition, Informatics and Medical Engineering (PRIME-2012), March 2012, pp. 393–397.

[117] K. NISHIKAWA, H. WASHIZAKI, Y. FUKAZAWA, K. OSHIMA, AND R. MIBE, *Recovering transitive traceability links among software artifacts*, in 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2015, pp. 576–580.

[118] N. NIU, T. BHOWMIK, H. LIU, AND Z. NIU, *Traceability-enabled refactoring for managing just-in-time requirements*, in 2014 IEEE 22nd International Requirements Engineering Conference (RE), 2014, pp. 133–142.

[119] N. NIU AND A. MAHMOUD, *Enhancing candidate link generation for requirements tracing: The cluster hypothesis revisited*, in 2012 20th IEEE International Requirements Engineering Conference (RE), Sep. 2012, pp. 81–90.

[120] A. NOYER, P. IYENGHAR, E. PULVERMUELLER, F. PRAMME, AND G. BIKKER, *Traceability and interfacing between requirements engineering and uml domains using the standardized reqif format*, in 2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), IEEE, 2015, pp. 1–6.

[121] I. OZKAYA AND O. AKIN, *Tool support for computer-aided requirement traceability in architectural design: The case of designtrack*, Automation in construction, 16 (2007), pp. 674–684.

[122] A. PANICHELLA, B. DIT, R. OLIVETO, M. DI PENTA, D. POSHYVANYK, AND A. DE LUCIA, *How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms*, in Proceedings of the 2013 International Conference on Software Engineering, ICSE ‚Äô13, IEEE Press, 2013, p. 522‚Äì531.

[123] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia, *When and how using structural information to improve ir-based traceability recovery*, in 2013 17th European Conference on Software Maintenance and Reengineering, March 2013, pp. 199–208.

[124] J.-w. Park, M.-W. Lee, J. Kim, S.-w. Hwang, and S. Kim, *Cost-aware triage ranking algorithms for bug reporting systems*, Knowledge and Information Systems, 48 (2016), pp. 679–705.

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

[126] K. Pohl, *Requirements engineering : fundamentals, principles, and techniques*, Springer, Heidelberg ;, 2010.

[127] K. Pohl and C. Rupp, *Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam - Foundation Level - IREB Compliant*, Rocky Nook, 1st ed., 2011.

[128] S. Polisetty, A. Miranskyy, and A. Başar, *On usefulness of the deep-learning-based bug localization models to practitioners*, in Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering, ACM, 2019, pp. 16–25.

[129] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, *Language models are unsupervised multitask learners*, OpenAI blog, 1 (2019), p. 9.

[130] M. Rahimi and J. Cleland-Huang, *Evolving software trace links between requirements and source code*, Empirical Software Engineering, 23 (2018), pp. 2198–2231.

[131] M. Rahimi, W. Goss, and J. Cleland-Huang, *Evolving requirements-to-code trace links across versions of a software system*, in 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), Oct 2016, pp. 99–109.

[132] B. Ramesh and M. Jarke, *Toward reference models for requirements traceability*, IEEE Transactions on Software Engineering, 27 (2001), pp. 58–93.

[133] M. Rath, D. Lo, and P. Mäder, *Analyzing requirements and traceability information to improve bug localization*, in 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), IEEE, 2018, pp. 442–453.

[134] M. Rath, J. Rendall, J. L. Guo, J. Cleland-Huang, and P. Mäder, *Traceability in the wild: automatically augmenting incomplete trace links*, in 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 834–845.

[135] P. Rempel and P. Mader, *A quality model for the systematic assessment of requirements traceability*, in 2015 IEEE 23rd International Requirements Engineering Conference (RE), 2015, pp. 176–185.

[136] H. Ruan, B. Chen, X. Peng, and W. Zhao, *Deeplink: Recovering issue-commit links based on deep learning*, Journal of Systems and Software, 158 (2019), p. 110406.

[137] P. Runeson, M. Alexandersson, and O. Nyholm, *Detection of duplicate defect reports using natural language processing*, in 29th International Conference on Software Engineering (ICSE'07), IEEE, 2007, pp. 499–510.

[138] M. Seiler, P. Hubner, and B. Paech, *Comparing traceability through information retrieval, commits, interaction logs, and tags*, in 2019 IEEE/ACM 10th International Symposium on Software and Systems Traceability (SST), May 2019, pp. 21–28.

[139] Y. Shinohara, *Adversarial multi-task learning of deep neural networks for robust speech recognition.*, in Interspeech, San Francisco, CA, USA, 2016, pp. 2369–2372.

[140] T. Standley, A. R. Zamir, D. Chen, L. Guibas, J. Malik, and S. Savarese, *Which tasks should be learned together in multi-task learning?*, 2020.

[141] Y. Sui, X. Cheng, G. Zhang, and H. Wang, *Flow2vec: Value-flow-based precise code embedding*, Proc. ACM Program. Lang., 4 (2020).

[142] A. TAMRAWI, T. T. NGUYEN, J. M. AL-KOFAHI, AND T. N. NGUYEN, *Fuzzy set and cache-based approach for bug triaging*, in Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, New York, NY, USA, 2011, Association for Computing Machinery, p. 365,Äì375.

[143] R. TSUCHIYA, H. WASHIZAKI, Y. FUKAZAWA, K. OSHIMA, AND R. MIBE, *Interactive recovery of requirements traceability links using user feedback and configuration management logs*, in International Conference on Advanced Information Systems Engineering, Springer, 2015, pp. 247–262.

[144] M. TUFANO, C. WATSON, G. BAVOTA, M. DI PENTA, M. WHITE, AND D. POSHYVANYK, *Deep learning similarities from different representations of source code*, in Proceedings of the 15th International Conference on Mining Software Repositories, MSR '18, New York, NY, USA, 2018, ACM, pp. 542–553.

[145] M. UNTERKALMSTEINER, T. GORSCHEK, R. FELDT, AND N. LAVESSON, *Large-scale information retrieval in software engineering-an experience report from industrial application*, Empirical Software Engineering, 21 (2016), pp. 2324–2365.

[146] T. VALE AND E. S. DE ALMEIDA, *Experimenting with information retrieval methods in the recovery of feature-code spl traces*, Empirical Software Engineering, 24 (2019), pp. 1328–1368.

[147] Y. WAN, Z. ZHAO, M. YANG, G. XU, H. YING, J. WU, AND P. S. YU, *Improving automatic source code summarization via deep reinforcement learning*, in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, 2018, pp. 397–407.

[148] W. WANG, A. GUPTA, N. NIU, L. DA XU, J.-R. C. CHENG, AND Z. NIU, *Automatically tracing dependability requirements via term-based relevance feedback*, IEEE Transactions on Industrial Informatics, 14 (2016), pp. 342–349.

[149] W. WANG, N. NIU, H. LIU, AND Z. NIU, *Enhancing automated requirements traceability by resolving polysemy*, in 2018 IEEE 26th International Requirements Engineering Conference (RE), IEEE, 2018, pp. 40–51.

[150] M. WHITE, M. TUFANO, M. MARTINEZ, M. MONPERRUS, AND D. POSHYVANYK, *Sorting and transforming program repair ingredients via deep learning code*

similarities, in 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2019, pp. 479–490.

[151] T. WOLFENSTETTER, K. FULLER, M. BOHM, H. KRCMAR, AND S. BRUNDL, *Towards a requirements traceability reference model for product service systems*, in 2015 International Conference on Industrial Engineering and Systems Management (IESM), 2015, pp. 1213–1220.

[152] S. XI, Y. YAO, X. XIAO, F. XU, AND J. LU, *An effective approach for routing the bug reports to the right fixers*, in Proceedings of the Tenth Asia-Pacific Symposium on Internetware, 2018, pp. 1–10.

[153] S.-Q. XI, Y. YAO, X.-S. XIAO, F. XU, AND J. LV, *Bug triaging based on tossing sequence modeling*, Journal of Computer Science and Technology, 34 (2019), pp. 942–956.

[154] X. XIA, Y. FENG, D. LO, Z. CHEN, AND X. WANG, *Towards more accurate multi-label software behavior learning*, in 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), IEEE, 2014, pp. 134–143.

[155] X. XIA, D. LO, Y. DING, J. M. AL-KOFAHI, T. N. NGUYEN, AND X. WANG, *Improving automated bug triaging with specialized topic model*, IEEE Transactions on Software Engineering, 43 (2017), pp. 272–297.

[156] X. XIA, D. LO, X. WANG, AND B. ZHOU, *Tag recommendation in software information sites*, in 2013 10th Working Conference on Mining Software Repositories (MSR), IEEE, 2013, pp. 287–296.

[157] Q. XIE, Z. WEN, J. ZHU, C. GAO, AND Z. ZHENG, *Detecting duplicate bug reports with convolutional neural networks*, in 2018 25th Asia-Pacific Software Engineering Conference (APSEC), 2018, pp. 416–425.

[158] J. XUAN, H. JIANG, Z. REN, AND W. ZOU, *Developer prioritization in bug repositories*, in 2012 34th International Conference on Software Engineering (ICSE), 2012, pp. 25–35.

[159] A. YADAV, S. K. SINGH, AND J. S. SURI, *Ranking of software developers based on expertise score for bug triaging*, Information and Software Technology, 112 (2019), pp. 1–17.

[160] K. C. YOUM, J. AHN, J. KIM, AND E. LEE, *Bug localization based on code change histories and bug reports*, in 2015 Asia-Pacific Software Engineering Conference (APSEC), Dec 2015, pp. 190–197.

[161] S. YU, L. XU, Y. ZHANG, J. WU, Z. LIAO, AND Y. LI, *Nbsl: A supervised classification model of pull request in github*, in 2018 IEEE International Conference on Communications (ICC), May 2018, pp. 1–6.

[162] Y. YU, H. WANG, G. YIN, AND T. WANG, *Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment?*, Information and Software Technology, 74 (2016), pp. 204 – 218.

[163] T. ZHANG, J. CHEN, G. YANG, B. LEE, AND X. LUO, *Towards more accurate severity prediction and fixer recommendation of software bugs*, Journal of Systems and Software, 117 (2016), pp. 166–184.

[164] T. ZHANG AND B. LEE, *An automated bug triage approach: A concept profile and social network based developer recommendation*, in International Conference on Intelligent Computing, Springer, 2012, pp. 505–512.

[165] Z. ZHANG, P. LUO, C. C. LOY, AND X. TANG, *Facial landmark detection by deep multi-task learning*, in European conference on computer vision, Springer, 2014, pp. 94–108.

[166] D. ZHOU, J. WANG, B. JIANG, H. GUO, AND Y. LI, *Multi-task multi-view learning based on cooperative multi-objective optimization*, IEEE Access, 6 (2017), pp. 19465–19477.

[167] C. ZIFTCI AND I. KRÜGER, *Getting more from requirements traceability: Requirements testing progress*, in 2013 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE), IEEE, 2013, pp. 12–18.