COMPLEMENTING DEFICIENT BUG REPORTS WITH MISSING INFORMATION LEVERAGING NEURAL TEXT GENERATION

by

Usmi Mukherjee

Submitted in partial fulfillment of the requirements for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
December 2023

To Baba and Maa for supporting me through all hardships and loving me unconditionally

Table of Contents

List of Tables

List of Figures

Abstract

Software bug reports often lack crucial information (e.g., steps to reproduce, expected behaviour), which makes bug resolution challenging. A recent study found that 78% of bug reports from open-source projects (e.g., Eclipse) contain less than 100 words each and thus require the developers to spend more time on bug resolution. According to an existing survey, 77% of 327 professional developers from major technology companies (e.g., Google, Meta) consider missing information a major problem and emphasize complementing them with useful information (e.g., environment configuration). In this thesis, we propose and evaluate two novel approaches that complement deficient bug reports with relevant information using Generative AI. In our first study, we propose — BugMentor — a novel approach that combines structured information retrieval and neural text generation (e.g., CodeT5) to generate appropriate answers to the follow-up questions from bug reports. Our approach identifies past, relevant bug reports to a given bug report, constructs the context and then leverages it to generate the answers. According to our evaluation, BugMentor generates good answers and outperforms three existing baselines significantly in terms of four appropriate metrics (e.g., BLEU, Semantic Similarity). We also conduct a developer study involving 10 participants where BugMentor's answers were found to be more accurate, precise, concise and useful. In our second study, we propose — BugEnricher — a novel approach that enriches bug reports with meaningful explanations using neural text generation. We fine-tuned the T5 model on software-specific vocabulary (e.g., Stack Overflow tags) to generate explanations against software-specific terms and jargon, which has the potential to enrich a bug report. Our evaluation using three performance metrics shows that BugEnricher generates understandable to good explanations according to Google's standards and outperforms two baselines from the literature. We also conduct a case study to demonstrate the benefit of our bug report enhancement and found that it was able to improve an existing technique in detecting textually dissimilar duplicate bug reports, which has been reported as a major challenge. Given the empirical evidence above, our approaches have strong potential to support bug resolution and bug report management.

List of Abbreviations Used

BLEU	Bi-Lingual Evaluation of Understudy.
IR	Information Retrieval.
METEOR	Metric for Evaluation of Translation with Explicit ORdering.
NLM	Neural Language Modelling.
NMT	Neural Machine Translation.
SEDE	Stack Exchange Data Explorer.
SS	Semantic Similarity.
TF-IDF	Term Frequency - Inverse Document Frequency.
VSM	Vector Space Model.
WMD	Word Mover Distance.

Acknowledgements

गुरु ब्रम्हा गुरु विष्णुः गुरु देवो महेश्वरः ।
गुरुः साक्षात् परंब्रह्म तस्मै श्री गुरवे नमः ॥

—by **Veda Vyāsa** in *Skanda Purana*

Meaning: My Guru (teacher) is the representative of *Brahma*, *Vishnu*, and *Shiva*. He creates, sustains knowledge and destroys the weeds of ignorance. I offer my obeisance to my Guru.

This *shlok* expresses the idea that *guru* is revered as a divine entity, embodying the creative, preserving, and enlightening aspects of the divine. I would like to acknowledge and offer my sincere gratitude to the people who have played a pivotal role in shaping my academic journey at Dalhousie University.

First, I thank the Almighty, who granted me good health, intellectual capacity, and the fortitude to carry out my thesis. Then I want to express immense gratitude to my supervisor, Dr. Masud Rahman, for allowing me to join his research group and supporting me throughout my Master's studies. I thank him for motivating me and believing in me constantly, especially when I could not. His unwavering support, insightful feedback, and patience have been invaluable to me throughout my thesis work.

I would like to express my sincere gratitude to Dr. Vlado Keselj and Dr. Tushar Sharma for their insightful feedback and thorough evaluation of my work. Their critical analysis and insightful comments have raised the standard and rigor of my thesis, and I sincerely appreciate their time and expertise.

To my beloved parents, Samar Kumar Mukherjee and Madhumita Mukherjee, for their support and love in every capacity throughout my life. Their guidance, encouragement, and sacrifices have been instrumental in shaping the person I am today and achieving this significant milestone in my life.

I would also like to thank my brother Udayan Mukherjee and my sister Shaniqua Mukherjee for guiding me all the way. Thank you for taking me to places that I never thought I could visit, for the amazing food and for also being my friends. I am also thankful to Flo for making me happy through the simplest of ways and being my unofficial mental health support dog.

I would like to acknowledge the irreplaceable support and affinity of my precious friends Nikita Rathi and Isaac Xu for their friendship, support, and shared experiences. In particular, I would like to thank Nikita for taking care of me when I was sick, teaching me how to cook and also making me food. I would like to thank Isaac, for being my support system through this journey, for cheering me up when I was sad, and for teaching me about transformers. I thank both of them for being my best friends, and I consider myself lucky to have them by my side.

To all of the members of the Intelligent Automation in Software EngineeRing (RAISE) Lab – Sigma Jahan, Ohiduzzaman Shuvo, Parvez Mahbub, Asif Samir, Riasat Mahbub, Shihui Gao, Lareina Yang, Callum MacNeil and Mehil B Shah with whom I have had the opportunity to grow as a researcher. I would also like to express my gratitude to all the members of the Software Maintenance and Analytics Research Team (SMART) Lab — Indranil Palit, Gautam Shetty, Mootez Saad, Saurabhsingh Rajput and Himesh Nandani. I appreciate their efforts in making our work in the labs enjoyable and for engaging in meaningful discussions. In particular, I thank Sigma, Parvez, Shuvo and Mehil for making me feel at home despite being away from home. Special thanks go to Indranil and Mehil for bringing laughter to the group, especially on gloomy days.

Furthermore, I would like to thank Dalhousie University and the Department of Computer Science for creating a stimulating learning environment and providing the necessary resources for my research. The outcome of this thesis has been greatly influenced by their drive for intellectual progress and their commitment to excellence. In particular, I would like to thank Dr. Michael McAllister and Megan Baker.

Lastly, I extend my heartfelt thanks to all those who have supported me in various ways throughout this research endeavour. Your encouragement, advice, and assistance have been invaluable, and I am deeply grateful for your presence in my life.

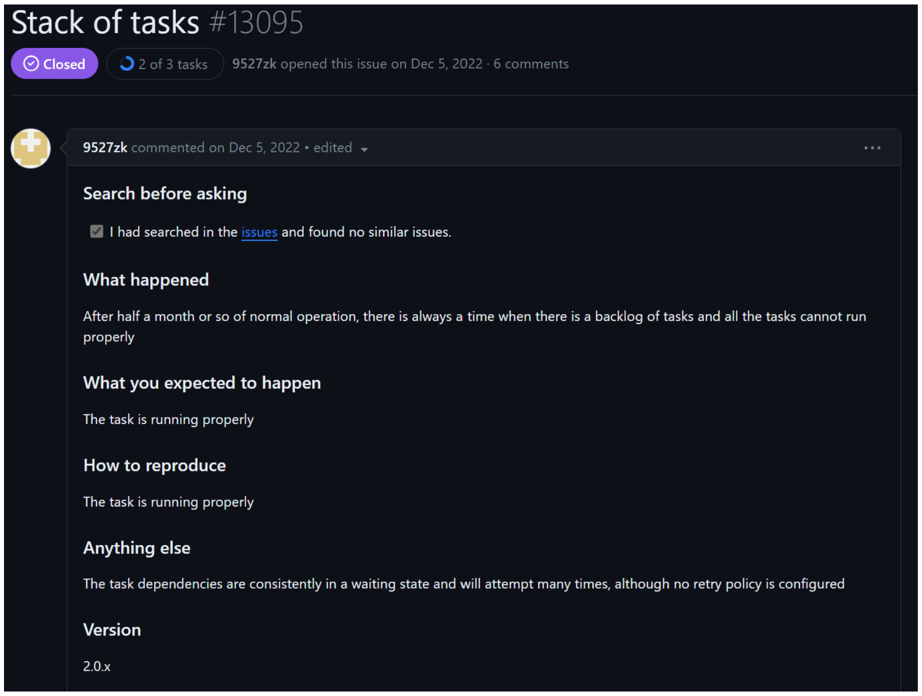
Chapter 1

Introduction

1.1 Motivation

Software bugs are human-made mistakes in a software system that prevent it from working as expected [1]. Existing studies have shown that software bugs cost the global economy billions of dollars every year [2], [3]. Hundreds of software bugs are submitted to bug-tracking systems like GitHub and JIRA as *bug reports* [4]. These bugs are then triaged, analyzed, and resolved by developers. Developers spend ∼50% of their programming time finding and fixing bugs [2]. Thus, bug resolution has been one of the major challenges in software maintenance [3]. A recent study suggests that up to 78% of 32,198 bug reports collected from four open-source projects (e.g., Eclipse, Mozilla, Firefox, GCC) have less than 100 words each, which might not be sufficient (a.k.a., short bug reports) [5]. These short bug reports required 121 days extra on average for their resolutions as opposed to the well-written bug reports [5]. That is, missing information in bug reports could lead to their delayed resolution [5]. According to a recent survey [3], 77% of 327 software practitioners (e.g., developers, testers, managers) from the major technology companies (e.g., Google, Meta, Amazon, Microsoft) consider missing information as a major problem and emphasize on complementing bug reports with useful information (e.g., steps to reproduce, environmental configuration) [3]. Missing information has also been found to be a key factor behind the non-reproducibility of software bugs [6]. Thus, missing information has been a major challenge and complementing the bug reports with relevant information would greatly benefit the developers in their bug resolution.

Ideally, bug reports should contain all the information, such as system configuration, expected behaviour, observed behaviour, and reproducing steps that help a developer resolve a bug [7]. However, in practice, they often do not contain all the required information for reproducing or resolving a bug [7]. Let us consider the example bug report in Fig. 1.1. It discusses a task backlog problem where the


Stack of tasks #13095

Closed · 2 of 3 tasks · 9527zk opened this issue on Dec 5, 2022 · 6 comments

9527zk commented on Dec 5, 2022 • edited

Search before asking

- [x] I had searched in the issues and found no similar issues.

What happened

After half a month or so of normal operation, there is always a time when there is a backlog of tasks and all the tasks cannot run properly

What you expected to happen

The task is running properly

How to reproduce

The task is running properly

Anything else

The task dependencies are consistently in a waiting state and will attempt many times, although no retry policy is configured

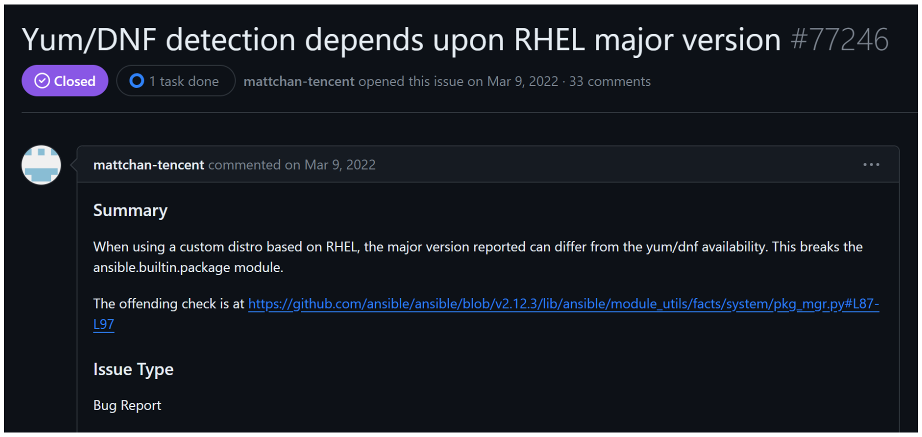
Version

2.0.x

Figure 1.1: An example bug report with missing information (ID #13095)

task dependencies are consistently in a waiting state. However the reporter does not provide any system configuration details, logs or steps to reproduce. As a result, the report was later marked as "needs more information" and then closed. According to existing literature [7], 64.8% of bug reports do not contain any expected behaviour of target software systems, and 48.6% of them do not explicitly describe the steps to reproduce a bug. Many software projects on GitHub now require the bug reports to adhere to specific templates or standard guidelines [8]. However, many bug reporters might fail to comply with them and might not be able to provide all the information during report submission [9]. Developers thus often pose *follow-up* questions to bug reporters soliciting the missing information. Unfortunately, the bug reporters often find it challenging to answer the follow-up questions in a timely fashion, according to a recent developer survey [6]. Such a lack of responses could lead to non-reproducible or unresolved bugs [10]. However, there has been only a little research investigating the follow-up questions from bug reports or their answers.

Answers to the follow-up questions provide more contextual information regarding a reported bug, which could help the developers resolve the bug. However, newcomers or novice developers to a project might need additional help to accurately understand


Yum/DNF detection depends upon RHEL major version #77246

Closed · 1 task done · mattchan-tencent opened this issue on Mar 9, 2022 · 33 comments

mattchan-tencent commented on Mar 9, 2022

Summary

When using a custom distro based on RHEL, the major version reported can differ from the yum/dnf availability. This breaks the ansible.builtin.package module.

The offending check is at https://github.com/ansible/ansible/blob/v2.12.3/lib/ansible/module_utils/facts/system/pkg_mgr.py#L87-L97

Issue Type

Bug Report

Figure 1.2: An example bug report (ID #77246)

or resolve a bug. In particular, complex contextual information and the incomplete or inaccurate content in bug reports could make bug understanding a challenging task [11]. Let us consider the example bug report in Fig. 1.2. The bug report uses several software-specific terms such as "custom distro", "RHEL" and "yum/dnf". It discusses the ansible version mismatch between the custom distribution of Red Hat Enterprise Linux (RHEL) and the Yellowdog Updater Modified (yum)/dandified YUM (dnf) package management tool. To a newcomer, all these terminologies could be daunting and discouraging. However, decoding them is essential to understand and diagnose the reported bug. According to an existing study [12], even with prior experience, developers often struggle to acquire a comprehensive understanding of any application domain and understand the discussions from a bug report. Thus, a lack of explanation for the domain-specific terms or jargon could be a major issue towards bug understandability.

1.2 Problem Statement

Bug reports are a valuable resource for software maintenance and continuous evolution. Over the last few decades, there has been extensive research to support various bug report management tasks, including bug triage [13], [14], issue report classification [15], [16], duplicate bug report detection [17], [18], and bug localization [19], [20]. However, the problem of missing information or domain-specific jargon in bug reports has not been comprehensively studied or addressed. Given the evidence above, adding complementary information to deficient bug reports could greatly benefit software practitioners in their work.

There have been existing studies that provide complementary information through Question Answering (QA) to support various software engineering tasks. Tian et al. [21] designed APIBot that can answer questions related to an API by analyzing relevant API documentation. Bansal et al. [22] designed a context-aware QA system to answer basic questions about subroutines. Lu et al. [23] proposed a QA approach that can provide answers by executing structured queries generated from bug templates. However, there has been only a little research investigating the follow-up questions from bug reports or their answers. Breu et al. [10] first conducted a mix of quantitative and qualitative analysis on follow-up questions and found that 32.34% of the questions were never responded to. They suggest that the questions in the bug reports were critical to the effective triaging, reproduction and resolution of a bug. Recently, Imran et al. [9] proposed a technique that recommends follow-up questions against a deficient bug report using structured information retrieval. Although both studies above deal with the follow-up questions from a bug report and are a source of inspiration, they do not answer the questions.

There have been existing studies to support newcomers or inexperienced developers who may struggle to comprehend software bug reports. An existing survey by Tan et al. [24] suggests that a clear description of a bug that does not rely on in-depth domain knowledge is necessary to help newcomers understand and resolve the bug. Recently, Correa et al. [25] suggest that the inclusion of web links (to external knowledge sources or artifacts) in the issue tracker discussion can benefit the developers. Zhang et al. [5] propose to supplement a bug report with a list of sorted sentences that are extracted from past, relevant bug reports. Dit et al. [26] proposed a technique that recommends relevant comments so that the developers can make explicit connections between the recommended comments and existing ones. Such connections could help the developers better understand a bug report. While the above approaches offer complementary information to support bug understanding, they do not focus on the domain-specific terms or jargon, which warrants for further investigation.

Given the above discussions, missing information is one of the key factors that affect developers when comprehending software bug reports and could lead to delayed bug delayed reproduction and resolution. It affects bug reports in two different ways. First, bug reports often do not contain sufficient information for timely resolution. Developers thus pose follow-up questions asking the bug reporter for missing information. However, bug reporters or any user facing a similar bug may find it challenging to answer them due to a lack of domain-knowledge. Second, bug reports may contain domain-specific terms or jargon that may not be well understood by novice or newcomer developers. Traditional bug tracking systems do not provide any support to comprehend such domain-specific terms or jargon. To the best of our knowledge, existing literature might also not be sufficient to enhance the bug reports plagued by missing information. We thus perform two studies to complement such deficient bug reports with missing information using automated tools and technologies.

1.3 Our Contribution

In this thesis, we propose and evaluate two novel techniques that support developers in bug resolution by complementing a deficient bug report in two different ways.

In our first study, we propose a novel technique — *BugMentor* — that can offer relevant answers to follow-up questions from bug reports by combining structured information retrieval and neural text generation. First, we capture textually relevant questions, answers, and bug reports against a follow-up question using structured information retrieval [27]. Then we capture each item's embeddings using Word2Vec [28] and re-rank them based on their semantic relevance to the question. Second, we generate meaningful answers to the follow-up question by leveraging the ranked items above as *context* with a neural text-generation technique (e.g., CodeT5).

We evaluate answers from *BugMentor* using four performance metrics — BLEU score [29], METEOR [30], Semantic Similarity [31], and WMD [32]. We achieve a BLEU score of 34.12 which indicates that our generated answers are *understandable* to *good* according to Google AutoML documentation [33]. We also conduct an ablation study to justify our combination of structured information retrieval and neural text generation in BugMentor. We find that BugMentor can capture a rich context leveraging structured information retrieval and thus can generate meaningful answers. BugMentor also outperforms all three baselines — Lucene [34], CodeT5 [35], AnswerBot [36] — in all four metrics. To further demonstrate its benefit, we conduct a developer study involving 10 participants. According to the participants, the answers from BugMentor were more accurate, more precise, more concise and more useful compared to the baseline answers.

In our second study, we propose – *BugEnricher* – a novel technique that can enhance bug reports with meaningful explanations to their domain-specific terms or jargon using neural text generation. First, we collect thousands of domain-specific vocabulary and their explanations from three different sources – StackOverflow, API documentation, and glossary. Second, we fine tune the T5 model [37] on our collected vocabulary and explanations. Third, we use TF-IDF to extract the infrequent domain-specific terms from each bug report. Finally, we generate natural language explanations to the domain-specific terms or jargon using our fine-tuned T5 model.

Our evaluation using three performance metrics shows that *BugEnricher* can generate *understandable* and *good* explanations according to Google's standard, and can outperform two existing baselines — T5 [37] and AnswerBot [36] — from the literature. To further demonstrate the benefit of our explanations, we conduct a case study using the bug reports enriched with explanations. We evaluate the performances of an existing technique [38] for duplicate bug report detection that is impacted by the problem of textually dissimilarity [39]. We find that the enriched bug reports were able to improve the performances of the existing technique in detecting textually dissimilar duplicate bug reports.

Given the empirical evidence, our proposed techniques have the potential to significantly improve the bug report management and their resolution.

1.4 Related Publications

Several parts of this thesis are either submitted or ready to be submitted to different conferences. We provide a list of papers here. In each of these papers, I am the primary author, and all the studies were conducted by me under the supervision of Dr. Masud Rahman. While I wrote these papers, the co-author took part in advising, editing, and reviewing the papers.

- *Usmi Mukherjee* and M. Masudur Rahman. *Answering Follow-up Questions from Bug Reports Leveraging Structured Information Retrieval with Neural*

Text Generation. In Proceedings of the 47th International Conference on Software Engineering (ICSE 2025), pp.13, Ottawa, Canada, April-May 2025.(Pre-submission)

- *Usmi Mukherjee* and M. Masudur Rahman. *BugEnricher: Explaining Domain-specific Terms and Jargon from Bug Reports with Neural Machine Translation.* In Proceeding of the 40th IEEE International Conference on Software Maintenance and Evolution (ICSME 2024), pp. 12, Flagstaff, Arizona, October 2024 (Pre-submission).

1.5 Outline of the Thesis

The thesis contains five chapters in total. To complement missing information in bug report effectively, we conduct two independent but interrelated studies, and this section outlines different chapters of the thesis.

- Chapter 2 discusses several background concepts (e.g., embedding, transformers, neural language modeling) that are required to follow the rest of the thesis.
- Chapter 3 discusses our first study that proposes *BugMentor*, a novel approach that combines neural text generation (e.g., CodeT5) and structured information retrieval to generate appropriate answers to the follow-up questions.
- Chapter 4 discusses our second study that proposes *BugEnricher*, a novel transformer-based generative model that generates natural language explanations for software-specific terms in Bug Reports.
- Chapter 5 concludes the thesis with a list of directions for future works.

Chapter 2

Background

In this chapter, we introduce the required terminologies and concepts to follow the remaining of the thesis. Section 2.1 discusses Neural Language Modelling (NLM), a deep learning based approach to learn the probability distribution of a textual corpus. Section 2.2 discusses Neural Machine Translation (NMT), a deep neural network based approach for automated translation. Section 2.3 discusses structured information retrieval for question answering. Section 2.4 describes embedding, the process of converting high dimensional vector data into low dimensional semantic representation. Section 2.5 discusses the definition of domain-specific terms or jargon that are present in software bug reports. Section 2.6 discusses TF-IDF, a statistical measure for determining the importance of a term in a document.

2.1 Neural Language Modelling

A language model is a probabilistic model for natural language texts. It generates the probability of occurrence of a word or a phrase within a given text corpus. Such probability distributions could be useful in various tasks. For example, in a text generation task, a language model predicts the next word w_L on the basis of all preceding words and their probability of co-occurrence [40], [41].

$$w_L = \arg\max_{w_v \in V} P(w_v | w_{L-1} w_{L-2} ... w_1) \tag{2.1}$$

where w_L is the next predicted word, V is the vocabulary, and $w_{L-1}, w_{L-2}, ..., w_1$ are the previously predicted words.

Neural language models use neural networks to capture the complex patterns and dependencies of natural language and leverage them to compute the probabilities of the next word. In particular, they attempt to predict the next word while estimating the numerical representation of the words and texts (a.k.a., embeddings) [42]. Our first study – BugMentor – uses neural language modelling to generate answers to follow-up questions. Our second study – BugEnricher – uses neural language modelling to generate explanations for domain-specific jargon.

2.2 Neural Machine Translation

Neural Machine Translation (NMT) is an automated translation technique based on deep neural networks [43]. NMT has made significant progress, capturing the attention of both researchers and practitioners. It supports various software engineering tasks such as automatic program repair [44], [45], commit message generation [46], and code summarization [47]. An NMT model consists of two main components: an encoder and a decoder. The encoder accepts a sequence as input and uses Neural Language Modeling to construct a numerical representation of the input called the context vector. This context vector is then fed into the decoder, which, based on this vector, sequentially generates the target sequence, one token at a time. We use Transformer [37], [48] based state of the art NMT models – CodeT5 [35] and T5 [37] as a part of our studies. In our first study – BugMentor, we use NMT for generating answers to the follow-up questions from bug reports. In our second study – BugEnricher, we use NMT for generating explanations for domain-specific terms or jargon from bug reports.

2.3 Structured Information Retrieval

Information Retrieval (IR) is a popular method that facilitates access to vast repositories of information (e.g., World Wide Web). In traditional IR, there are two major components: query and corpus. The query consists of a few keywords, whereas the corpus represents a collection of searchable documents. These queries and corpus are preprocessed using standard natural language preprocessing techniques such as text normalization, stopword removal and lemmatization. The query corpus is also indexed by collecting various statistics such as term frequency (TF, the number of times a term occurs in a given document) and document frequency (DF, the number of documents in which the term appears) [27]. These terms and document statistics are then used in algorithms such as TF-IDF and BM25. Traditional IR, however, does not take into account the underlying structure of the query and the corpus.

Structured IR is found to be more effective in retrieving. It has the potential to reduce noise and spurious matching due to the structured nature of the search. For example, in our first study, BugMentor uses three main components of the bug reports – title, description, and follow-up questions – that are used to match with the corpus bug reports. Structured IR divides both query and document into granular abstractions and calculates the lexical similarity between them [27].

2.4 Word Embedding

Word embedding is a vector representation of words in a Vector Space Model (VSM). It encodes the meaning of the word in such a way that semantically similar words are closer to each other within the vector space [49]. An embedding function $E : \mathcal{X} \rightarrow \mathbb{R}^d$ takes an input X in the domain $\mathcal{X}$ and generates its vector representation in a d-dimensional vector space [50]. Each word or phrase is mapped to a vector of real numbers that represent the meaning of the input [51]. Word embedding can overcome issues faced by traditional VSM, such as the sparse representation problem of one-hot encoding or the vocabulary mismatch issue of TF-IDF and BM25. Several techniques employ neural networks to learn richer word representations, such as Word2Vec [52] and GloVe [53]. We use Word2Vec embeddings to determine semantic relevance between bug reports in our first study, BugMentor.

2.5 Domain-specific Terms or Jargon

Domain-specific terms or jargon are specialized vocabularies for a particular application domain such as programming language, library or software. They may have specific meanings or interpretations unique to that domain. They can be categorized based on their type, application domain (e.g., Android, Firefox), domain concepts (e.g.,semantic labels/classes) [54], [55]. The explanations or meanings for domain-specific terms can be found in their corresponding documentation and glossary. In our second study, BugEnricher, we explain the domain-specific terms or jargon from several domains including two programming languages - Java and Python.

2.6 TF-IDF

Term Frequency - Inverse Document Frequency (TF-IDF) is a statistical measure of the importance of a term within a document. Each word in the document can have a TF-IDF score. First, the Term Frequency (TF) is computed using the occurrence of each term within a document and can be normalized using the total number of terms in the document as follows.

$$\mathrm{tf}(t,d) = \frac{\text{number of times term } t \text{ appears in document } d}{\text{total number of terms in document } d} \tag{2.2}$$

Second, the Document Frequency (DF) refers to the number of documents containing the term. Words unique to a small percentage of documents (e.g., technical jargon terms) receive higher importance than the common words across all documents (e.g., a, the, and). Inverse Document Frequency (IDF) measures the rarity or uniqueness of a term across the entire collection of documents by inverting the DF as follows.

$$DF(t) = occurrence\ of\ term\ t\ in\ N\ documents \tag{2.3}$$

$$IDF(t) = log(N/(DF(t)+1)) \tag{2.4}$$

Finally, the TF-IDF score is the product of the TF and IDF scores. The greater the TF-IDF score, the more significant the term in that document [56].

$$\text{tf-idf}(t,d) = \mathrm{tf}(t,d) \times \mathrm{idf}(t) \tag{2.5}$$

We compute the TF-IDF score to determine the less frequent but important domain-specific terms or jargon from a bug report in our second study, BugEnricher.

2.7 Summary

In this chapter, we introduce different terminologies and background concepts to help one follow the remainder of the thesis. We discuss Neural Language Modelling in Section 2.1, deep learning based approach to learn the probability distribution of a textual corpus; Neural Machine Translation in Section 2.2, a deep learning approach for automated translation; Structured Information Retrieval in Section 2.3 for effective retrieval of documents for a given query; Word Embeddings in Section 2.4 for vector representation of text; Section 2.5 defines domain-specific terms or jargon and TF-IDF in Section 2.6 for determining important keywords in a document with respect to a collection of documents.

Chapter 3

BugMentor: Answering Follow-up Questions from Bug Reports Leveraging Structured Information Retrieval with Neural Text Generation

Software bug reports often lack crucial information (e.g., steps to reproduce), which makes bug resolution challenging. Developers thus ask follow-up questions to capture additional information. However, according to existing evidence, bug reporters often face difficulties answering them, which leads to the premature closing of bug reports without any resolution. Recent studies suggest follow-up questions to support the developers, but answering the follow-up questions still remains a major challenge. In this chapter, we propose BugMentor, a novel approach that combines neural text generation (e.g., CodeT5) and structured information retrieval to generate appropriate answers to the follow-up questions.

The rest of this chapter is organized as follows. Section 3.1 introduces our study and discusses the novelty of our work. Section 3.2 discusses a motivating example to demonstrate the effectiveness of BugMentor. Section 3.3 discusses our proposed technique. Section 3.4 presents our experimental design, datasets, and evaluation metrics. Section 3.5 discusses the evaluation results of BugMentor. Section 3.6 discusses relevant works to our study. Section 3.7 discusses the threats to the validity of our study. Finally, Section 3.8 summarizes this study.

3.1 Introduction

Software bugs are human-made errors in a software system that prevent the software from working as expected [1]. Studies have shown that software bugs cost the global economy billions of dollars every year [2], [3]. Developers also spend ∼50% of their programming time finding and fixing bugs [2]. Thus, bug resolution has been one of the major challenges in software maintenance [3]. Hundreds of software bugs are submitted to bug-tracking systems like GitHub and JIRA as *bug reports* [4]. These bugs are then triaged, analyzed, and resolved by developers.

Ideally, bug reports should contain all the information, such as system configuration, expected behaviour, observed behaviour, and reproducing steps that help a developer resolve a bug [7]. However, in practice, they often do not contain all the required information for reproducing or resolving the bug [7]. According to existing literature [7], 64.8% of bug reports do not contain any expected behaviour of target software systems, and 48.6% of them do not explicitly describe the steps to reproduce a bug. Missing information like this has been found to be a key factor behind the non-reproducibility of software bugs [6]. In a survey conducted by Zou et al. [3], 77% of 327 professional developers from the major technology companies (e.g., Google, Meta, Amazon, Microsoft) consider missing information as a major problem and emphasize on complementing bug reports with useful information (e.g., steps to reproduce, environmental configuration) [3]. In short, missing information has been a key challenge toward cost-effective bug resolution.

Many software projects on GitHub now require bug reports to adhere to specific templates or standard guidelines [8]. Unfortunately, many bug reporters might fail to comply with them and might not be able to provide all the information during report submission [9]. Developers thus often pose *follow-up* questions to bug reporters soliciting the missing information. Unfortunately, the bug reporters often find it challenging to answer the follow-up questions in a timely fashion, according to a recent developer survey [6]. For instance, Fig. 3.1 shows how a bug report was closed prematurely without any resolution due to a lack of responses to the follow-up question posed by the developer.

Over the last few decades, there has been extensive research to support various bug report management tasks, including bug triage [13], [14], issue report classification [15], [16], duplicate bug report detection [17], [18], and bug localization [19], [20]. However, there has been only a little research investigating the follow-up questions from bug reports or their answers. Breu et al. [10] first performed both quantitative and qualitative analyses on follow-up questions and found that 32.34% of the questions were never responded to. According to them, these unanswered questions in the bug reports were critical to the effective triaging, reproduction and debugging of a bug.

Recently, Imran et al. [9] proposed a technique that recommends follow-up questions against a deficient bug report using information retrieval. While both studies above deal with the follow-up questions of a bug report, they do not answer them.

Automated Question Answering (QA) has been an active research topic for decades in Information Retrieval (IR) and Natural Language Processing (NLP) communities [21]–[23], [36], [57]–[62]. There also have been several works in the context of software engineering. Tian et al. [21] designed APIBot that can answer questions related to an API by analyzing relevant API documentation. Bansal et al. [22] designed a context-aware QA system to answer basic questions about subroutines. Lu et al. [23] proposed a QA approach that can provide answers by executing structured queries generated from bug templates. However, their approach might fail when a bug report does not contain the requested information. Xu et al. [36] designed AnswerBot that can synthesize answers for a non-factoid technical question from StackOverflow Q&A website. While the above approaches are a source of inspiration, they do not answer the follow-up questions posed by developers against the bug reports.

In this chapter, we propose a novel technique — *BugMentor* — that can offer relevant answers to follow-up questions from bug reports by combining structured information retrieval and neural text generation. First, we capture textually relevant questions, answers, and bug reports against a follow-up question using structured information retrieval [27]. Then we capture each item's embeddings using Word2Vec [28] and re-rank them based on their semantic relevance to the question. Second, we generate meaningful answers to the follow-up question by leveraging the ranked items above as *context* with a neural text-generation technique (e.g., CodeT5) [63].

We select the top 20 popular projects from GitHub that use four programming languages and collect 30,869 bug reports from them for our experiments. We evaluate our technique using four popular metrics for text generation, namely BLEU score [29], METEOR [30], Semantic Similarity [31], and WMD [32]. We achieve a BLEU score of 34.12 which indicates that our generated answers are *understandable to good* according to Google AutoML documentation [33]. We also conduct an ablation study to justify our combination of structured information retrieval and neural text generation in BugMentor. We find that BugMentor can leverage the rich context captured through structured information retrieval and thus can generate meaningful answers. BugMentor also outperforms all three baselines — Lucene [34], CodeT5 [35], AnswerBot [36] in all four metrics. To further demonstrate its benefit, we conduct a developer study involving 10 participants. According to the participants, the answers from BugMentor were more accurate, precise, concise and useful compared to the baseline answers.

To summarize, we make three contributions in this work:

(a) A novel technique — BugMentor — that can generate relevant answers to follow-up questions from bug reports by combining structured information retrieval and neural text generation (e.g., CodeT5).

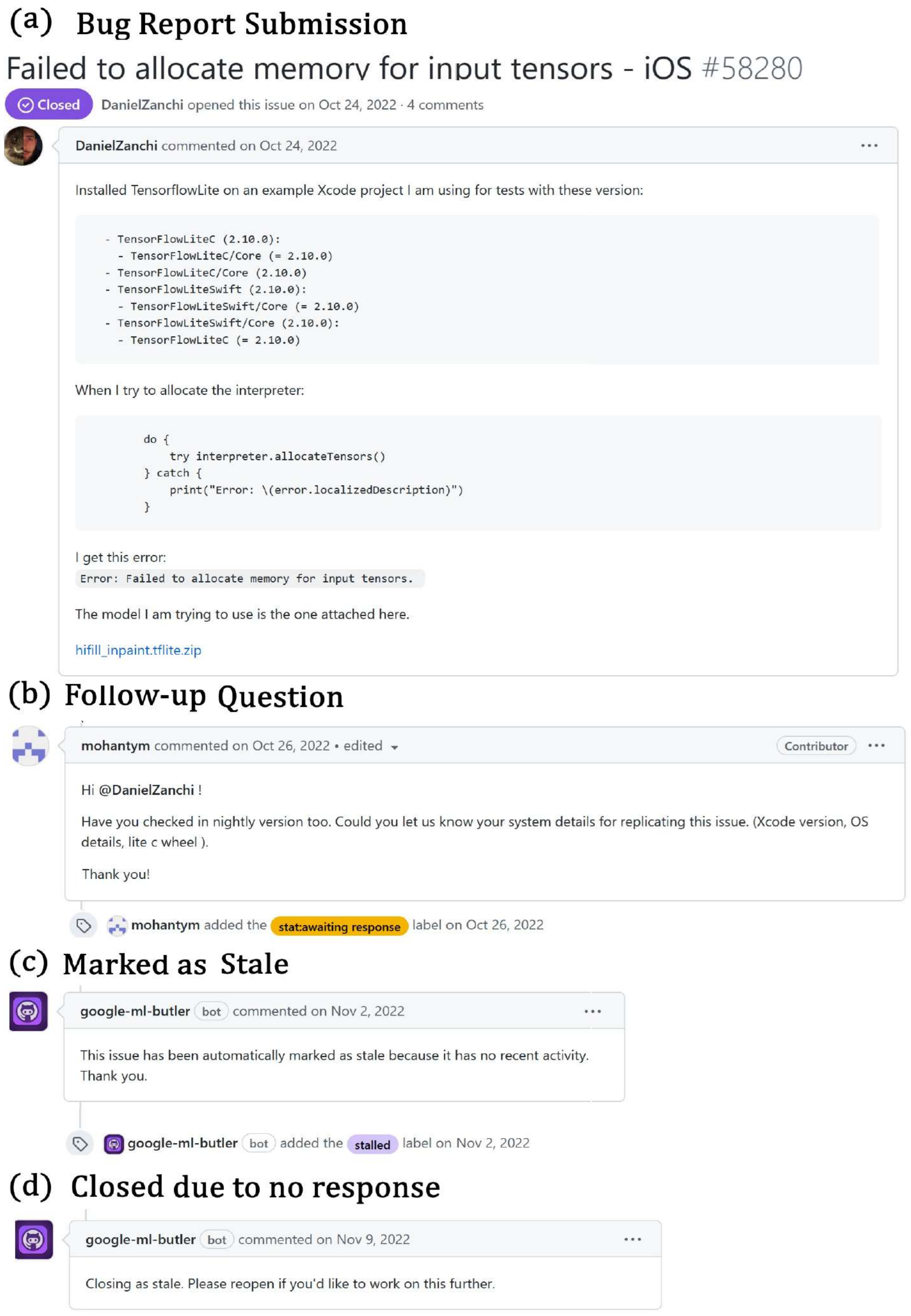
(b) A comprehensive evaluation and validation of BugMentor using both popular performance metrics (e.g., BLEU score, METEOR score, WMD, Semantic Similarity) and a developer study involving 10 participants.

(c) A replication package [64] that includes our working prototype, experimental dataset, and other configuration details for the replication or third-party reuse.

3.2 Motivating Example

To demonstrate the potential benefit of our work, let us consider the example bug report in Fig. 3.1. It has been taken from the *tensorflow* repository which is maintained by the organization *tensorflow* on GitHub [65]. The bug report (Step (a), Fig 3.1) discusses a memory allocation issue and provides minimal steps to reproduce the issue. However, it does not follow the standard issue template provided by GitHub [66] and lacks sufficient information about the system configuration. Subsequently, the developer (*@mohantym*) requests the missing information as a follow-up question inquiring whether they experienced the same issue on *nightly* version of the software (Step (b), Fig. 3.1). However, neither the bug reporter (*@DanielZanchi*) nor anyone else provided an answer to the follow-up question. As a result, the bug report was initially marked as stale (Step (c), Fig. 3.1) and later closed due to a lack of response (Step (d), Fig. 3.1).

As shown in Fig 3.1, without proper support in gathering missing information, software bugs either take longer to be resolved or remain unresolved and are ultimately closed. Several bugs, such as memory allocation bugs, are considered to be severe [67]


(a) Bug Report Submission

Failed to allocate memory for input tensors - iOS #58280

Closed DanielZanchi opened this issue on Oct 24, 2022 · 4 comments

DanielZanchi commented on Oct 24, 2022

Installed TensorflowLite on an example Xcode project I am using for tests with these version:

```
  - TensorFlowLiteC (2.10.0):
    - TensorFlowLiteC/Core (= 2.10.0)
  - TensorFlowLiteC/Core (2.10.0)
  - TensorFlowLiteSwift (2.10.0):
    - TensorFlowLiteSwift/Core (= 2.10.0)
  - TensorFlowLiteSwift/Core (2.10.0):
    - TensorFlowLiteC (= 2.10.0)
```

When I try to allocate the interpreter:

```
        do {
            try interpreter.allocateTensors()
        } catch {
            print("Error: \(error.localizedDescription)")
        }
```

I get this error:
`Error: Failed to allocate memory for input tensors.`

The model I am trying to use is the one attached here.

hifill_inpaint.tflite.zip

(b) Follow-up Question

mohantym commented on Oct 26, 2022 • edited Contributor

Hi @DanielZanchi !

Have you checked in nightly version too. Could you let us know your system details for replicating this issue. (Xcode version, OS details, lite c wheel).

Thank you!

mohantym added the stat:awaiting response label on Oct 26, 2022

(c) Marked as Stale

google-ml-butler bot commented on Nov 2, 2022

This issue has been automatically marked as stale because it has no recent activity. Thank you.

google-ml-butler bot added the stalled label on Nov 2, 2022

(d) Closed due to no response

google-ml-butler bot commented on Nov 9, 2022

Closing as stale. Please reopen if you'd like to work on this further.

Figure 3.1: An example of a bug report (ID #58280) being closed due to a lack of response to the follow-up question

that significantly impact software quality. Our work — BugMentor — delivers meaningful answers to the follow-up questions, as shown in the answer for issue #58280.

Answer for issue #58280

Question: Have you checked in nightly version too? Could you let us know your system details for replicating this issue (Xcode Version, OS details, lite c wheel)?
Generated Answer: Adding two flags to Xcode's 'Other Linker Flags' settings and modify the Podfile to use the nightly TensorFlow build, specifically 'TensorFlowLiteSwift' and 'TensorFlowLiteSelectTfOps'.

We see that BugMentor was able to capture the context of the discussed problem above and provide useful suggestions. According to the bug report (Fig. 3.1), integrating the TensorFlow library into an Xcode project resulted in a memory allocation bug. BugMentor suggests modifying the *"Other Linker Flags"* from Xcode to link the IDE to various versions of the TensorFlow library, such as *TensorFlowLiteSwift* or *TensorFlowLiteSelectTfOps*. It should be noted that these library versions were carefully extracted by our technique from the problem context in the bug report. Furthermore, BugMentor was able to deliver complementary information (e.g., *Other Linker Flags*) that could be useful in resolving the library integration problem. The effectiveness of such an idea was further confirmed by a discussion on StackOverflow Q&A site [68].

3.3 BugMentor: Proposed Technique

Fig. 3.2 shows the schematic diagram of our proposed technique — BugMentor. As the input, it accepts a bug report of interest, its follow-up question, and a corpus of past bug reports with their follow-up questions and corresponding answers. As the output, our technique generates a relevant answer to the follow-up question. We discuss different steps of our technique in detail in the following sections.

3.3.1 Constructing the Corpus

We construct our corpus using past bug reports and their discussion history from 20 real-world open-source software systems on GitHub (Step 1, Fig 3.2).

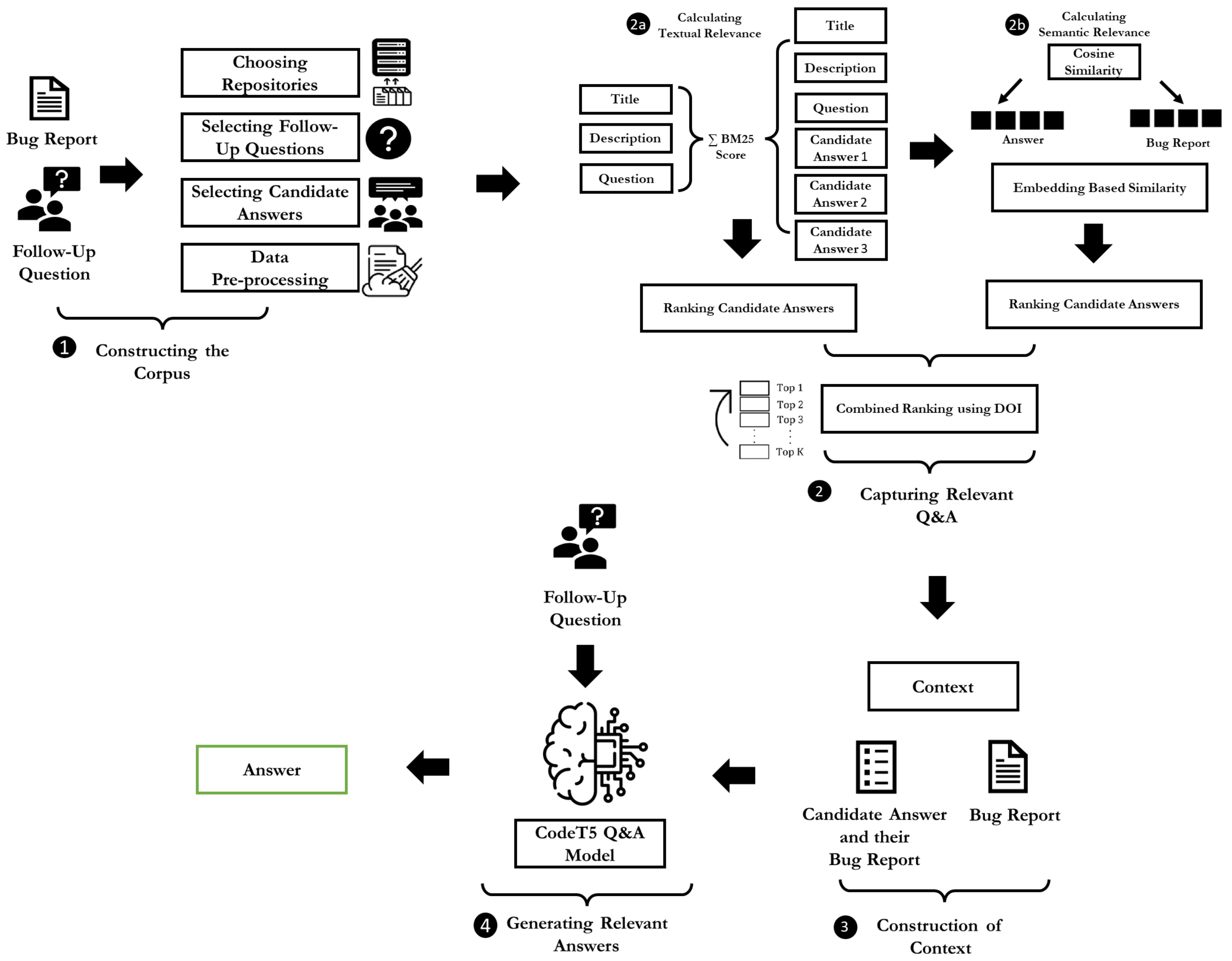

Bug Report
Follow-Up Question
Choosing Repositories
Selecting Follow-Up Questions
Selecting Candidate Answers
Data Pre-processing
1 Constructing the Corpus
2a Calculating Textual Relevance
Title
Description
Question
∑ BM25 Score
Title
Description
Question
Candidate Answer 1
Candidate Answer 2
Candidate Answer 3
Ranking Candidate Answers
2b Calculating Semantic Relevance
Cosine Similarity
Answer
Bug Report
Embedding Based Similarity
Ranking Candidate Answers
Top 1
Top 2
Top 3
Top K
Combined Ranking using DOI
2 Capturing Relevant Q&A
Context
Candidate Answer and their Bug Report
Bug Report
3 Construction of Context
Follow-Up Question
CodeT5 Q&A Model
4 Generating Relevant Answers
Answer

Figure 3.2: Schematic diagram of BugMentor

Choosing the Repositories for the Corpus

To construct our corpus, we collect high-quality repositories using a semi-automated approach. We follow the approach of Imran et al. [9] and choose GitHub [69] as a data source. GitHub [69] is a popular open-source platform that supports various software maintenance practices, including bug report management. We select the repositories and collect the bug reports as follows.

First, we select the most starred active repositories containing a minimum of 500 issues (reported as of May 2023) using GitHub's advanced search [70]. We then categorize the repositories into four subsets based on their programming languages — Python, Java, Javascript and C++ — where each programming language had five repositories.

Choosing the Bug Reports for the Corpus

From each repository, we then select the issues that were closed within the last five years. We select the issues that are labelled as "bug", "crash", or "defect" to ensure that they are discussing software bugs or defects. We also select the bug reports labelled as "needs more info" and "stale" that were closed due to a lack of activity. We use GitHub's REST API [71] to collect the bug reports and their discussion history. Each of our collected bug reports consists of several fields, namely issue ID, title, bug description, bug reporter, label, creation time, and resolution time.

Selecting Follow-up Questions

To select follow-up questions from each bug report, we first collect their issue comments using a GitHub API client [72]. From each comment, we capture four different fields, namely — comment ID, author of the comment, comment, and comment time. Following the strategy of Imran et al. [9], we collect the comments that begin with an interrogative word and end with a question mark. We use NLTK's Classifier [73] to identify these comments, as was applied previously [74]. We also consider comments that requested additional information using words such as 'please' or 'can you' as valid comments. Then we select the first valid, interrogative comment that is not written by the author as our follow-up question from each bug report. We manually check up to 30 comments from each bug report to identify our follow-up questions.

Selecting Candidate Answers

The next step in our corpus construction is to select candidate answers against the follow-up questions above. We apply three criteria to the selection of our candidate answers: (a) Candidate Answer 1 — the first comment after the follow-up question that was not authored by the question submitter [9], (b) Candidate Answer 2 — the first comment after the follow-up question that was authored by the bug reporter, and (c) Candidate Answer 3 — the most similar comment to the follow-up question based on BM25 algorithm [75]. Finally, our corpus consisted of hundreds of bug reports where we capture the Bug ID, title, description, follow-up question and three candidate answers from each bug report.

Data Pre-processing

We apply standard natural language pre-processing to each bug report, follow-up question and candidate answer from our corpus. First, we remove redundant or noisy elements such as escape sequences, special characters, URLs, stack traces or images from each item [9]. We use appropriate regular expressions from NLTK [76] to retain the natural language text and code elements while discarding the rest. Second, we perform lemmatization on all items in our corpus. This step ensures that words are transformed to their root forms, facilitating better analysis [77].

3.3.2 Capturing Relevant Candidate Answers

We then capture relevant candidate answers against each follow-up question (Step 2, Fig 3.2). We use the ElasticSearch implementation [78] of Lucene [34], [79], [80], a widely adopted search engine combining Boolean search and Vector Space Model (VSM), for our task. We employ the Okapi BM25 algorithm [81] from the engine for similarity calculation. In particular, we calculate two BM25-based relevance scores where we adapt an existing work of Saha et al. [27]:

$$s'\left(\vec{d}, \vec{q}\right) = \sum_{r \in Q} \sum_{f \in D} s(d_f, q_r) \tag{3.1}$$

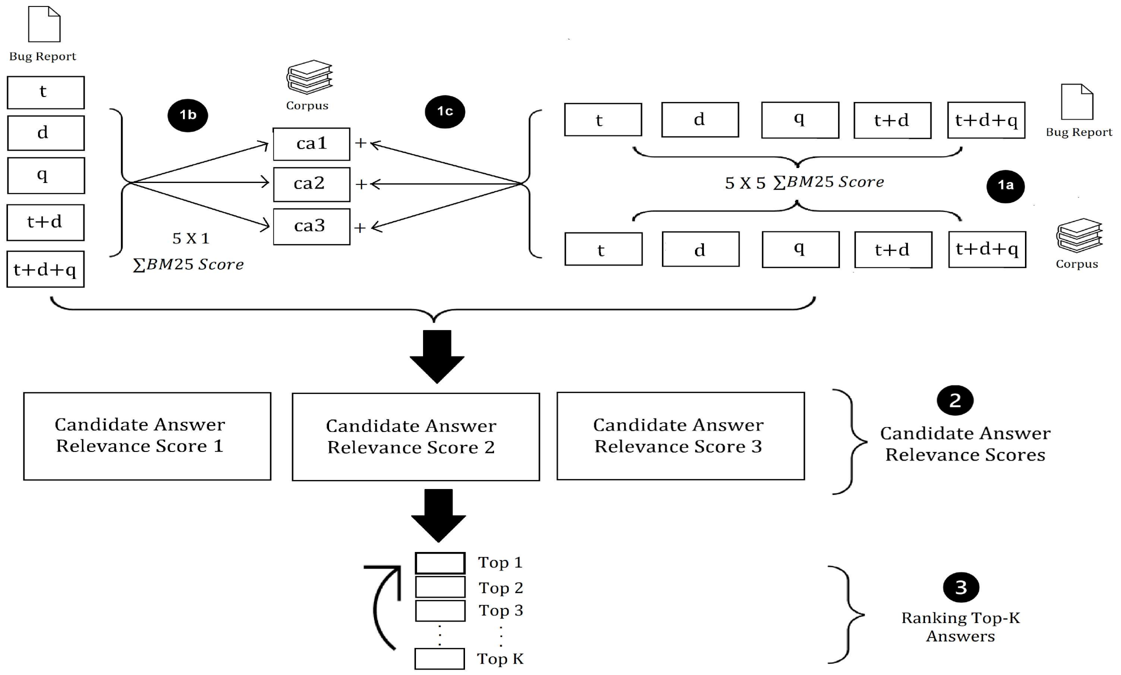


Figure 3.3: Capturing relevant answers with structured Information Retrieval

Here q_r is a query representation, and d_f is a field from the past bug report (e.g., title, description).

Detecting Relevant Candidate Answers

First, we capture five items from each of the given and past bug reports — title (t), description (d), follow-up question (q), title + description (t+d), and title + description + question (t+d+q). Then we conduct 25 (5x5) similarity calculations between these bug reports using Eq 3.1, and add all 25 similarity scores as shown in Step 1a in Fig. 3.3. According to existing literature [27], such an element-based similarity calculation can help prevent spurious matching. This score indicates the general relevance between a given and past bug reports.

Second, we capture five items from the given bug report — title (t), description (d), and follow-up question (q), title + description (t+d), and title + description + question (t+d+q) and three candidate answers (ca1, ca2, ca3) from each past bug report. Then, we conduct 5 (5x1) similarity calculations using Eq. 3.1, for each of the three candidate answers, as shown in Step 1b, in Fig. 3.3. This score indicates the relevance between the follow-up question (and its bug report) and each candidate answer.

While the score from Step 1a indicates a general relevance between bug reports, the score from Step 1b measures more granular relevance between the question and answer. Thus, the above two scores capture relevance from two different aspects that could be complementary to each other. We thus combine the scores from Step 1a and Step 1b to determine the relevance of each answer against the given follow-up question (and corresponding bug report) in Step 1c.

Ranking Based on Textual Relevance

We rank the candidate answers based on their BM25-based relevance scores calculated above (Steps 1-2, Fig. 3.3). In particular, we capture the top K (e.g., 5) relevant candidate answers from the corpus against a follow-up question (Steps 3, Fig. 3.3). It should be noted that these answers can come from various bug reports.

Ranking Based on Semantic Relevance

BM25 algorithm relies on keyword matching for relevance estimation, which could suffer from the vocabulary mismatch problem [82]. We thus incorporate embedding-based similarity into our approach and detect the semantically relevant candidate answers. We capture word embeddings, trained by Word2Vec [83] on Stack Overflow [28], and calculate the cosine similarity between the embeddings of the follow-up question and that of each candidate answer. We then re-rank the answers based on their semantic relevance to the question and return the top K answers.

Capturing the Top Relevant Answers

We combine both BM25-based ranking and semantic relevance-based ranking using the Degree of Interest (DOI) method. Rahman et al. [84] use the following formulae to combine two orthogonal rankings:

$$DOI = \frac{I}{N} \tag{3.2}$$

Here, I is the position of an answer in the ranked list and N is the total number of answers. First, we calculate the DOI score of each answer within the BM25-based list and then, we calculate the DOI score within the semantic relevance-based ranked list. Then we combined the DOI scores for each answer and rank the answers based on their combined DOI score.

3.3.3 Constructing the Context

Generative question-answering models such as CodeT5 rely on *context* to comprehensively understand the semantics or intent of a question and to generate an answer to the question [85], [86]. We thus construct the context to enrich our follow-up question (Step 3, Fig 3.2). To construct the context, we use three items from the previous step — one answer from the ranked list, its bug report, and the given bug report. The answer and its bug report are likely to contain additional information to compensate for the missing information in the given bug report that triggers a follow-up question. We repeat the context construction for each of the K candidate answers and send them to our CodeT5 model for final answers.

3.3.4 Generating Relevant Answers

We then generate relevant answers to the follow-up question by leveraging our context above with CodeT5, a pre-trained encoder-decoder Transformer model for neural text generation. CodeT5 has been pre-trained on 8.35M methods from open-source code accompanied by documentation and adopts an encoder-decoder network to generate texts [35] (Step 3, Fig 3.2). The model requires two components to operate, a question and its context. We provide the model with a follow-up question and its context from the previous step and capture an answer to the follow-up question from the model. For example, our technique — BugMentor — offers the following answer against the example question in Fig. 3.1.

Generated Answer: Adding two flags to Xcode's 'Other Linker Flags' settings and modify the Podfile to use the nightly TensorFlow build, specifically 'TensorFlowLiteSwift' and 'TensorFlowLiteSelectTfOps'

3.4 Experiment

We curated a large dataset containing 30,869 bug reports and their follow-up questions from 20 subject systems and evaluate BugMentor using four appropriate metrics from relevant literature — BLEU [29], METEOR [30], Semantic Similarity [31] and WMD [32]. To place our work in the literature, we also conduct an ablation study [34], [35] and compare our technique with three baseline techniques. Through our experiments, we answer four research questions as follows:

(a) **$\mathbf{RQ_1}$**: How does our technique perform in answering follow-up questions in terms of different automatic evaluation metrics?

(b) **$\mathbf{RQ_2}$**: Can our technique outperform the existing baselines in terms of automatic evaluation metrics?

(c) **$\mathbf{RQ_3}$**: How do different components impact the overall performance of BugMentor?

(d) **$\mathbf{RQ_4}$**: How accurate, precise, useful, and concise are the answers from BugMentor?

3.4.1 Dataset Construction

Corpus Creation

To construct our corpus, we chose the top 20 popular projects from GitHub written in 4 different programming languages and collected 30,869 bug reports from them. We also capture a follow-up question and three candidate answers from each bug report. Finally, our corpus consisted of hundreds of bug reports where we capture the Bug ID, title, description, follow-up question and three candidate answers from each bug report. We apply standard natural language pre-processing to each item from our corpus. Please check Section 3.3.1 for further details on corpus construction.

Ground-Truth Construction

To evaluate BugMentor, we first construct a randomly sampled held-out dataset (i.e., 95% confidence level and 4.06% error margin) containing 550 bug reports (∼27 bug

Table 3.1: Inter-annotator Agreement

Bucket 1		**Bucket 2**		**Bucket 3**		**Bucket 4**		**Bucket 5**		**Bucket 6**	
Pairs	κ	**Pairs**	κ	**Pairs**	κ	**Pairs**	κ	**Pairs**	κ	**Pairs**	κ
A1 & A2	0.72	A2 & A3	0.58	A3 & A4	0.44	A4 & A5	0.66	A5 & A6	0.49	A1 & A2	0.41
A2 & A3	0.24	A3 & A4	0.43	A4 & A5	0.53	A5 & A6	0.40	A1 & A5	0.44	A2 & A6	0.29
A3 & A1	0.39	A4 & A2	0.58	A5 & A3	0.65	A6 & A4	0.27	A6 & A1	0.49	A6 & A1	0.40

reports x 20 systems). We then involve six human annotators (e.g., graduate students) to determine the ground truth answers against the follow-up question from each bug report. We divided 550 bug reports into six buckets (Table 3.1), each containing ∼90 bug reports, their questions, and candidate answers. Each bucket was annotated by three annotators, resulting in ∼270-275 bug reports per annotator. We used majority voting [87] to determine the ground truth answers. That is, the answer having the majority of votes was chosen as the ground truth answer against a follow-up question. When the answers did not have a clear majority, i.e. for 3% of the dataset, the three annotators engaged in discussions to resolve conflicts and determine the ground truth answer [87]. Each annotator spent ∼2.5-3 hours to complete the annotation.

We compute the Cohen's κ for all pairs of annotators, and the result is reported in Table 3.1. Although we use majority voting for annotation, our calculated metrics show the agreement level for each pair of annotators. We found an average of 0.46, which indicates a moderate agreement between any two annotators.

3.4.2 Evaluation Metrics

To evaluate BugMentor's answers against the ground truth, we use four relevant metrics from literature — BLEU Score [29], METEOR Score [30], Word Mover's Distance [32] and Semantic Similarity metric [31]. They are defined as follows:

1) BLEU — Bi-Lingual Evaluation of Understudy:

BLEU score is a commonly used metric for evaluating translation [29], which has found application in many software engineering tasks (e.g., comment generation [88]). It compares a candidate text to a reference text and determines how similar they are based on the matching of their n-grams. The BLEU score is calculated as follows:

$$BLEU = BP \cdot exp\left(\sum_{n=1}^{N} w_n log(p_n)\right) \tag{3.3}$$

where N is the maximum n-gram order, w_n is the weight assigned to the n-gram order, BP is the brevity penalty - a factor that penalizes the BLEU score when the candidate text is shorter than the reference text and p_n is the modified n-gram precision, which measures the ratio of the overlapping n-grams (between the candidate text and the reference text), and the total number of n-grams in the candidate text.

2) METEOR — Metric for Evaluation of Translation with Explicit ORdering:

The METEOR score is a metric for evaluating the quality of machine translation output based on both lexical and syntactic information [30]. It measures the similarity between a candidate text and the reference text by sequentially applying exact match, stemmed match and wordnet-based synonym match between the texts.

3) WMD — Word Mover's Distance:

WMD [32] is a similarity measure between two texts based on the meaning or relationships between their words. It is the minimum cost to transform one text into another by calculating the Euclidean distance between their word embeddings.

4) SS — Semantic Similarity:

In a recent work, Haque et al. [31] investigate which metric reflects human assessment of similarity the best. They suggest that Sentence-BERT [89] provides semantically meaningful sentence embeddings. Thus when a candidate text is compared with the reference text based on these embeddings using cosine-similarity, it has the highest correlation with human-evaluated similarity. Semantic similarity is computed as follows:

$$SemSim(ref, gen) = \cos(\mathrm{sbert}(ref), \mathrm{sbert}(gen)) \tag{3.4}$$

where $sbert(ref)$, $sbert(gen)$ are the numerical representations from Sentence-BERT for the reference text and generated text respectively.

3.5 Evaluation of BugMentor

3.5.1 Answering RQ$_1$ — How does our technique perform in answering follow-up questions in terms of different automatic evaluation metrics?

In this experiment, we analyze the performance of BugMentor using four different evaluation metrics - BLEU score [29], Semantic Similarity [31], METEOR [30] and WMD [32]. We divide our held-out dataset into four subsets based on their corresponding programming languages and report the results for each subset. Table 3.2 shows the performance details of BugMentor in various settings - within the project and cross-project. It should be noted that a higher value for BLEU, METEOR, and Semantic Similarity and a lower value for WMD metrics are desirable in our experiments.

Table 3.2: Performance of BugMentor

Metrics	**Top K**	**Python**	**Java**	**JavaScript**	**C++**	**Whole Dataset**
Within Project						
BLEU ↑	Top 1	26.20	22.36	26.06	25.73	24.47
	Top 3	29.25	32.70	31.76	28.03	28.90
	Top 5	34.12	33.82	32.38	30.25	31.94
METEOR ↑	Top 1	0.14	0.24	0.30	0.21	0.24
	Top 3	0.23	0.26	0.34	0.26	0.36
	Top 5	0.29	0.57	0.36	0.29	0.42
SS ↑	Top 1	43.20	42.90	46.30	46.13	50.63
	Top 3	58.80	42.90	45.50	54.30	53.49
	Top 5	64.50	54.70	54.20	56.10	57.01
WMD ↓	Top 1	5.09	4.97	4.91	5.35	4.18
	Top 3	4.89	4.64	4.75	4.93	3.80
	Top 5	4.82	3.27	4.29	4.56	3.65
Cross Project						
BLEU ↑	Top 1	16.71	14.24	12.36	16.03	14.84
	Top 3	20.52	15.12	16.99	18.97	17.90
	Top 5	21.86	17.70	19.30	19.92	19.70
METEOR ↑	Top 1	0.12	0.13	0.11	0.09	0.11
	Top 3	0.14	0.16	0.13	0.11	0.14
	Top 5	0.15	0.19	0.14	0.12	0.15
SS ↑	Top 1	26.80	26.23	27.76	27.30	27.02
	Top 3	29.15	29.52	29.75	29.27	29.42
	Top 5	31.89	31.80	30.93	30.75	31.34
WMD ↓	Top 1	5.23	5.62	5.10	5.43	5.34
	Top 3	5.02	5.64	5.01	5.38	5.26
	Top 5	4.87	5.68	4.98	5.36	5.22

BugMentor achieves an average BLEU Score of 24.47 for Top 1 answer, and a maximum of 31.94 for Top 5 answers when our whole dataset is considered. In the case of cross-project setting, these scores drop to 14.84 and 19.70, respectively. However, our technique achieves a maximum of 26.20 for the Top 1, 32.7 for Top 3 and 34.12 for Top 5 answers across all four subsets. These BLEU scores indicate that our generated answers are *understandable* and *good* according to Google's AutoML Translation documentation [33]. This also shows that the answers generated by BugMentor have a significant overlap with the ground truth in terms of words and word order. However, BLEU score primarily focuses on capturing the precision of an answer against the ground truth. Hence we also evaluate our answers using the METEOR score, where recall is captured by taking into account additional information such as synonyms, word forms, and sentence structure [30].

As shown in Table 3.2, BugMentor achieves an average METEOR score of 0.24 for Top 1 answer, and a maximum of 0.42 for Top 5 answers against the whole dataset, which are considered to be reasonable [90]. In cross-project setting, our technique achieves an average METEOR score of 0.11 for Top 1 answer, and a maximum of 0.15 for Top 5 answers. It achieves a maximum of 0.30 for Top 1, 0.34 for Top 3, and 0.57 for Top 5 answers across four subsets. This shows that BugMentor was able to produce a significant part of the ground truth texts in the generated answers. However, since BLEU and METEOR scores rely on keyword matching between a generated answer and the ground truth answer, they may not capture the semantic relevance between them. Hence we also evaluate our technique using WMD [32] and Semantic Similarity [31]. They also have been shown to correlate better with human judgement of relevance [29], [91].

In Table 3.2, we find that BugMentor achieves an average WMD of 4.18 for Top 1 answer, and a minimum of 3.65 for Top 5 answers, when the whole dataset is considered. In cross-project setting, our technique achieves an average WMD of 5.34 for Top 1 answer, and a minimum of 5.22 for Top 5 answers. These distance scores show that BugMentor was able to generate answers semantically similar to the ground truth and were worded closely to the ground truth. However, WMD may not be sufficient to reflect the importance and context of words in a sentence [91], [92]. We thus also evaluate our answers using semantic similarity against the ground truth. The metric is appropriate when there may not be any syntactic overlap between the answers, which is a common phenomenon in question answering, according to existing literature [93], [94].

From Table 3.2, we also find that the answers from BugMentor have an average Semantic Similarity score of 50.63% for Top 1 answer, and a maximum of 57.01% for Top 5 answers when the whole dataset is considered. It achieves an average Semantic Similarity score of 27.02% for Top 1 answer, and a maximum 31.34% for Top 5 answer in cross-project setting. BugMentor achieves a maximum of 46.13% for Top 1, and a maximum of 64.5% for Top 5 answers when all subsets are considered. All these numbers indicate a high similarity in meaning and content between BugMentor's answers and the ground truth.

Summary of RQ$_1$: BugMentor can generate relevant answers to follow-up questions that are *understandable* to *good* according to Google's Standard and achieves an average BLEU score of 31.94. Its answers also have a high semantic overlap with the ground truth answers and thus achieve an average semantic similarity score of 57%.

3.5.2 Answering RQ$_2$ — Can our technique outperform the existing baselines in terms of automatic evaluation metrics?

To the best of our knowledge, there exists no work that can offer relevant answers to follow-up questions from bug reports. However, Lucene [34] is a popular IR-based tool that has been used to recommend answers in the programming Q&A site [95], [96]. CodeT5 [35] is a large language model for generating answers to questions. AnswerBot [36] can synthesize answers for technical, non-factoid questions on StackOverflow. We thus consider them as our baselines for the comparison. We call them Baseline$_{Lucene}$, Baseline$_{CodeT5}$ and AnswerBot, respectively in this experiment. We evaluate the answers from all three baselines against the ground truth using four evaluation metrics. Tables 3.3 and 3.4 show the comparison details between BugMentor and these baseline techniques.

To implement Baseline$_{Lucene}$, we provide a follow-up question as the *query* and all candidate answers as the *corpus* to the Lucene tool. Then we collect the top K answers from the tool by executing the query for our evaluation. Baseline$_{Lucene}$

Table 3.3: Comparison of BugMentor with Baseline$_{Lucene}$ and Answerbot

Metrics	**Top K**	**AnswerBot**	**Baseline$_{Lucene}$**	**BugMentor**
BLEU ↑	Top 1	4.31	8.60	**24.47**
	Top 3	6.71	10.54	**28.9**
	Top 5	4.24	14.31	**31.94**
METEOR ↑	Top 1	0.05	0.18	**0.24**
	Top 3	0.12	0.19	**0.36**
	Top 5	0.07	0.21	**0.42**
SS ↑	Top 1	36.47	42.75	**50.63**
	Top 3	47.69	45.45	**53.49**
	Top 5	47.69	48.78	**57.01**
WMD ↓	Top 1	5.7	4.55	**4.18**
	Top 3	4.81	4.46	**3.80**
	Top 5	5.01	4.37	**3.65**

Table 3.4: Comparison of BugMentor with Baseline$_{CodeT5}$

Metrics	**Top K**	**Baseline$_{CodeT5}$**	**BugMentor**
BLEU ↑	Top 1	2.4	**24.47**
METEOR ↑		0.04	**0.24**
SS ↑		11.01	**50.63**
WMD ↓		7.32	**4.18**

achieves a BLEU score of 8.60, METEOR score of 0.18, Semantic Similarity of 42.75 and WMD of 4.55. On the other hand, BugMentor achieves a BLEU score of 24.47, METEOR score of 0.24, Semantic Similarity of 50.63 and WMD of 4.18. Thus, our technique outperforms the baseline in all four metrics.

To implement Baseline$_{CodeT5}$, we provide a follow-up question as the *query* and its corresponding bug report as *context* to the CodeT5 model, which generates an answer. We observe that Baseline$_{CodeT5}$ performs significantly poorly when compared to BugMentor and Baseline$_{Lucene}$. For example, Baseline$_{CodeT5}$ achieves a BLEU score of 2.4, METEOR score of 0.04, Semantic Similarity of 11.01 and WMD of 7.32, which are 42%-90% lower than the corresponding measures from BugMentor.

To implement AnswerBot, we use the replication package provided by the authors [36], [97]. We provide a follow-up question as the *query* and all bug reports along with their candidate answers as the *corpus.* We observe that AnswerBot performs significantly poorly when compared to BugMentor and Baseline$_{Lucene}$. For example, AnswerBot achieves a maximum BLEU score of 6.71, METEOR score of

0.12, Semantic Similarity of 47.69 and a minimum WMD of 4.81, which are lower than the corresponding measures from BugMentor.

We also perform Mann-Whitney Wilcoxon test [98] to check if the performance of Baseline$_{Lucene}$, Baseline$_{CodeT5}$, AnswerBot are significantly lower than that of BugMentor using BonFerroni Correction [99]. We find that BugMentor performs significantly higher than Baseline$_{Lucene}$ and AnswerBot, i.e., p-value = $0.010<0.016$ in terms of all four metrics.

Besides the comparison with traditional baselines, it is important to consider the other the competitive landscape, particularly with the emergence of popular Language Model-based approaches (LLMs) such as ChatGPT. Upon conducting a limited qualitative analysis, we observe that while ChatGPT exhibit a profound understanding of the bug report context, they fall short in providing answers to follow-up questions with the desired level of specificity. Furthermore, the responses generated by LLMs tend to be more verbose, lacking the precision found in BugMentor's answers. BugMentor gains a deeper understanding of the bugs from historical bug reports, excels in capturing a broader context, and thus was able to generate more accurate responses to follow-up questions. In future, we plan to extensively compare BugMentor with the modern LLM-based approaches such as ChatGPT.

Summary of RQ$_2$: BugMentor performs better in answer generation than all three baselines in terms of four evaluation metrics. According to the statistical significance test, BugMentor outperforms the closest competitors Baseline$_{Lucene}$ and AnswerBot – by a statistically significant margin.

3.5.3 Answering RQ$_3$ — How do different components impact the overall performance of BugMentor?

Our technique has three key components — (a) structured information retrieval, (b) embedding similarity-based ranking and (c) neural text generation. In this experiment, we conduct an ablation study to determine the contribution of each component. In particular, we design different variants of BugMentor with each component and evaluate their performance in answer generation. Table 3.5 summarizes our results from the ablation study.

Table 3.5: Experimental Results from the Ablation Study

Metrics	**Top K**	**Components**				
		BugMentor$_L$	**BugMentor$_{CT5}$**	**BugMentor$_{L+E}$**	**BugMentor$_{L+CT5}$**	**BugMentor**
BLEU ↑	Top 1	7.62	2.40	5.65	23.58	**24.47**
	Top 3	12.05		6.63	30.50	**28.90**
	Top 5	19.40		8.35	33.41	**31.94**
METEOR ↑	Top 1	0.21	0.04	0.28	0.26	**0.24**
	Top 3	0.23		0.28	0.30	**0.36**
	Top 5	0.25		0.31	0.32	**0.42**
SS ↑	Top 1	50.36	11.01	53.28	45.48	**50.63**
	Top 3	53.96		53.49	50.38	**58.11**
	Top 5	57.09		60.33	53.69	**57.01**
WMD ↓	Top 1	4.29	7.32	3.97	4.92	**4.18**
	Top 3	4.15		3.89	4.62	**3.80**
	Top 5	4.02		3.72	4.42	**3.65**

BugMentor$_L$ = BugMentor$_{Lucene}$, **BugMentor$_{CT5}$** = BugMentor$_{CodeT5}$,
BugMentor$_{L+E}$ = BugMentor$_{Lucene+Embedding}$,
BugMentor$_{L+CT5}$ = BugMentor$_{Lucene+CodeT5}$

We find that BugMentor outperforms all of its variants based on either individual components or their combinations. The combination of structured information retrieval and neural text generation (a.k.a., BugMentor$_{Lucene+CodeT5}$) is a close second when compared using BLEU and Semantic Similarity scores. We see that the absence of the embedding component reduces its performance from that of BugMentor by 3.63% in terms of BLEU and 10.17% in terms of Semantic Similarity and increases in WMD by 17.70% when the Top 1 answer is captured. Thus, the Embedding component improves the semantic closeness between the generated answers and ground truth.

From Table 3.5, we also note that the combination of Lucene and Embedding (a.k.a., BugMentor$_{Lucene+Embedding}$) is the closest competitor to BugMentor when BLEU is considered. However, the absence of the text generation component (a.k.a., CodeT5) reduces the performance of the variant by 76.91% in terms of BLEU. It also shows a decrease in performance by 22.22% in terms of METEOR, 7.95% in terms of Semantic Similarity and an increase in WMD by 2.36% when Top 3 answers are considered. Thus, the CodeT5 component has a significant impact on improving not only the understandability of our generated answers (i.e., BLEU) but also their syntactic and semantic relevance to the ground truth answers (i.e. METEOR, WMD, Semantic Similarity).

From Table 3.5, we also find that the components from BugMentor do not perform well when evaluated individually. For example, if we omit both Embedding and CodeT5 components and only use Lucene, the performance degrades by 68.85% in terms of BLEU, 12.5% in terms of METEOR and 0.53% in terms of Semantic Similarity and WMD increases by 2.63% for Top 1 answer. Similarly, if we use only the CodeT5 component, the performance degrades by 90.19% in terms of BLEU, 83.33% in terms of METEOR and 78.25% in terms of Semantic Similarity and WMD increases by 75.11% for Top 1 answer.

Summary of RQ$_3$: Our ablation study demonstrates the contribution of the three components — structured information retrieval, embedding similarity-based ranking, and neural text generation (a.k.a CodeT5) — towards the overall performance of BugMentor. We also found that BugMentor outperforms its four variants based on individual components or their combinations, which justifies the presence of all three components in BugMentor.

3.5.4 Answering RQ$_4$ — How accurate, precise, useful, and concise are the answers from BugMentor?

The metric-based evaluation above demonstrates the benefits of our technique in answering follow-up questions using four similarity measures. We conduct a developer study to further demonstrate the benefits of our technique in a practical setting. Given a bug report (e.g., title, description, and follow-up question), we evaluate how *accurate*, *precise*, *useful*, and *concise* (details in Table 3.6) our answers are according to human developers.

Table 3.6: Quality Aspects of the Generated Answers

Quality	**Overview**
Accurate	It provides the same factual information as the reference.
Precise	It can answer the question completely
Concise	It is short and still answers the question.
Useful	The provided information has the potential to answer the question.

Study Participants: The target group for our study consists of English-speaking software developers with professional experience in four programming languages — Python, Java, JS, and C++ — that were used by our collected projects. We sent an open invitation through our personal connections, and 10 graduate students responded to our invitation. Each of them had professional experience in software development and at least two years of experience in the aforementioned programming languages. We provide them with a quick overview of our project using relevant examples and a secured link containing our survey. None of the participants knew the specifics of our designed technique — BugMentor.

Study Setup: For our study, we use 12 use cases where each use case consists of a bug report and a follow-up question. To select these use cases, we apply random sampling without replacement to the held-out dataset (Section 3.4.1). To avoid information overload, we select such bug reports that (a) do not have any stack trace information and (b) do not warrant any project-specific knowledge to understand the bug. We select three randomly sampled bug reports from each subset (based on programming language, Section 3.4.1) and collect corresponding follow-up questions, ground truth answers, and the answers from both BugMentor and three baseline techniques — AnswerBot, Baseline$_{Lucene}$ and Baseline$_{CodeT5}$.

We present all 12 use cases to each of the participants. Then the participants were instructed to assess the accuracy, precision, usefulness and conciseness of the generated answers (by BugMentor and baselines) with respect to the ground truth answers. We also instruct the participants to submit their evaluation on a five-point Likert scale, where 1 indicates strongly disagree and 5 indicates strongly agree. **We also anonymize the source of all generated answers to avoid any potential bias towards any of the techniques.** We thus collect a total of 192 data points (12 questions $\times$ 4 explanations $\times$ 4 evaluation aspects) from each of the 10 participants.

The design of our developer survey has been reviewed and approved by the Dalhousie University Research Ethics Board (REB file #: 2023-6885).

Study Results and Discussion: Table 3.7 summarizes our findings from the developer study. We note that, on average, the participants found the answers from BugMentor to be the most accurate, precise, useful, and concise. Based on the median and mode values, we see that the participants agree with our answers the

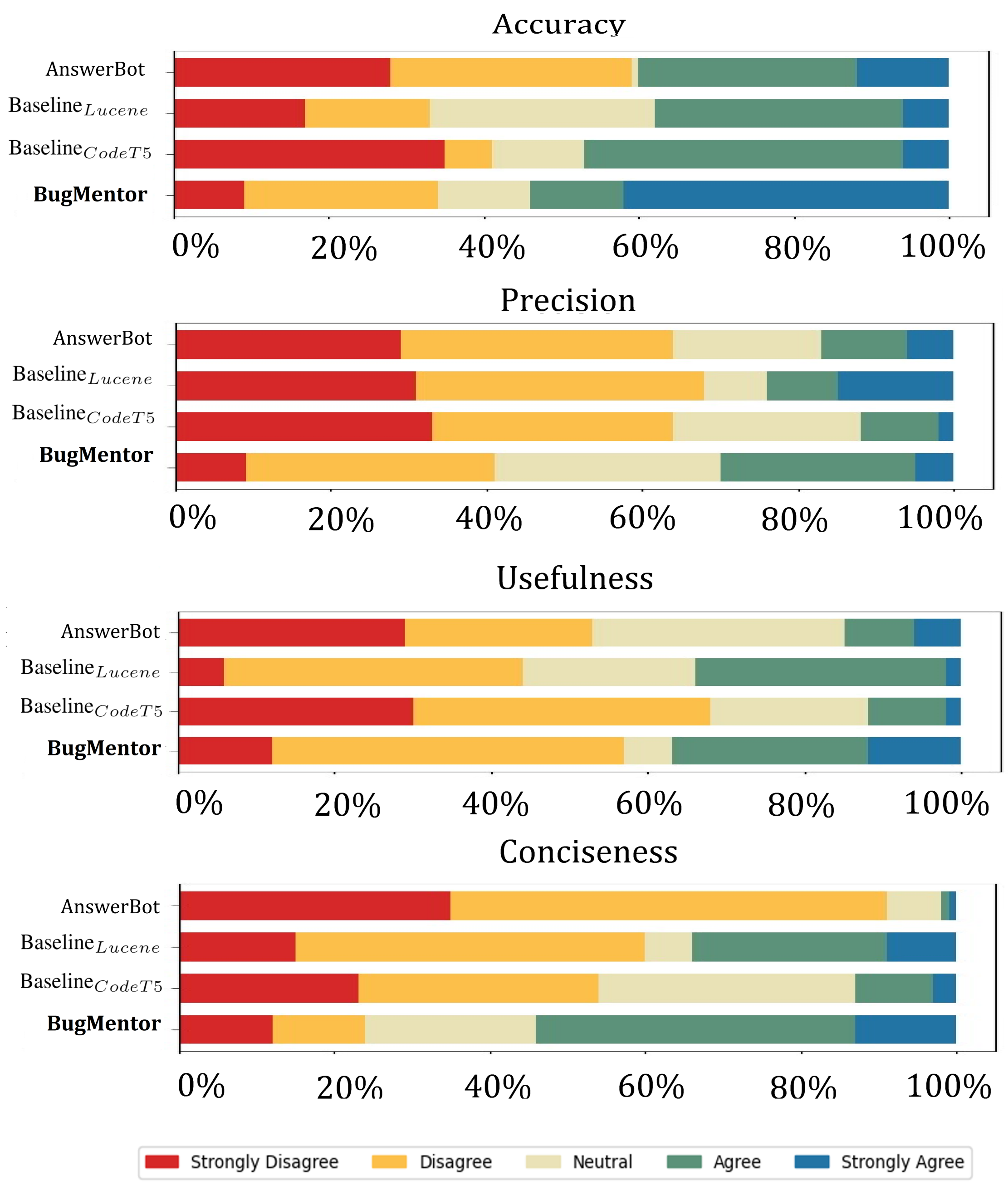


Figure 3.4: Comparison of BugMentor with the baseline techniques using the Likert scores

most. Similar to the findings in RQ_2, the participants found the closest competitor of BugMentor to be $Baseline_{Lucene}$ in terms of precision, usefulness and conciseness and $Baseline_{CodeT5}$ in terms of accuracy. According to the median values, the developers agree with AnswerBot, $Baseline_{Lucene}$ and $Baseline_{CodeT5}$ in several cases. However,

Table 3.7: Comparison of BugMentor with the Baseline Techniques using a Developer Study

Quality	**Model**	**Mean**	**Median**
Accurate	Baseline_{Lucene}	3.03	3
	Baseline_{CodeT5}	2.55	3
	AnswerBot	2.65	2
	BugMentor	**3.10**	**3**
Precise	Baseline_{Lucene}	2.25	2
	CodeT5	2.56	3
	AnswerBot	2.34	2
	BugMentor	**2.96**	**3**
Useful	Baseline_{Lucene}	2.62	2
	Baseline_{CodeT5}	2.46	2
	AnswerBot	2.4	2
	BugMentor	**2.94**	**3**
Concise	Baseline_{Lucene}	2.68	2
	AnswerBot	1.7	2
	Baseline_{CodeT5}	2.95	3
	BugMentor	**3.32**	**4**

based on the mode values, we note that the participants disagree with their answers in many more cases.

Fig. 3.4 shows the distribution of participants' agreement levels with different quality aspects of the answers. We see that the participants strongly agree with BugMentor for a substantial part of the time (e.g., ~40% for accuracy), and strongly disagree only a few times (e.g., <20% times), which none of the baselines achieved. On the other hand, nearly half of the time, the participants disagree with AnswerBot, Baseline_{Lucene} and Baseline_{CodeT5} regarding various quality aspects of their provided answers.

We also perform Mann-Whitney Wilcoxon test [98] to check if the developers' preferences to Baseline_{Lucene}, Baseline_{CodeT5} and AnswerBot are significantly lower than that of BugMentor using BonFerroni Correction [99]. We find that the preference levels for BugMentor are significantly higher than all three baselines, i.e., $p = 0.0010 < 0.016$ for Baseline_{Lucene}, $p = 0.0016 < 0.016$ for Baseline_{CodeT5}, and $p = 0.00016 < 0.016$ for AnswerBot.

Table 3.8: Manual Analysis

Dataset	**AC**	**AP**	**AP + AddInfo**	**AddInfo**
Java	7	9	12	20
Python	8	9	17	14
C++	4	4	16	24
JavaScript	8	5	16	19
Average%	**14.06**	**14.06**	**31.77**	**40.10**

AC = Answers Completely, **AP** = Answers Partially,
AddInfo = Additional Information

Manual Analysis: To further investigate the usefulness of BugMentor's answers, we perform a manual analysis on 192 bug reports (i.e. 48 samples for each programming language). We select these samples from the whole collection with a 95% confidence level and 4.95% error margin. We collect the bug reports, follow-up questions, ground truth and generated answers. Table 3.8 shows the summary of our analysis.

We analyze our generated answers to the follow-up questions, contrast them against the ground truth, and determine whether they respond to the question completely, partially or simply provide additional information. We find that BugMentor, on average, was able to answer the questions completely for 14.06% of the cases from each programming language. It was able to answer 14.06% of the questions partially while adding complementary information to 31.77% of the answers. Furthermore, in 40% cases, our technique delivered such answers that did not match with the ground truth answers but were complementary or somewhat relevant to the questions. For example, let us consider the bug report shown below discussing the issue of links in Atom. It does not provide the version of the operating system that the reporter uses. The ground truth answer indicates the OS version and the issue persistence in the safe mode of the browser. We see that BugMentor's answer captures the context and points out that the problem might lie with Ubuntu version rather than other running applications. Thus, BugMentor can provide complementary information that can benefit the developer.

Title: Links do not work in Atom
Description: After upgrading to Atom, links no longer open in a new tab in Chrome. For example, clicking on any of the release notes links does nothing.

Question: What OS do you experience this issue in safe mode?
Actual Answer: Ubuntu; safe mode exhibits the same issue.
Generated Answer: Since upgrading to Ubuntu, I've had no issues. However, similar problems with other apps make me suspect it's related to the Ubuntu version. Even in safe mode, Atom still exhibits the problem.

Summary of RQ$_4$: Developers with professional experience found the answers of BugMentor to be accurate, precise, concise, and useful, with respect to the ground truth answers. Their preference levels for BugMentor were also higher than those of the three baseline techniques by a statistically significant margin.

3.6 Related Work

Question Answering (QA) has been an active research topic in both Information Retrieval (IR) and Natural Language Processing (NLP) communities [21]–[23], [36], [57]–[62]. There also have been several works that focus on question-answering in the context of software engineering. Breu et al. [10] first analyzed follow-up questions from bug reports and found that 32.34% of them were never responded to. Recently, Imran et al. [9] proposed Bug-AutoQ that recommends follow-up questions against a deficient bug report leveraging development history using information retrieval. However, their technique does not answer the follow-up questions.

Murgia et al. [100] leverage the search feature of StackOverflow Q&A site to suggest relevant questions against error messages from a version control system. However, their technique was trained to provide only simple, recurring questions related to Git error messages. Tian et al. [21] propose APIBot that can answer questions related to an API by analyzing the relevant API documentation. However, their solution was limited to API-related questions only. Bansal et al. [22] design a context-aware QA system to answer basic questions about subroutines. Lu et al. [23] propose another QA approach that can provide answers by executing structured queries generated from a bug report template. Xu et al. [36] designed AnswerBot to synthesize answers for technical, non-factoid questions from StackOverflow. However, they only use the title of a question overlooking the detailed problem context (e.g., question body), and thus their answers might be unaware of the problem context. We compare BugMentor with AnswerBot using experiments, and the detailed comparison can be found in Section 3.5.2. Abdellatif et al. [58] designed MSRBot to answer the most common questions related to software development and maintenance. However, their answers might be limited by the available information in the mined repositories. Song et al. [101] designed BURT to support bug reporters of Android applications, but their approach might not generalize to other software applications.

Recently, Language Model-based approaches (LLMs) such as ChatGPT have emerged as a powerful text generation tool. After conducting a limited qualitative analysis (Section 3.5.2), we note that while ChatGPT exhibit an understanding of a given bug report, they often struggle to come up with precise answers to follow-up questions. BugMentor has a better understanding of past bug reports and thus captures a broader context. In the future, we plan on conducting a thorough comparison between BugMentor and contemporary LLM-based approaches like ChatGPT.

In short, existing relevant works focus on improving deficient bug reports and answering specific questions related to API, subroutines and Git error messages. To the best of our knowledge, our proposed technique is the first to automatically answer the follow-up questions from bug reports, which makes our work *novel*. We also combine structured information retrieval with neural text generation (e.g., CodeT5) to generate the answers, which were found to be meaningful, accurate, precise, useful, and concise according to two types of evaluation — automated metrics and developer study. Our technique also outperforms three baselines.

3.7 Threats to Validity

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

External Validity: Threats to external validity refer to the lack of generalizability in the findings [102]. One threat could stem from our selection of subject systems. We select 20 software systems written in four programming languages: Python, Java, JavaScript, and C++, which might not represent all systems at GitHub. However, the underlying algorithm of BugMentor is not bound to any programming language and thus can be easily adapted to any other platforms.

Another threat stems from the small sample size of the held-out dataset for evaluation (e.g., 550). However, to mitigate this concern, we selected them carefully through random sampling from all four subsets (95% confidence level, 4.06% error margin, Section 3.4.1). We also maintain diversity in selecting our 20 subject systems (Section 3.4.1).

Construct Validity: Construct validity refers to the extent to which the experiment measures what it intends to measure [103]. Inappropriate use of evaluation metrics could be a threat to construct validity. However, we chose our evaluation metrics — BLEU, METEOR, Semantic Similarity, and WMD — based on relevant literature [29]–[32]. We also chose the four quality aspects of generated answers based on relevant literature [9], [104]. Thus, threats to construct validity might be mitigated.

Internal Validity: Threats to internal validity relate to experimental errors and subjective biases [105]. We use manually annotated ground truth to answer both RQ1 and RQ2, which could be a source of threat. However, to mitigate this, the annotators were given appropriate training for their annotation tasks. We also employ majority voting [87] for decision-making and calculate Cohen's κ to demonstrate the agreement levels between annotators [87]. In the developer study, the assessment of answers can be influenced by subjective bias. However, we anonymize the source of all answers to avoid any bias towards any technique. Another source of threat could be the replication of the baseline techniques. For the replication of CodeT5, we collected the pre-trained model from HuggingFace [106], and for the replication of Lucene, we used Elastic-Search [78], a standard library. To replicate AnswerBot [36], we used the replication package from the original authors [97]. Furthermore, we followed the documentation closely for any customizations. Thus, threats to internal validity might be mitigated.

3.8 Summary

To summarize, in this study, we propose BugMentor, a novel technique to answer follow-up questions from deficient bug reports by combining structured information retrieval and neural text generation. Our technique leverages the relevance between past and current bug reports to gather additional context, which helps us generate an appropriate answer to the question. Our evaluation using four performance metrics shows that BugMentor can generate understandable and good answers to follow-up questions, as per Google's Standard. Our technique outperforms three existing baselines. We also evaluate BugMentor using a user study using 10 developers. The developers found the answers from BugMentor to be more accurate, precise, concise and useful compared to the baseline answers. Thus, BugMentor has the potential to support bug resolution with complementary information in the form of answers to follow-up questions. However, newcomers or novice developers often struggle to understand bug reports due to their lack of in-depth knowledge about an application. Thus we perform another study to further improve the quality of bug reports by providing explanations to their domain-specific terms or jargon.

Chapter 4

BugEnricher: Explaining Domain-specific Terms and Jargon from Bug Reports with Neural Machine Translation

Existing studies have shown that about 78% of bug reports from open-source projects (e.g., Eclipse, Firefox) include less than 100 words each and claim more time from developers for bug resolution [5]. Our first study in Chapter 3 aims to support the developers by generating answers to follow-up questions from deficient bug reports. While our answers have been found useful, novice developers might need more help in their bug understanding. In this chapter, we propose — *BugEnricher* — that can supplement bug reports with meaningful explanations to their domain-specific terms or jargon. Our evaluation using three performance metrics (e.g., BLEU, METEOR, Semantic Similarity) shows that *BugEnricher* can generate understandable and good explanations according to Google's standard and can outperform existing baselines from the literature.

The rest of this chapter is organized as follows. Section 4.1 introduces our study and reports the gap in the literature and our contribution. Section 4.2 illustrates the usefulness of our technique with a motivating example. Section 4.3 presents our proposed technique for explaining software-related terms. Section 4.4 discusses our experimental design and datasets. Section 4.5 discusses our evaluation results. Section 4.6 discusses relevant studies from the literature. Section 4.7 identifies possible threats to the validity of our work. Finally, Section 4.8 summarizes this study.

4.1 Introduction

Software bugs are human-made mistakes that prevent a software system from operating as expected. According to a recent study [2], [3], software bugs cause the global economy to suffer enormously and lose billions of dollars every year. Bug finding and corrections take up approximately 50% of a developer's programming time. Bug resolution is, therefore, one of the most challenging issues in software maintenance [3]. Hundreds of software bugs are submitted as *bug reports* to bug-tracking systems such as GitHub and JIRA [4]. The developers then examine and resolve these bugs by carefully analyzing the corresponding bug reports.

Given a reported bug, developers need to first understand its root cause and symptom before they come up with a solution [107]. A recent study suggests that information in the majority of bug reports is incomplete and inaccurate [108], [109]. Zhang et al. [5] found that up to 78% of bug reports from four open-source projects (e.g., Eclipse, Mozilla, Firefox, GCC) contain less than 100 words each (a.k.a., short bug reports). These short bug reports, on average, took 121 extra days to get resolved when compared to the well-written bug reports due to the lack of information [5]. Thus, understanding the bug reports could be a challenge due to incomplete or inaccurate information with complex problem context [11]. This challenge could exacerbate for newcomers or novice developers to a project who need additional assistance to understand or resolve a bug. According to a recent study [12], even with prior experience, developers often need help to acquire a comprehensive understanding of any application domain and understand the discussions from a bug report. One significant obstacle to bug understanding for novice developers could be the lack of explanation for the domain-specific terms or jargon in the bug reports.

There have been existing studies to support newcomers or inexperienced developers who may struggle to comprehend the bug reports. An existing survey by Tan et al. [24] found that a clear bug description, which does not rely on in-depth domain knowledge, is crucial to assist newcomers in understanding and resolving the bug. Recently, Correa et al. [25] suggest that including web links in issue tracker discussions can benefit developers by providing external knowledge sources or artifacts. Zhang et al. [5] recommend complementing bug reports with carefully curated sentences from relevant past bug reports. Dit et al. [26] propose a technique that suggests relevant comments from past bug reports so that developers can make explicit connections between the suggested and existing comments. Including such comments in a bug report can be helpful for the developers to gain a better understanding of the issue. While the above approaches offer complementary information to support bug understanding, they do not focus on domain-specific terms or jargon, which warrants further investigation.

In this chapter, we propose a novel technique — BugEnricher — that can supplement bug reports by generating explanations for their domain-specific terms and jargon using neural text generation. First, we construct a vocabulary for two popular programming languages — Java and Python. We scrape the domain-specific terms or jargon and their explanations from three different sources — StackOverflow, Glossary, and API documentation. Second, we fine-tune a transformer-based text-generation model (e.g., T5) with the domain-specific terms or jargon and their corresponding explanations collected above. Third, we generate the explanations for domain-specific terms from bug reports using our fine-tuned model and examine their effectiveness using a case study.

We collect 28,7690 Java, 21,365 Python and 141,567 miscellaneous domain-specific terms or jargon and their explanations from the aforementioned sources for our experiments. We evaluate our technique — BugEnricher — using three popular metrics on text generation, namely BLEU score [29], METEOR [30], Semantic Similarity Score [31]. We achieve a BLEU score of 28.85, which is understandable to good according to Google AutoML documentation [33]. Our technique also outperforms two baselines — AnswerBot [36] and T5 [37] — in all three metrics. We also conduct a case study using duplicate bug reports and attempt to enrich duplicate bug reports that are textually dissimilar [39]. We find that the enrichment of bug reports by BugEnricher led to an improvement in the performance of an existing technique for duplicate bug detection. Thus, the empirical findings above suggest that our technique has the potential to enrich a bug report significantly, which could lead to improved bug understanding and management.

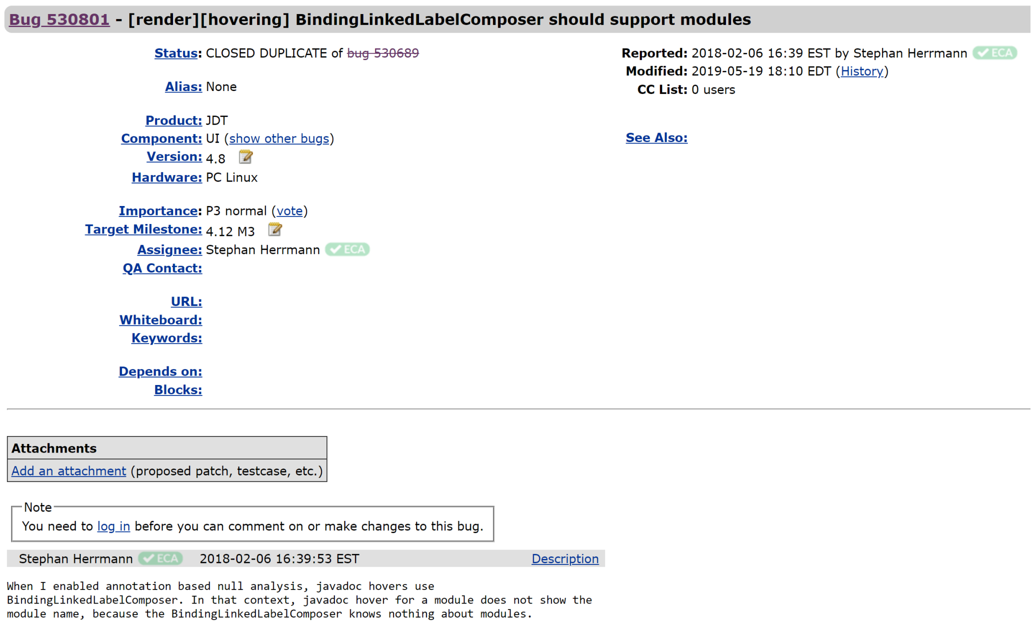
We thus make the following contributions in this study:

(a) A large dataset of 141,567 domain-specific terms and jargon and their corresponding explanations that are carefully curated from Stack Overflow Q&A site, glossary, and API documentation.

(b) A novel approach — BugEnricher — that can complement bug reports with meaningful explanations of their domain-specific terms or jargon using neural text generation (e.g., fine-tuned T5).

(c) A replication package [110] that includes our working prototype, experimental dataset, and other configuration details for the replication or third-party reuse.

4.2 Motivating Example



Bug 530801 - [render][hovering] BindingLinkedLabelComposer should support modules

Status: CLOSED DUPLICATE of ~~bug 530689~~

Alias: None

Product: JDT
Component: UI (show other bugs)
Version: 4.8
Hardware: PC Linux

Importance: P3 normal (vote)
Target Milestone: 4.12 M3
Assignee: Stephan Herrmann ✓ ECA
QA Contact:

URL:
Whiteboard:
Keywords:

Depends on:
Blocks:

Reported: 2018-02-06 16:39 EST by Stephan Herrmann ✓ ECA
Modified: 2019-05-19 18:10 EDT (History)
CC List: 0 users

See Also:

Attachments
Add an attachment (proposed patch, testcase, etc.)

Note
You need to log in before you can comment on or make changes to this bug.

Stephan Herrmann ✓ ECA 2018-02-06 16:39:53 EST Description

```
When I enabled annotation based null analysis, javadoc hovers use
BindingLinkedLabelComposer. In that context, javadoc hover for a module does not show the
module name, because the BindingLinkedLabelComposer knows nothing about modules.
```

Figure 4.1: An example of a bug report from BugZilla (ID #530801)

Table 4.1: Generated Explanations by BugEnricher

Domain-Specific Terms or Jargon	**Explanations**
Javadoc	It is documentation generated
BindingLinkedLabelComposer	It is for composing labels
annotation	It is used to describe an annotation object
null-analysis	It is a Java library for analyzing null data
module	It is a unit of Java code

To demonstrate the potential benefits of our work, let us consider the example bug report in Fig. 4.1. It has been taken from the *Eclipse* project on BugZilla[1]. The example report discusses a bug related to BindingLinkedLabelComposer that lacks awareness of modules. The problem stems from Javadoc when using the annotation-based null analysis. Specifically, the module name does not appear when hovering over a module. Table 4.1 shows the explanations generated by BugEnricher. These explanations were used to enrich the example bug report, and the enriched bug report can be found below.

[1]https://bugs.eclipse.org/bugs/show_bug.cgi?id=530801

Enriched Bug Report
When I enabled annotation (It is used to describe an annotation object) based null analysis (It is a Java library for analyzing null data), Javadoc (It is documentation generated) hovers use BindingLinkedLabelComposer (It is for composing labels). In that context, Javadoc hover for a module (It is a unit of Java code) does not show the module name, because the BindingLinkedLabelComposer knows nothing about modules.

In our case study, the enriched bug report improved the rank of its duplicate report from the 19th to the 13th position in the ranked list when detected by a BM25-based technique [38], [39]. This significant improvement in the ranking highlights the importance of our provided explanations, demonstrating an improvement in the quality of bug reports.

4.3 Methodology

As input, our technique takes a bug report containing domain-specific terms or jargon that require additional explanation. As output, it generates explanations for those terms. Fig. 4.2 shows the schematic diagram of our proposed technique. In the following sections, we discuss different steps of our approach.

4.3.1 Vocabulary Construction

First, we construct a vocabulary of domain-specific terms along with their meanings (a.k.a., explanations) for two popular programming languages — Python and Java. We collect them from three different sources — StackOverflow Tags, API Documentation and Glossary.

(a) StackOverflow Tags

To collect the domain-specific terms and their meanings, we use StackOverflow as our first source. Each post on StackOverflow consists of several tags that convey the key concepts of the post. The meaning of each tag is defined on StackOverflow. We collect 10,022 Java tags, 9,594 Python tags and 105,822 miscellaneous tags from Stack Exchange Data Explorer (SEDE) using the SQL query — "select $*$ TagName

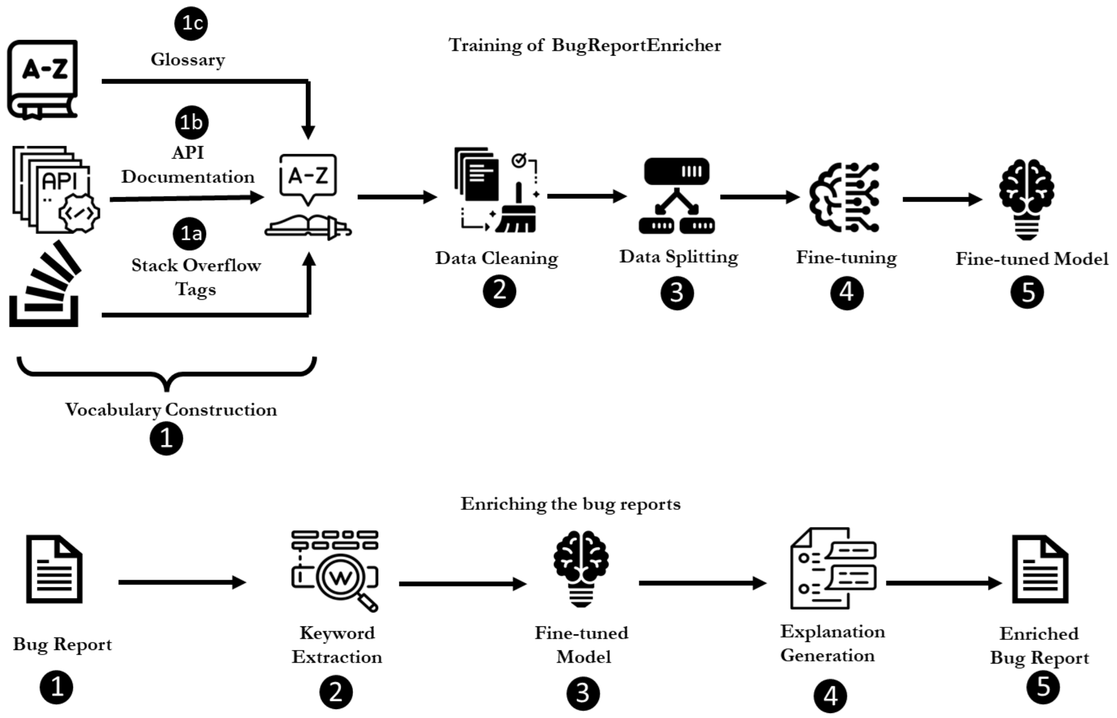


Figure 4.2: Schematic diagram of BugEnricher

from Tags". The "Tags" table contains all the name of the tags in the "TagName" field. To capture the Java or Python related tags, we collect a list of Tags that contain the keyword "Java" and "Python" or "py" in their names. We then scrape the explanations of all the collected Tag names using Beautiful Soup[2]. An example of the StackOverflow tag is shown below.

Tag and Explanation at StackOverflow

Tag: google-chrome-extension

Explanation: Extension development for the Google Chrome web browser. You write them using web technologies such as HTML, JavaScript, and CSS.

Java related Tag and Explanation at StackOverflow

Tag: javafx-11

Explanation: The JavaFX platform enables developers to create client applications based on JavaSE that behave consistently across multiple platforms. Built on Java

[2]https://pypi.org/project/beautifulsoup4/

technology since JavaFX 2.0, it was part of the default JDK since JDK 1.8, but starting Java 11, JavaFX is offered as a component separate from the core JDK.

Python related Tag and Explanation at StackOverflow

Tag: python-mode

Explanation: Python-mode is a vim plugin that helps you to create Python code very quickly by utilizing libraries including pylint, rope, pydoc, pyflakes, pep8, and mccabe for features like static analysis, refactoring, folding, completion, documentation, and more.

(b) API Documentation

We collect the API documentation of the most recent stable version of both Python (3.11) and Java (17) programming languages from their official documentation [111], [112]. We use Beautiful Soup[3] and Request[4] libraries for all our scraping.

First, we scrape the overview page from the official documentation containing the names of all the modules, their explanations and the URLs to further description of the defined packages and services. We store these module names and their explanations. Then, from the URLs collected in the previous step, we further scrape the package and service names, their explanations, and the URLs to further description of the defined classes and interfaces. Similarly, from the URLs collected in the previous step, we collect the classes and interfaces names, their explanations, and the URLs to the fields, methods and constructors names and their explanations. In total, we collect 18,738 Java and 11,771 Python terms and their explanations from the API documentation. An example of the terms and corresponding explanations from Java 17 and Python 3.11 API documentation is shown below.

Term and Explanation from Java 17 API Documentation

java.io: Provides for system input and output through data streams, serialization and the file system.

[3] `https://pypi.org/project/beautifulsoup4/`

[4] `https://pypi.org/project/requests/`

java.lang: Provides classes that are fundamental to the design of the Java programming language.

Term and Explanation from Python 3.11 API Documentation
str.splitlines: Return a copy of the string with the leading and trailing characters removed.
int.bit_count: Return the number of ones in the binary representation of the absolute value of the integer. This is also known as the population count.

(c) Glossary

As our third source, we collect 126 Java and 244 Python language-specific terms defined in the glossary. For Java we scrape from the oracle glossary [5] and for python we scrape from the python glossary[6].

Term and Explanation from Java and Python Glossary
Python:
DOM: Document Object Model. A tree of objects with interfaces for traversing the tree and writing an XML version of it, as defined by the W3C specification.
Java:
immutable: An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example, as a key in a dictionary.

After collecting the data from the three data sources, we discard any duplicates based on their terms and explanations. We divide the data into three different subsets — Java, Python and Miscellaneous. The miscellaneous subset consists of terms and explanations from different programming languages. Table 4.2 contains the descriptive statistics of our dataset. We find that the average length of each domain-specific term is approximately 13 characters, and their explanations have an average length of approximately 188 characters.

[5]`https://www.oracle.com/java/technologies/glossary.html`
[6]`https://docs.python.org/3.11/glossary.html`

Table 4.2: Dataset Details

PL	**Source**	**Size**	**ATL (characters)**	**AEL (characters)**	**Complete Size**
Python	Stack Overflow	10,022	10.58	115.27	28,760
	API Documentation	18,738	16.72	360.10	
	Glossary	126	14.31	361.40	
Java	Stack Overflow	9,594	11.88	121.38	21,365
	API Documentation	11,771	15.90	89.61	
	Glossary	244	10.87	154.19	
Micellaneous	Stack Overflow	105,822	11.55	114.02	105,822

PL = Programming Language, **ATL** = Average Term Length,
AEL = Average Explanation Length

4.3.2 Data Cleaning

We use standard natural language pre-processing techniques to clean the domain-specific terms or jargon and their explanations. First, we remove the noisy elements like — HTML tags and URLs. Second, we use the *"pyspellchecker"*, a spell-checking library[7] to correct the spellings of any misspelled words. We then perform lemmatization on all items in our corpus. This step ensures that words are transformed into their root forms, facilitating better analysis [77].

4.3.3 Data Splitting

After we obtain the cleaned data from the previous step, we split each subset of terms and explanations into training, validation and test sets. We split each subset of our dataset (Java, Python, Miscellaneous) into training, validation and testing with the following ratios: 80% training, 10% validation, and 10% testing data.

4.3.4 Fine Tuning the Model

Model Input, Output and Structures: We fine-tune the T5 model from HuggingFace[8] on the T5ForConditionalGeneration variant [106], with our collected data. We use domain-specific terms or jargon as input (a.k.a., source sentence) and their

[7]`https://pypi.org/project/pyspellchecker/`
[8]`https://huggingface.co/t5-base`

explanations as output (a.k.a., target sentence). We train the T5 model with its associated encoder and tokenizer [37]. The model has a 512-dimensional embedding size, a 6-layer encoder, and eight attention heads per layer. The model also has positional embeddings for sequences up to 512 tokens, contributing to its ability to handle diverse input lengths.

Hyperparameter Tuning: Applying grid search for hyper-parameter tuning is not feasible due to the large number of parameters in a T5 model (e.g., 60M to 220M parameters) [113]. We thus perform heuristic-based hyperparameter tuning. We fine-tuned the T5 model through multiple iterations until it reached a stable BLEU score by tuning parameters such as learning rate, maximum sequence length, training batch size, and number of training epochs. We also repeat our training with ten random splits of the Java, Python and Miscellaneous datasets using scikit-learn's library [114] and report the average performance. We set the following parameters for our model training — the train and valid batch sizes are both 8; the learning rate is $1e-4$; the maximum source and target text lengths are 128 and 512 tokens, respectively; and a random seed of 42 for reproducibility. Further details about the hyperparameters can be found in the replication package [110].

Model Optimization and Regularization: In configuring the model architecture, the feed-forward dimension is set to 2048, and dropout with a rate of 0.1 is applied for regularization. We also use the AdamW optimizer [115], a variant of the Adam optimizer that corrects the weight decay regularization. The T5 model is trained on the Colossal Clean Crawled Corpus (C4), a large collection of approximately 750GB of English texts sourced from Common Crawl for text generation tasks. Thus, we did not pre-train it for our task since our dataset consists of natural language English language texts [37].

Hardware Configuration and Training Time: Our experiments are run on one NVidia A100 GPU with 40GB of memory. For the Java Dataset, the average model training time was approximately 30 hours, and the model was trained for 18 epochs. The average model training time for the Python dataset was approximately 30 hours, and the model was trained for 16 epochs. In the case of the Miscellaneous Dataset, a more extended training period is required. The average model training time for this dataset was approximately 120 hours, and the model is trained for 12 epochs.

4.4 Experiment

We evaluate our technique using three datasets containing domain-specific terms or jargon and their corresponding explanations using appropriate metrics from the relevant literature —- BLEU score [29], Semantic Similarity (SS) [31], and METEOR [30]. Our datasets are based on Stack Overflow posts, API documentation, and glossary from two programming languages (Table 4.2). To place our work in the literature, we also compare our technique with two baseline techniques. Through our experiments, we answer three research questions as follows:

(a) $\mathbf{RQ_1}$: How does our technique perform in explaining domain-specific terms or jargon according to the automatic evaluation metrics?

(b) $\mathbf{RQ_2}$: Can our technique outperform the existing baseline techniques in generating explanations to domain-specific terms or jargon?

(c) $\mathbf{RQ_3}$: Does our enrichment of bug reports help improve an existing technique for duplicate bug report detection?

4.4.1 Dataset Construction

To evaluate different aspects of our technique through experiments, we construct two datasets as follows:

(a) Test Vocabulary

To answer $\text{RQ}_\mathbf{1}$ and $\text{RQ}_\mathbf{2}$, we reuse the dataset that we constructed earlier (Section 4.3) and perform splitting to get 10% testing data. Our test dataset contains 2,876 Java, 2,136 Python and 10,582 Miscellaneous terms and their explanations. We call them Java_{TEST}, Python_{TEST}, and $\text{Miscellaneous}_{TEST}$ respectively.

(b) Bug Report Vocabulary

To answer $\text{RQ}_\mathbf{3}$, we collect 92,854 bug reports from an existing benchmark constructed from three open-source systems — Eclipse, Firefox and Mobile [39]. We follow the approach of Jahan et al. [39] and apply standard natural language pre-processing techniques to each bug report. We discard stopwords since it does not capture any semantic meaning. We then split the bug report into tokens and remove noisy elements such as non-alphanumeric characters, numbers, HTML tags, and URLs. Lastly, we convert each bug report into lowercase text.

To obtain the infrequent, domain-specific terms or jargon from a bug report, we apply TF-IDF based scoring to its content. We then collect the top 10 least frequent terms as domain-specific keywords from each bug report for explanation generation.

4.4.2 Generating Explanations and Enriching Bug Report

Using our fine-tuned T5 model, we generate the explanations for each of the domain-specific terms or jargon that are obtained from the previous step as follows.

(a) Test Vocabulary:

We generate an explanation for each term from all three datasets — Java$_{TEST}$, Python$_{TEST}$, and Miscellaneous$_{TEST}$. We thus collect 2,876 Java, 2,136 Python and 10,582 Miscellaneous terms and generated explanations.

(b) Bug Report Vocabulary:

Using our fine-tuned model, we also generate explanations for the top 10 domain-specific terms or jargon from each bug report. The explanations are then injected into relevant places (see Section 4.2) within the texts to construct the enriched bug reports. We repeat this for all three subject systems — Eclipse, Firefox and Mobile. We call these enriched bug reports — Eclipse$_{Enriched}$, Firefox$_{Enriched}$ and Mobile$_{Enriched}$ and use them to answer RQ$_{\mathbf{3}}$.

4.4.3 Evaluation Metrics

To evaluate the explanations from BugEnricher (a.k.a., fine-tuned T5 model) against the ground truth, we use three relevant metrics from literature — BLEU Score [29], METEOR Score [30], and Semantic Similarity metric [31]. They are defined as follows:

BLEU — Bi-Lingual Evaluation of Understudy

BLEU score is a commonly used metric for evaluating translation [29], which has found application in many software engineering tasks (e.g., comment generation [40], text summarization [116]). It compares a candidate text to a reference text and determines how similar they are based on the matching of their n-grams. The BLEU score is calculated as follows:

$$BLEU = BP \cdot exp\left(\sum_{n=1}^{N} w_n log(p_n)\right) \tag{4.1}$$

where N is the maximum n-gram order, w_{n} is the weight assigned to the n-gram order, BP is the brevity penalty — a factor that penalizes the BLEU score when the candidate text is shorter than the reference text, and p_{n} is the modified n-gram precision, which measures the ratio of the overlapping n-grams (between the candidate text and the reference text), and the total number of n-grams in the candidate text.

SS — Semantic Similarity

In a recent work, Haque et al. [31] investigate which metric best reflects human similarity assessment. They suggest that Sentence-BERT [89] provides semantically meaningful sentence embeddings. Thus, when a candidate text is compared with the reference text based on these embeddings using cosine similarity, it has the highest correlation with human-evaluated similarity. The semantic similarity score is computed as follows:

$$SemSim(ref, gen) = \cos(\mathrm{sbert}(ref), \mathrm{sbert}(gen)) \tag{4.2}$$

where $sbert(ref)$, and $sbert(gen)$ are the numerical representations from Sentence-BERT for the reference text and generated text, respectively.

METEOR — Metric for Evaluation of Translation with Explicit ORdering

The METEOR score is a metric for evaluating the quality of machine translation output based on both lexical and syntactic information [30]. It measures the similarity between a candidate text and the reference text by sequentially applying exact match, stemmed match and wordnet-based synonym match between the texts.

4.5 Evaluation of BugEnricher

4.5.1 Answering $\mathbf{RQ_1}$ — How does our technique perform in explaining domain-specific terms or jargon according to automatic evaluation metrics?

In this experiment, we analyze the performance of BugEnricher using three evaluation metrics - BLEU score [29], Semantic Similarity [31] and METEOR score [30]. We evaluate our fine-tuned model using a total of 15,594 domain-specific terms or jargon from three datasets — Java$_{TEST}$, Python$_{TEST}$, and Miscellaneous$_{TEST}$ (Section 4.4.2). We collect explanations from BugEnricher for each of these terms and compare them against the ground truth explanations. Table 4.3 shows the performance details of BugEnricher. It should be noted that a higher value for BLEU, METEOR, and Semantic Similarity is desirable in our experiments.

BugEnricher achieves a maximum BLEU score of 28.85 for Java and 24.63 for Python, which are considered *understandable to good* according to Google's AutoML Translation documentation [33]. This shows that the explanations from our model have a significant overlap with the ground truth in terms of individual words and phrases. However, the BLEU score emphasizes capturing the precision of a response against the ground truth. Thus, we also evaluate our answers using the METEOR score, which takes into account additional information such as synonyms, word forms, and sentence structure when capturing recall [30].

In Table 4.3, we find that our model achieves a maximum METEOR score of 0.27 for Java and 0.23 for Python. This shows that BugEnricher was able to produce a significant part of the ground truth texts in its generated explanations. However, since BLEU and METEOR scores rely on keyword matching between a generated explanation and the ground truth explanation, they may not capture the semantic relevance between them. Hence, we also evaluate our generated explanations using Semantic Similarity (SS) metric. Explanations from BugEnricher achieve a maximum of 53.26% Semantic Similarity (SS) for Java and 48.85% for Python, which indicates a major semantic overlap with the ground truth explanations.

In Table 4.3, we also report the performance of BugEnricher in cross-language settings (a.k.a, BugEnricher$_{Miscellaneous}$) where the domain terms or jargon are related

Table 4.3: Performance of BugEnricher

Model	BLEU	METEOR	SS
BugEnricher$_{Python}$	24.63	0.23	48.85
BugEnricher$_{Java}$	28.85	0.27	53.26
BugEnricher$_{Miscellaneous}$	24.27	0.18	41.57

to multiple programming languages and general software engineering. The goal was to determine the generality of our technique. We collect an explanation for each term from the Miscellaneous$_{TEST}$ dataset that contains 10,522 domain-specific terms or jargon and compare them against the ground truth explanation. We find that BugEnricher achieves a maximum BLEU score of 24.27, METEOR of 0.18 and Semantic Similarity (SS) of 41.57%. This shows that the generated and ground truth explanations are semantically and contextually similar even with this dataset. Despite the performance drop, BugEnricher can offer reasonable explanations across different programming languages, which is promising.

Summary of RQ$_1$: BugEnricher can generate meaningful explanations for domain-specific terms or jargon in Python and Java. The generated explanations are *understandable* to *good* according to Google's Standard, and they achieve a maximum BLEU score of 28.85, METEOR score of 0.27 and Semantic Similarity score of 53.26, which are promising.

4.5.2 Answering RQ$_2$ — Can our technique outperform the existing baseline techniques in generating explanations to domain-specific terms or jargon?

In this experiment, we compare our technique with two existing baselines in terms of evaluation metrics. To the best of our knowledge, there exists no work that automatically generates relevant explanations against domain-specific terms or jargon from a bug report. However, two existing question-answering techniques could be closely relevant to ours. AnswerBot [36] can synthesize answers for Java-related technical, non-factoid questions from StackOverflow. T5 [37] is an encoder-decoder model trained on a large natural language corpus and has found numerous applications,

Table 4.4: Baseline Performances

Model	BLEU	METEOR	SS
Baseline$_{AnswerBot}$	15.23	0.16	32.47
Baseline$_{T5}$	10.62	0.13	27.89
BugEnricher$_{Java}$	28.85	0.27	53.26

including Question Answering. We thus consider these two techniques as our baselines for the comparison. We call them Baseline$_{AnswerBot}$, and Baseline$_{T5}$ respectively. Table 4.4 shows the comparison details between BugEnricher and these baselines.

To implement Baseline$_{AnswerBot}$, we use the replication package provided by the authors [36], [97]. To adapt it to our task, we append "What is" as a prefix to each domain-specific term in the Java$_{TEST}$ dataset (Section 4.4.2) to construct the question. We use the author-provided corpus to capture answers (a.k.a., explanations) from this baseline. We observe that Baseline$_{AnswerBot}$ performs significantly poorly compared to BugEnricher but performs better than Baseline$_{T5}$. For example, Baseline$_{AnswerBot}$ achieves a maximum BLEU score of 15.23, METEOR score of 0.16 and Semantic Similarity of 32.47, which are lower than the corresponding measures of BugEnricher. BugEnricher achieves an overall performance gain [117] of 72.12% across all three metrics compared to Baseline$_{AnswerBot}$.

To implement Baseline$_{T5}$, we use the T5-base model on HuggingFace [106]. We provide the model with the domain-specific terms from the Java$_{TEST}$ (Section 4.4.2) for our experiment. We use the T5ForConditionalGeneration implementation [106] of the baseline and capture explanations for each domain-specific terms or jargon. From Table 4.4, we observe that Baseline$_{T5}$ performs poorly compared to Baseline$_{AnswerBot}$ and BugEnricher in generating explanations. For example, Baseline$_{T5}$ achieves a BLEU score of 10.62, a METEOR score of 0.13 and a Semantic Similarity of 27.89. This shows very limited contextual or semantic overlap between the generated explanation and the ground truth. Overall, BugEnricher achieves a performance gain [117] of 88.34% across all three metrics, compared to Baseline$_{T5}$.

Summary of RQ$_2$: BugEnricher performs better in generating explanations than both baselines in terms of three evaluation metrics. BugEnricher outperforms both baselines Baseline$_{T5}$ and Baseline$_{AnswerBot}$ by 72.12% and 88.34% respectively, across all three metrics.

4.5.3 Case Study: Answering RQ_3 — Does our enrichment of bug reports help improve an existing technique for duplicate bug report detection?

Textually dissimilar duplicate bug reports differ from textually similar duplicate bug reports in terms of their underlying semantics and structures. For instance, textually dissimilar duplicate bug reports often have missing components (e.g., observed behaviours) or components that are written differently, which could lead to their overall textual differences [39].

In this experiment, we examine whether our enrichment of bug reports with meaningful explanations can improve the detection of textually dissimilar but duplicate bug reports [39]. We use a popular IR-based technique — BM25 — to detect duplicate bug reports, as chosen by an existing work [39]. First, we extract the top ten infrequent terms from each bug report using TF-IDF (See Section 2.6). Then, we generate their explanations using BugEnricher and inject these explanations within the bug report. We call the datasets containing enriched bug reports — $\text{Eclipse}_{Enriched}$, $\text{Firefox}_{Enriched}$ and $\text{Mobile}_{Enriched}$ in our experiment. Please check Section 4.4 for further details on the experimental setup. We calculate, the Recall-rate@K of BM25 technique [38] performance metric for k = 1, 5, 10 and 100, as shown in Table 4.5.

We observe that the Recall-rate@K of the IR-based technique improves for the enriched bug reports across the three subject systems — $\text{Eclipse}_{Enriched}$, $\text{Firefox}_{Enriched}$ and $\text{Mobile}_{Enriched}$. Interestingly, it performs better with $\text{Eclipse}_{Enriched}$ compared to $\text{Firefox}_{Enriched}$ and $\text{Mobile}_{Enriched}$. We also conduct the Mann-Whitney Wilcoxon test [98] to check if the performance metrics with $\text{Eclipse}_{Enriched}$, $\text{Firefox}_{Enriched}$ and $\text{Mobile}_{Enriched}$ are significantly higher. We find that the performance improves by a statistically significant margin ($p\text{-value} = 0.0004 < 0.05$). Thus, according to the above evidence, BugEnricher was able to offer complementary information to the bug reports through the explanations of domain-specific terms or jargon.

Table 4.5: Performance of a Duplicate Bug Report Detection technique using Enriched Bug Reports

BM25 for Duplicate Bug Report Detection				
Dataset	**Textually Dissimilar Duplicates (Recall-rate@k)%**			
	k=1	**k=5**	**k=10**	**k=100**
Eclipse	21.83	37.5	41.27	52.58
Eclipse$_{Enriched}$	28.16	40.24	44.94	56.17
Firefox	11.64	20.82	26.56	46.72
Firefox$_{Enriched}$	15.47	21.05	26.71	49.63
Mobile	15.73	28.09	35.96	59.55
Mobile$_{Enriched}$	24.14	28.16	38.22	61.24

Summary of RQ$_3$: BugEnricher is able to offer complementary information to textually dissimilar duplicate bug reports and enrich them through its explanations for domain-specific terms or jargon. These enriched bug reports were also found to improve the performance of an existing duplicate bug report detection technique by a statistically significant margin.

4.6 Related Work

Several existing studies attempt to help newcomers comprehend bug reports by complementing them with additional information. Correa et al. [25] proposed the inclusion of web links in issue tracker discussions that can benefit developers by providing external knowledge sources or artifacts. Zhang et al. [5] suggest enhancing bug reports with ranked sentences extracted from historical bug reports using information retrieval. Dit et al. [26] introduce a technique that recommends relevant comments to help developers establish explicit connections between the recommended comments and existing comments. Moran et al. [118] proposed FUSION, a tool for enhancing Android bug reports with reproduction steps of an Android bug. Mattia et al. [119] proposed EBUG that can provide a real-time understanding of reproduction steps in software bug reports. Xu et al. [36] developed AnswerBot to generate responses for technical, non-factoid questions from StackOverflow. We compare BugEnricher with AnswerBot [36] using experiments. The detailed comparison can be found in Section 4.5.2.

In short, existing techniques provide additional information to complement bug reports through external resources and past relevant bug reports. However, they do not address the challenges novice or newcomer developers face in comprehending bug reports. To the best of our knowledge, our technique is the first to enrich bug reports with domain-specific terms or jargon explanations using neural text generation, which makes our work *novel*. We found that the generated explanations from our technique outperform two existing baselines according to automated metrics. We also found that the enrichment of textually dissimilar bug reports with explanations for domain-specific terms or jargon improved the performance of an existing duplicate bug detection.

4.7 Threats to Validity

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

External Validity: Threats to external validity refer to the lack of generalizability in the findings [102]. One threat could stem from our selection of data sources. We select the API documentation and glossary of two programming languages and the Stack Overflow tags, which might not represent all relevant sources for software-specific terms or jargon. However, the underlying algorithm of BugEnricher is not bound to any programming language and thus can be easily adapted to any other data sources.

Construct Validity: Construct validity refers to the extent to which the experiment measures what it intends to measure [103]. Inappropriate use of evaluation metrics could be a threat to construct validity. However, we chose our evaluation metrics — BLEU, METEOR, and Semantic Similarity — based on relevant literature [29]–[32]. Thus, threats to construct validity might be mitigated.

Internal Validity: Threats to internal validity relate to experimental errors and subjective biases [105]. A source of threat could be the replication of the baseline techniques. For the replication of T5 [37], we collected the pre-trained model from HuggingFace [106]. To replicate AnswerBot [36], we used the replication package from the original authors [97]. Furthermore, we followed the documentation closely for any customizations. Thus, threats to internal validity might be mitigated.

Limitation: Our study contributes to explaining domain-specific terms or jargon that can be valuable to a developer in comprehending the bug report. However, it is essential to acknowledge certain limitations that may impact the generalizability and interpretation of the findings. Our selection of data could also introduce sampling bias. In future, we will consider more sources for domain-terms and jargon.

(a) Generalizability of Data: In our study, we only use three different sources for collecting our data — StackOverflow, Glossary and API documentation. However, they may not capture all domain-specific terms or jargon that could be present on a bug report.Our selection of data could also introduce sampling bias. In future, we will consider more sources for domain-terms and jargon.

(b) Dependency on Pre-trained Models: Our study utilizes pre-trained models such as T5 for generating explanations. The effectiveness of these models is contingent upon the quality and representativeness of the pre-training data. Issues such as biases present in the pre-trained models or their limited understanding of certain domain-specific nuances could impact the accuracy of explanations generated for bug reports.

4.8 Summary

Our previous study (Chapter 3) provides missing information to deficient bug reports by answering their follow-up questions. In this chapter, we propose BugEnricher that generates explanations to software-specific terms by learning from thousands of domain-specific terms and their explanations from Stack Overflow, official API documentation, and an online glossary. Our evaluation using three performance metrics shows that BugEnricher is able to generate understandable to good explanations to the domain-specific terms when compared against the ground truth as per Google's standards. Our technique was also able to outperform two existing baselines across three metrics. Furthermore, we also conduct a case study using duplicate bug reports and attempt to enrich duplicate bug reports that are textually dissimilar. We find that the enrichment of bug reports by BugEnricher improved the performance of an existing technique for duplicate bug detection. Thus, the empirical findings above suggest that our technique has the potential to enrich a bug report significantly, which could lead to improved bug understanding and management.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Missing information in bug reports makes bug resolution challenging for developers [3]. Bug reports often do not contain all the required information for reproducing or resolving the bug [5]. Developers pose follow-up questions to bug reporters seeking missing information. However, bug reporters often fail to provide answers in a timely fashion. Furthermore, complex contextual information in bug reports could make bug understanding challenging. Newcomers or novice developers of a project might thus need additional help to understand or resolve a bug accurately. First, there have been several existing studies that provide complementary information to bug reports using automated techniques [9], [10], [21]–[23]. However, there has been little research investigating the follow-up questions from bug reports or their answers. Second, existing studies offer complementary information to support bug understanding, leveraging external resources and past relevant bug reports [5], [25], [26], [36], [118], [119]. However, they do not focus on the domain-specific terms or jargon, which warrants for further investigation. This thesis addresses the issue of missing information by complementing deficient bug reports with additional information. Thus, we conduct two separate but complementary studies (Chapter 3) and Chapter 4), and we have the following outcomes.

- The first study (Chapter 3) proposes a novel technique — BugMentor — that can offer relevant answers to follow-up questions from bug reports by combining structured information retrieval and neural text generation. We evaluate our technique on top 20 (5 Java, 5 Python, 5 C++ and 5 JavaScript) GitHub projects and four evaluation metrics (i.e., BLEU, Semantic Similarity, WMD and METEOR). Our evaluation using four performance metrics shows that

BugMentor can generate understandable and good answers to follow-up questions, as per Google's Standard. Our technique was also able to outperform three existing baselines across all four metrics. We also evaluate BugMentor using a user study using 10 developers. The developers found the answers from BugMentor to be more accurate, precise, concise and useful compared to the baseline answers. Thus, BugMentor has the potential to support bug resolution with complementary information in the form of answers to follow-up questions.

- The second study (Chapter 4) proposes a novel technique — BugEnricher — that generates explanations to software-specific terms by learning from thousands of domain-specific terms and their explanations from Stack Overflow, official API documentation, and an online glossary. We evaluate our technique on Python, Java, and Miscellaneous (a.k.a., cross-language) and three evaluation metrics (i.e., BLEU, Semantic Similarity, and METEOR). BugEnricher is able to generate understandable to good explanations to the domain-specific terms when compared against the ground truth as per Google's standards. Our technique was also able to outperform two existing baselines across three metrics. Furthermore, we also conduct a case study using duplicate bug reports and attempt to enrich duplicate bug reports that are textually dissimilar. We find that the enrichment of bug reports by BugEnricher improved the performance of an existing technique for duplicate bug detection.

5.2 Future Work

We have several directions for future research from both our studies. We present the potential future work for each study below.

5.2.1 BugMentor

There are several avenues for future work from BugMentor. First, we plan to design a tool that can be integrated with real-world platforms like GitHub or JIRA to assist the bug reporters and the developers in their work. In particular, real-time feedback from the stakeholders can be leveraged to improve our retrieval algorithm. Second, BugMentor does not use a fine-tuned version of CodeT5. If the labelled dataset (Refer to Section 3.4.1 for details) can be extended, it can be used to fine-tune the CodeT5 model, which could lead to better answers.

5.2.2 BugEnricher

In the future, there are numerous potential directions for BugEnircher. First, we plan to investigate how image and video data attached to the bug report can provide additional context to the bug report. Second, we plan on fine-tuning BugEnricher with vocabulary from popular libraries from both Python and Java and improving the explanations further to generate code examples and investigate if these enriched bug reports can improve existing automated bug localization techniques.

Bibliography

[1] IEEE, "Ieee standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990. DOI: `10.1109/IEEESTD.1990.101064`.

[2] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software-quantify the time and cost saved using reversible debuggers," *University Cambridge: Cambridge, UK*, 2013.

[3] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu, "How practitioners perceive automated bug report management techniques," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 836–862, 2018.

[4] 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.

[5] T. Zhang, J. Chen, H. Jiang, X. Luo, and X. Xia, "Bug report enrichment with application of automated fixer recommendation," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, IEEE, 2017, pp. 230–240.

[6] M. M. Rahman, F. Khomh, and M. Castelluccio, "Why are some bugs non-reproducible?:–an empirical investigation using data fusion–," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2020, pp. 605–616.

[7] O. Chaparro *et al.*, "Detecting missing information in bug descriptions," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 396–407.

[8] *Github issue template documentation.* [Online]. Available: `https://shorturl.at/jsJU2`.

[9] M. M. Imran, A. Ciborowska, and K. Damevski, "Automatically selecting follow-up questions for deficient bug reports," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, IEEE, 2021, pp. 167–178.

[10] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information needs in bug reports: Improving cooperation between developers and users," in *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, 2010, pp. 301–310.

[11] R. Velly Lotufo, "Towards next generation bug tracking systems," 2013.

[12] M. Guizani *et al.*, "The long road ahead: Ongoing challenges in contributing to large oss organizations and what to do," *Proceedings of the ACM on Human-Computer Interaction*, vol. 5, no. CSCW2, pp. 1–30, 2021.

[13] Y. Zhao, T. He, and Z. Chen, "A unified framework for bug report assignment," *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 04, pp. 607–628, 2019.

[14] E. Bodden, W. Sch, A. van Deursen, A. Zisman, *et al.*, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering: ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017.* Association for Computing Machinery (ACM), 2017.

[15] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *2012 19th Working Conference on Reverse Engineering*, IEEE, 2012, pp. 205–214.

[16] M. Nayrolles and A. Hamou-Lhadj, "Towards a classification of bugs to facilitate software maintainability tasks," in *Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies*, 2018, pp. 25–32.

[17] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 70–79.

[18] O. Chaparro, J. M. Florez, U. Singh, and A. Marcus, “Reformulating queries for duplicate bug report detection,” in *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*, IEEE, 2019, pp. 218–229.

[19] W. Zhang, Z. Li, Q. Wang, and J. Li, “Finelocator: A novel approach to method-level fine-grained bug localization by query expansion,” *Information and Software Technology*, vol. 110, pp. 121–135, 2019.

[20] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi, “Improving bug localization with word embedding and enhanced convolutional neural networks,” *Information and Software Technology*, vol. 105, pp. 17–29, 2019.

[21] Y. Tian, F. Thung, A. Sharma, and D. Lo, “Apibot: Question answering bot for api documentation,” in *2017 32nd IEEE/ACM international conference on automated software engineering (ASE)*, IEEE, 2017, pp. 153–158.

[22] A. Bansal, Z. Eberhart, L. Wu, and C. McMillan, “A neural question answering system for basic questions about subroutines,” in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2021, pp. 60–71.

[23] J. Lu, X. Sun, B. Li, L. Bo, and T. Zhang, “Beat: Considering question types for bug question answering via templates,” *Knowledge-Based Systems*, vol. 225, p. 107 098, 2021.

[24] X. Tan, M. Zhou, and Z. Sun, “A first look at good first issues on github,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 398–409.

[25] D. Correa, S. Lal, A. Saini, and A. Sureka, “Samekana: A browser extension for including relevant web links in issue tracking system discussion forum,” in *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, vol. 1, 2013, pp. 25–33.

[26] B. Dit and A. Marcus, “Improving the readability of defect reports,” in *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, 2008, pp. 47–49.

[27] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, “Improving bug localization using structured information retrieval,” in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2013, pp. 345–355.

[28] V. Efstathiou, C. Chatzilenas, and D. Spinellis, “Word embeddings for the software engineering domain,” in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 38–41.

[29] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: A method for automatic evaluation of machine translation,” in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[30] S. Banerjee and A. Lavie, “Meteor: An automatic metric for mt evaluation with improved correlation with human judgments,” in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.

[31] S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, “Semantic similarity metrics for evaluating source code summarization,” in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 36–47.

[32] G. Huang, C. Guo, M. J. Kusner, Y. Sun, F. Sha, and K. Q. Weinberger, “Supervised word mover’s distance,” *Advances in neural information processing systems*, vol. 29, 2016.

[33] *Google automl documentation.* [Online]. Available: `https://cloud.google.com/translate/automl/docs/evaluate`.

[34] M. McCandless, E. Hatcher, O. Gospodnetić, and O. Gospodnetić, *Lucene in action*. Manning Greenwich, 2010, vol. 2.

[35] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” *arXiv preprint arXiv:2109.00859*, 2021.

[36] B. Xu, Z. Xing, X. Xia, and D. Lo, “Answerbot: Automated generation of answer summary to developers’ technical questions,” in *2017 32nd IEEE/ACM international conference on automated software engineering (ASE)*, IEEE, 2017, pp. 706–716.

[37] C. Raffel *et al.*, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[38] C.-Z. Yang, H.-H. Du, S.-S. Wu, and X. Chen, “Duplication detection for software bug reports based on bm25 term weighting,” in *2012 Conference on Technologies and Applications of Artificial Intelligence*, IEEE, 2012, pp. 33–38.

[39] S. Jahan and M. M. Rahman, “Towards understanding the impacts of textual dissimilarity on duplicate bug report detection,” in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2023, pp. 25–36.

[40] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, “Deep code comment generation,” in *Proceedings of the 26th conference on program comprehension*, 2018, pp. 200–210.

[41] P. Mahbub, *Comprehending software bugs leveraging code structures with neural language models*, 2023.

[42] Y. Zhang and Z. Teng, *Natural language processing: a machine learning perspective*. Cambridge University Press, 2021.

[43] Y. Wu *et al.*, “Google’s neural machine translation system: Bridging the gap between human and machine translation,” *arXiv preprint arXiv:1609.08144*, 2016.

[44] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, “Coconut: Combining context-aware neural translation models using ensemble for program repair,” in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.

[45] N. Jiang, T. Lutellier, and L. Tan, “Cure: Code-aware neural machine translation for automatic program repair,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 1161–1173.

[46] S. Jiang, “Boosting neural commit message generation with code semantic analysis,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2019, pp. 1280–1282.

[47] A. LeClair, S. Jiang, and C. McMillan, “A neural model for generating natural language summaries of program subroutines,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 795–806.

[48] A. Vaswani *et al.*, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.

[49] R. Moser, W. Pedrycz, and G. Succi, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” in *Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 181–190.

[50] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, “When deep learning met code search,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference*, 2019, pp. 964–974.

[51] P. Mahbub, O. Shuvo, and M. M. Rahman, “Explaining software bugs leveraging code structures in neural machine translation,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, 2023, pp. 640–652.

[52] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” *Advances in neural information processing systems*, vol. 26, 2013.

[53] 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.

[54] S. N. Kim and L. Cavedon, “Classifying domain-specific terms using a dictionary,” in *Proceedings of the Australasian Language Technology Association Workshop 2011*, 2011, pp. 57–65.

[55] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[56] B. Stecanella, *Understanding TF-ID: A simple introduction.* 2019. [Online]. Available: `https://monkeylearn.com/blog/what-is-tf-idf/`.

[57] D. Ravichandran and E. Hovy, “Learning surface text patterns for a question answering system,” in *Proceedings of the 40th Annual meeting of the association for Computational Linguistics*, 2002, pp. 41–47.

[58] A. Abdellatif, K. Badran, and E. Shihab, “Msrbot: Using bots to answer questions from software repositories,” *Empirical Software Engineering*, vol. 25, pp. 1834–1863, 2020.

[59] E. Brill, S. Dumais, and M. Banko, “An analysis of the askmsr question-answering system,” in *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing (EMNLP 2002)*, 2002, pp. 257–264.

[60] D. L. Waltz, “An english language question answering system for a large relational database,” *Communications of the ACM*, vol. 21, no. 7, pp. 526–539, 1978.

[61] M. Iyyer, J. Boyd-Graber, L. Claudino, R. Socher, and H. Daumé III, “A neural network for factoid question answering over paragraphs,” in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 633–644.

[62] M. Asaduzzaman, A. S. Mashiyat, C. K. Roy, and K. A. Schneider, “Answering questions about unanswered questions of stack overflow,” in *2013 10th Working Conference on Mining Software Repositories (MSR)*, IEEE, 2013, pp. 97–100.

[63] T. Yu, X. Gu, and B. Shen, “Code question answering via task-adaptive sequence-to-sequence pre-training,” in *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2022, pp. 229–238.

[64] *Replication package.* [Online]. Available: `https://git.cs.dal.ca/umukherjee/bugmentor`.

[65] Tensorflow, *Tensorflow/tensorflow issue 58280*. [Online]. Available: `https://github.com/tensorflow/tensorflow/issues/58280`.

[66] [Online]. Available: `https://github.com/tensorflow/tensorflow/blob/master/ISSUE_TEMPLATE.md`.

[67] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now? an empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, 2006, pp. 25–33.

[68] M. Dickens, *Adding linker flags in xcode*. [Online]. Available: `https://stackoverflow.com/questions/2125079/adding-linker-flags-in-xcode`.

[69] *Github*. [Online]. Available: `https://github.com/`.

[70] [Online]. Available: `https://github.com/search/advanced`.

[71] *Github rest api documentation*. [Online]. Available: `https://docs.github.com/en/rest?apiVersion=2022-11-28`.

[72] *Koshuke github api documentation*. [Online]. Available: `https://github-api.kohsuke.org/`.

[73] *Nltk documentation*. [Online]. Available: `https://www.nltk.org/_modules/nltk/classify`.

[74] Q. Yuan, G. Cong, and N. M. Thalmann, "Enhancing naive bayes with various smoothing methods for short text classification," in *Proceedings of the 21st international conference on world wide web*, 2012, pp. 645–646.

[75] J. S. Whissell and C. L. Clarke, "Improving document clustering using okapi bm25 feature weighting," *Information retrieval*, vol. 14, pp. 466–487, 2011.

[76] [Online]. Available: `https://www.nltk.org/`.

[77] R. Pramana, J. J. Subroto, A. A. S. Gunawan, *et al.*, "Systematic literature review of stemming and lemmatization performance for sentence similarity," in *2022 IEEE 7th International Conference on Information Technology and Digital Applications (ICITDA)*, IEEE, 2022, pp. 1–6.

[78] [Online]. Available: `https://www.elastic.co/elasticsearch/`.

[79] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 842–851.

[80] L. Moreno *et al.*, "Query-based configuration of text retrieval solutions for software engineering tasks," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 567–578.

[81] C. Kamphuis, A. P. de Vries, L. Boytsov, and J. Lin, "Which bm25 do you mean? a large-scale reproducibility study of scoring variants," in *European Conference on Information Retrieval*, Springer, 2020, pp. 28–34.

[82] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais, "The vocabulary problem in human-system communication," *Communications of the ACM*, vol. 30, no. 11, pp. 964–971, 1987.

[83] K. W. Church, "Word2vec," *Natural Language Engineering*, vol. 23, no. 1, pp. 155–162, 2017.

[84] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowdsourced knowledge," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, vol. 1, 2016, pp. 349–359.

[85] H. Face. [Online]. Available: `https://huggingface.co/docs/transformers/tasks/question_answering`.

[86] Y. Seonwoo, J.-H. Kim, J.-W. Ha, and A. Oh, "Context-aware answer extraction in question answering," *arXiv preprint arXiv:2011.02687*, 2020.

[87] M. Kuhrmann, D. M. Fernández, and M. Daneva, "On the pragmatic design of literature studies in software engineering: An experience-based guideline," *Empirical software engineering*, vol. 22, pp. 2852–2891, 2017.

[88] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, vol. 25, pp. 2179–2217, 2020.

[89] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," *arXiv preprint arXiv:1908.10084*, 2019.

[90] A. Agarwal and A. Lavie, "Meteor, m-bleu and m-ter: Evaluation metrics for high-correlation with human rankings of machine translation output," in *Proceedings of the Third Workshop on Statistical Machine Translation*, 2008, pp. 115–118.

[91] M. Kusner, Y. Sun, N. Kolkin, and K. Weinberger, "From word embeddings to document distances," in *International conference on machine learning*, PMLR, 2015, pp. 957–966.

[92] R. Sato, M. Yamada, and H. Kashima, "Re-evaluating word mover's distance," in *International Conference on Machine Learning*, PMLR, 2022, pp. 19 231–19 249.

[93] X. Yao, B. Van Durme, C. Callison-Burch, and P. Clark, "Answer extraction as sequence tagging with tree edit distance," in *Proceedings of the 2013 conference of the North American chapter of the association for computational linguistics: human language technologies*, 2013, pp. 858–867.

[94] A. Severyn and A. Moschitti, "Learning to rank short text pairs with convolutional deep neural networks," in *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*, 2015, pp. 373–382.

[95] J. Atwood, *Sql 2008 full-text search problems*, Oct. 2017. [Online]. Available: `https://stackoverflow.blog/2008/11/01/sql-2008-full-text-search-problems/`.

[96] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Leveraging crowd knowledge for software comprehension and development," in *2013 17th European Conference on Software Maintenance and Reengineering*, IEEE, 2013, pp. 57–66.

[97] *Maxxbw54/answerbot: Replication package of the paper "answerbot: An answer summary generation tool based on stack overflow"*. [Online]. Available: `https://github.com/maxxbw54/AnswerBot`.

[98] J. Cuzick, "A wilcoxon-type test for trend," *Statistics in medicine*, vol. 4, no. 1, pp. 87–90, 1985.

[99] E. W. Weisstein, "Bonferroni correction," *https://mathworld. wolfram. com/*, 2004.

[100] A. Murgia, D. Janssens, S. Demeyer, and B. Vasilescu, "Among the machines: Human-bot interaction on social q&a websites," in *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, 2016, pp. 1272–1279.

[101] Y. Song *et al.*, "Toward interactive bug reporting for (android app) end-users," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 344–356.

[102] L. Ferguson, "External validity, generalizability, and knowledge utilization," *Journal of Nursing Scholarship*, vol. 36, no. 1, pp. 16–22, 2004.

[103] G. T. Smith, "On construct validity: Issues of method and measurement.," *Psychological assessment*, vol. 17, no. 4, p. 396, 2005.

[104] A. Joshi, S. Kale, S. Chandel, and D. K. Pal, "Likert scale: Explored and explained," *British journal of applied science & technology*, vol. 7, no. 4, pp. 396–403, 2015.

[105] T. J. Christ, "Experimental control and threats to internal validity of concurrent and nonconcurrent multiple baseline designs," *Psychology in the Schools*, vol. 44, no. 5, pp. 451–459, 2007.

[106] *Mariorossi/t5-base-finetuned-question-answering.* [Online]. Available: `https://huggingface.co/MaRiOrOsSi/t5-base-finetuned-question-answering`.

[107] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[108] S. Davies and M. Roper, "What's in a bug report?" In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2014, pp. 1–10.

[109] S. Bugde, N. Nagappan, S. Rajamani, and G. Ramalingam, "Global software servicing: Observational experiences at microsoft," in *2008 IEEE International Conference on Global Software Engineering*, IEEE, 2008, pp. 182–191.

[110] *Replication package.* [Online]. Available: `https://git.cs.dal.ca/umukherjee/bugenricher`.

[111] *Python 3.11.6 documentation.* [Online]. Available: `https://docs.python.org/3.11/`.

[112] *Java 17 documentation.* [Online]. Available: `https://docs.oracle.com/en/java/javase/17/docs/api/index.html`.

[113] H. Palivela, "Optimization of paraphrase generation and identification using language models in natural language processing," *International Journal of Information Management Data Insights*, vol. 1, no. 2, p. 100 025, 2021.

[114] [Online]. Available: `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html`.

[115] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[116] E. Shi *et al.*, "On the evaluation of neural code summarization," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1597–1608.

[117] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, "Predicting defective lines using a model-agnostic technique," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1480–1496, 2020.

[118] K. Moran, "Enhancing bug reports for mobile apps," *arXiv preprint arXiv:1801.05932*, 2018.

[119] M. Fazzini, K. Moran, C. Bernal-Cardenas, T. Wendland, A. Orso, and D. Poshyvanyk, "Enhancing mobile app bug reporting via real-time understanding of reproduction steps," *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1246–1272, 2022.

Appendix A

Complementary Materials

A.1 Replication Package

A.1.1 BugMentor

GitLab Repository: https://git.cs.dal.ca/umukherjee/bugmentor

A.1.2 BugEnricher

GitLab Repository: https://git.cs.dal.ca/umukherjee/bugenricher