COMPREHENDING SOFTWARE BUGS LEVERAGING CODE STRUCTURES WITH NEURAL LANGUAGE MODELS

by

Parvez Mahbub

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

at

Dalhousie University
Halifax, Nova Scotia
August 2023

Dalhousie University is located in Mi'kma'ki,
the ancestral and unceded territory of the Mi'kmaq.
We are all Treaty people.

Fariha Islam Mouri.

Thank you for tolerating me when I was intolerable, loving me when I was unlovable, and understanding me when I was incomprehensible.

Contents

List of Tables

List of Figures

Abstract

Software bugs claim $\approx$ 50% of development time and cost the global economy billions of dollars every year. Unfortunately, despite the use of many software quality assurance (SQA) practices in software development (e.g., code review, continuous integration), defects may still exist in the official release of a software product. If software defects can be predicted at the line level, that can help the developers prioritize SQA efforts for the vulnerable areas of a codebase and thus achieve a high-quality software release. However, a defect prediction technique could be less helpful without any meaningful explanation of the defect. In this thesis, we propose and evaluate two novel techniques that support developers in identifying software defects at the line level and provide natural language explanations for those defects. In our first study, we propose – Bugsplorer – a novel deep-learning technique for line-level defect prediction. It leverages a hierarchical structure of transformer models to represent two types of code elements: code tokens and code lines. Our evaluation with five performance metrics shows that Bugsplorer can predict defective lines with 26-72% better accuracy than that of the state-of-the-art technique. It can also rank the first 20% defective lines within the top 1-3% vulnerable lines. In our second study, we propose Bugsplainer – a transformer-based generative model that generates natural language explanations for software bugs by leveraging structural information and buggy patterns from the source code. Our evaluation using three performance metrics shows that Bugsplainer can generate *understandable* and *good* explanations according to Google's standard and can outperform multiple baselines from the literature. We also conducted a developer study involving 20 participants where the explanations from Bugsplainer were found to be more accurate, more precise, more concise and more useful than the baselines. Given the empirical evidence, our techniques have the potential to significantly reduce the SQA costs.

List of Abbreviations Used

APE	Absolute Positional Embedding.
AST	Abstract Syntax Tree.
AuROC	Area under the Receiver Operating Characteristic.
BLEU	Bi-lingual Evaluation of Understudy.
BPE	Byte-Pair Encoding.
BPTT	Backpropagation Through Time.
CFG	Control Flow Graph.
CNN	Convolutional Neural Network.
DBN	Deep Belief Network.
DFG	Data Flow Graph.
DL	Deep Learning.
FN	False Negative.
FP	False Positive.
FPR	False-Positive Rate.
GRU	Gated Recurrent Unit.
IFA	Initial False Alarm.
IR	Information Retrieval.
ITS	Issue Tracking System.

LIME Local Interpretable Model-agnostic Explanations.

LSTM Long Short-term Memory.

ML Machine Learning.

NLM Neural Language Modelling.

NLP Natural Language Processing.

NMT Neural Machine Translation.

PDG Program Dependency Graph.

PR Pull Request.

RNN Recurrent Neural Network.

ROC Receiver Operating Characteristic.

RPE Relative Positional Embedding.

SBT Structure-Based Traversal.

SQA Software Quality Assurance.

T5 Text-To-Text Transfer Transformer.

TN True Negative.

TNR True-Negative Rate.

TP True Positive.

TPR True-Positive Rate.

Acknowledgements

I would like to express my deepest gratitude and appreciation to the following individuals and groups who have played a pivotal role in shaping my academic journey at Dalhousie University.

First and foremost, I am deeply grateful to my thesis supervisor, Dr. Masud Rahman. His profound wisdom, expertise, and dedicated mentorship have been invaluable throughout this journey. His insightful feedback and constructive criticism have helped me navigate through the complexities of my research.

I would also like to express my sincere gratitude to Dr. Tushar Sharma and Dr. Srinivas Sampalli for their invaluable advisement and meticulous evaluation of my thesis. Their scholarly perspectives and thoughtful input have elevated the quality and rigor of my research, and I am truly appreciative of their time and expertise.

To my beloved family members, including my sister Farzana Tasnim Oishee, my mother Mst. Khaleda Begum, my brother Arman Mahmud, and my father Md. Jalilur Rahman, I extend my deepest affection. Their unwavering support, love, and belief in my abilities have been the bedrock of my accomplishments, and I am profoundly grateful for their presence in my life.

I would like to acknowledge the irreplaceable support and affinity of my precious friends Naz Zarreen Oishie, Fariha Islam Mouri, and Sadia Mahjabin – *You have loved me beyond my utmost imagination and been there for me in my annihilating angst.* I would also like to thank Ohiduzzaman Shuvo, Sigma Jahan, Mahfujul Alam Antu, Abu Sayed, Alamin Hossain, and Komol Hasan. Their friendship, encouragement, and shared experiences have added vibrant colors to my academic journey, and I am fortunate to have them by my side.

The collaborative efforts of the talented members of RAISELab, including Usmi Mukherjee, Mehil B Shah, Asif Samir, and many others, deserve special recognition. Their insightful discussions, intellectual contributions, and collective enthusiasm have significantly enriched the development of my research. I extend my sincere gratitude to each one of them.

Furthermore, I would like to express my appreciation to Dalhousie University and the Department of Computer Science for fostering an inspiring academic environment and providing the necessary resources for my research. Their commitment to excellence and dedication to fostering intellectual growth have played a crucial role in shaping the outcome of this thesis.

I would like to thank Metabob, our industry partner, for their generous supports in terms of scholarships, computing resources and mentorship. During my internship there, I have worked with quite a few passionate, joyous, and hardworking colleagues and mentors including Ben Reaves, Avinash Gopal, Axel Lönnfors, and Massimiliano Genta. There constant suggestions and counsel helped me take the quality of my thesis to the next level.

Lastly, I extend my heartfelt thanks to all those who have supported me in various ways throughout this research endeavor. 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

A software bug (a.k.a. software defect) is an incorrect step, process, or data definition in a computer program that prevents the program from producing the correct result [1]. Resolving software bugs has been one of the major tasks of software development and maintenance [2]. According to several studies, it claims $\approx 50\%$ of the development time [3], consumes up to 40% of the total budget [4] and costs the global economy billions of dollars each year [5], [6]. Software Quality Assurance (SQA) practices (e.g., code review, continuous integration) play a major role in preventing these defects. However, despite using many SQA practices in the development phase, software defects may still exist in the official release of a software product [7], [8]. A recent study [9] suggests that only $\approx 3\%$ lines of code from the whole release could lead to most of the defects. Hence, prioritizing SQA efforts for the vulnerable areas of the code is essential to ensure software quality. Nonetheless, even with a precise defect prediction technique, it is the task of the developers to find the root cause of the bugs in the source code and fix them. Reportedly, developers spend $\approx 50\%$ of their time comprehending software code during maintenance [10]. Thus, automated identification of the vulnerable areas of the code with meaningful explanations of potential defects could greatly benefit the developers in their tasks.

Defect prediction has been a popular research topic [11]–[14] that predicts defects in software code. It helps prioritize the SQA efforts to highly suspicious areas of the code [9], which could be helpful to improve the quality of a software product before its release. Defects can be predicted at various abstraction levels of code such as module [15], [16], file [17], [18], method [19], and line [9], [20], [21], where line-level defect prediction provides the most fine-grained defect location. According to a recent developer survey [9], line-level defect prediction would be more helpful for identifying defective code than other abstraction levels (e.g., file-level). However, existing techniques for line-level defect prediction [9], [20], [21] might fail to capture the local context of a software defect and could be limited by the noise (e.g., repetitive keywords, punctuation) in the source code document. As a result, these defect prediction techniques might not be cost-effective when used in the real world [22].

While defect prediction techniques identify specific parts of the code as buggy, they could only be more helpful with a meaningful explanation [23]. Developers are thus generally responsible for understanding a bug from the identified code before making any changes. Understanding bugs by looking at the code claims significant debugging time. Unfortunately, neither many studies attempt to explain the bugs in the source code to the developers, nor are they practical and scalable enough for industry-wide use [23], [24]. Therefore, to prioritize or reduce SQA efforts, developers need precise defect prediction (e.g., line-level) with meaningful, natural language explanations about the defects.

1.2 Problem Statement

The majority of the works on defect prediction use Machine Learning (ML) or Deep Learning (DL) based techniques [9], [20], [21], [25]–[29]. Recent approaches for line-level defect prediction first train their ML models to predict the defective source code documents [9], [20] or commits [21]. Then, if a file or commit is predicted as defective, they identify the tokens in the file, which help explain the defects using various techniques (e.g., attention mechanism [30]). Finally, they mark the code lines from the source file as defective, containing many defect-explaining tokens. However, such an approach poses two major challenges, as follows.

Existing models might not optimally represent code elements: In natural language texts, the semantics of a word is often determined by its context (e.g., surrounding words) [31]. Similarly, the surrounding tokens from both sides could influence the meaning and intent of a code token. For example, Fig. 1.1 shows a piece of defective code, where a code token – `name_str` – contains an erroneous value after the program execution. That is, inside the for loop, the variable `name` should be concatenated (i.e., `+=` operator) to `name_str` instead of being assigned (i.e., `=` operator). Therefore, the code token `name_str` is triggered to be buggy by another code

```
name_str = ""
for name in names:
    name_str = name
name_str = sanitize(name_str)
```

Figure 1.1: An example of defective code

token "=" on its right side. The intent of the token `name_str` is also influenced by the tokens on its left side (e.g., `for`). Such a phenomenon indicates that we need information on the surrounding tokens from both sides to represent a token optimally. However, the state-of-the-art defect prediction technique [9] can only focus on a single direction (a.k.a. unidirectional), which could be either tokens on the left or on the right. Then, it concatenates two unidirectional representations of a token's context to generate a bidirectional representation. Such a representation could be verbose as it contains two different representations for the same set of tokens and be inefficient since it trains two different DL models independently. Furthermore, Reimers *et al.* [32] suggest that simple concatenation of two vectors might not produce an optimal representation for an input (e.g., a token or line).

Existing models might fail to capture the local context of a defect: During the training phase of the existing models [9], [20], [21], the attention values [33] or importance values [34] for the tokens are optimized for file-level defect prediction. In other words, these values are optimized to predict whether the whole file is defective or not. However, source code documents are often quite large, containing thousands of tokens, which could make them noisy. Therefore, the attention or importance values from existing models might fail to adequately capture the local context of a software defect since, in a codebase, only 0.03% – 2.9% lines could be defective [9]. Thus, relying on these attention or importance values might not be sufficient to detect line-level defects accurately.

Once software defects are identified through defect prediction, they need to be explained to the developers, which is a highly challenging task. Many static analysis tools such as FindBugs [35], PMD [36], SonarLint [37], PyLint [38], and pyflakes [39] employ complex hand-crafted rules to detect the defects and vulnerabilities in source code. Then, they use pre-defined message templates to explain the identified defects and vulnerabilities. Unfortunately, their utility could be limited due to their high false-positive results and the lack of actionable insights in their explanations [40]–[43]. In particular, their explanations are often too generic and unaware of the context due to their pre-defined, templated nature [44]. Thung *et al.* [45] also suggest that static analysis tools suffer from many *false negative* results, which could leave the software systems vulnerable to defects.

Unlike traditional, rule-based approaches (e.g., static analysis tools), explaining software defects can be viewed as a translation task, where the defective code is the source language and the corresponding explanation is the target language. In recent years, machine translation, especially Neural Machine Translation (NMT) [31], has found numerous applications in several software engineering tasks including, but not limited to, code summarization [46]–[48], code comment generation [49]–[52], commit message generation [53]–[56], and automatic program repair [2], [57]–[59]. However, explanation generation from the defective source code using NMT poses two major challenges.

Understanding the structures of source code: Natural language is loosely structured, which exhibits phenomena like ambiguity and word movement [31]. Word movement is the appearance of words in a sentence in different orders but still being grammatically correct. On the contrary, programming languages are more structured, syntactically restricted, and less ambiguous [60]. From the two programs having the same vocabulary, one could be buggy, and the other could be correct due to their structural differences (e.g., Fig. 4.3b, 4.3c). Thus, capturing and understanding the code structure is essential to explaining the defective code. Unfortunately, traditional NMT-based techniques often treat source code as a sequence of tokens and thus might fail to capture the structures of source code [61].

Understanding and detecting defective code patterns: From a high-level perspective, NMT models translate words from the source language into words from the target language. However, to generate explanations from the defective code, the model must accurately reason about the bug from the defective code and its structures. Such reasoning is non-trivial and warrants the model to be aware of defective code patterns. Traditional NMT models might not be sufficient to tackle all these challenges due to their simplified assumptions about sequential inputs and outputs. According to Ray *et al.* [62], defective code is less repetitive than regular code, which could exacerbate the above challenges.

As discussed above, the existing approaches for line-level defect prediction [9], [20], [63] and explanation generation [35]–[39] might fall short. Thus, developers are in dire need of tools and techniques that can accurately identify software defects accompanied by meaningful explanations.

1.3 Our Contribution

In this thesis, we propose and evaluate two novel techniques that support developers in identifying software defects at the line level and provide natural language explanations for those defects. We analyze the structural aspect of source code and leverage the contextual information from the code to predict the defective lines and generate explanations for them.

In the first study, we propose – *Bugsplorer* – a novel deep-learning technique for line-level defect prediction. It leverages a hierarchical structure of transformer models to estimate the attention values for two levels of code elements: code tokens and code lines. Bugsplorer can address the previously discussed challenges posed to line-level defect prediction (see Section 1.2), which makes our work *novel.* First, unlike existing techniques [9], [20], [21], Bugsplorer is directly trained for line-level defect prediction and thus can better capture the local context of a defect. Second, unlike sequential models used in several existing studies, Bugsplorer can learn the representation of a code element by simultaneously capturing its context from both the left and the right sides. Thus, our approach is better suited to predict line-level defects.

We train and evaluate Bugsplorer with two different benchmark datasets. The first dataset [64] consists of $\approx$ 230K Python source code document from 24 GitHub repositories. The second dataset [20] consists of 32 software releases that span nine open-source Java software systems. We find that Bugsplorer can predict defective code lines with 26-68% higher accuracy than the state-of-the-art technique [9]. It can also reduce the effort in finding defective lines by 72-81%. We further show that

(a) optimizing deep learning models for line-level defect prediction and (b) generating bidirectional representations of code elements (e.g., tokens and lines) can significantly influence the performance of our technique.

In the second study, we propose – *Bugsplainer* – a novel transformer-based generative model that generates natural language explanations for software bugs by learning from a large corpus of bug-fix commits (i.e., commits that correct bugs). Bugsplainer can address the challenges posed to explanation generation (see Section 1.2), which makes our work *novel*. First, Bugsplainer can leverage code structures in explanation generation by applying structure-based traversal [47] to the defective code. Second, we train Bugsplainer using both defective source code and its corrected version, which helps the model understand and detect defective code patterns during its explanation generation.

We train Bugsplainer with $\approx$ 150K bug-fix commits collected from GitHub and evaluate using three different metrics – BLEU [65], Semantic Similarity [66] and Exact Match. We find that the explanations from Bugsplainer are *understandable* and *good* according to Google's AutoML Translation Documentation[1]. We compare our technique with four appropriate baselines – pyflakes [39], CommitGen [53], NNGen [56], and Fine-tuned CodeT5 [67]. Bugsplainer outperforms all four baselines in all metrics by a statistically significant margin. One major strength of Bugsplainer is understanding the structure of the code and buggy code patterns, where the baselines might be falling short. To further evaluate our work, we conducted a developer study involving 20 developers from six countries, where the identities of both our tool and the baselines were kept hidden. The study result shows that explanations from Bugsplainer are more accurate, more precise, more concise, and more useful compared to that of the baselines.

1.4 Related Publications

Several parts of this thesis have been accepted and published at different conferences. We provide the list of publications here. In each of these papers, I am the primary author, and I conduct all the studies under the supervision of Dr. Masud Rahman. While I wrote these papers, the co-authors took part in advising, editing,

[1]https://bit.ly/3wGpCIx

and reviewing the papers.

- *Parvez Mahbub*, Ohiduzzaman Shuvo, and M. Masudur Rahman. *Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation.* In Proceeding of The 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023), pp. 640-652, Melbourne, Australia, May 2023.

- *Parvez Mahbub*, Ohiduzzaman Shuvo, and M. Masudur Rahman. *Defectors: A Large, Diverse Python Dataset for Defect Prediction.* In Proceeding of The 20th International Conference on Mining Software Repositories (MSR 2023), pp. 393-397, Melbourne, Australia, May 2023.

- *Parvez Mahbub*, M. Masudur Rahman, Ohiduzzaman Shuvo, and Avinash Gopal. *Bugsplainer: Leveraging Code Structures to Explain Software Bugs with Neural Machine Translation.* In Proceeding of The 39th IEEE International Conference on Software Maintenance and Evolution (ICSME 2023), pp. 5, Bogota, Columbia, October 2023 (to appear).

Based on this thesis work, two more papers are ready to be submitted to a major software engineering conference.

- *Parvez Mahbub*, and M. Masudur Rahman. *Predicting Line-Level Defects by Capturing Code Contexts with Hierarchical Transformers.* In Proceeding of The 31st IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2024), pp. 12, Rovaniemi, Finland, March 2024 (to be submitted).

- *Parvez Mahbub*, and M. Masudur Rahman. *Bugsplorer: Predicting Line-Level Defects by Capturing Code Contexts with Hierarchical Transformers.* In Proceeding of The 31st IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2024), pp. 5, Rovaniemi, Finland, March 2024 (to be submitted).

Finally, the findings of our second study (Bugsplainer) inspired another research paper in a major software engineering conference where I am the second author.

- Ohiduzzaman Shuvo, *Parvez Mahbub*, and M. Masudur Rahman. Recommending Code Reviews Leveraging Code Changes with Structured Information Retrieval. In Proceeding of The 39th IEEE International Conference on Software Maintenance and Evolution (ICSME 2023), pp. 12, Bogota, Columbia, October 2023 (To appear).

1.5 Outline of the Thesis

The thesis contains five chapters in total. To deal with software defects effectively and efficiently, 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 and structure-based traversal) that are required to follow the rest of the thesis.
- Chapter 3 discusses our first study that proposes *Bugsplorer*, a novel transformer-based technique that predicts defects at the line level leveraging a hierarchical structure of transformer models.
- Chapter 4 discusses our second study that proposes *Bugsplainer*, a novel transformer-based generative model to generate natural language explanations for software defects by leveraging the structural information of code.
- 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 remainder of this thesis. Section 2.1 introduces the Recurrent Neural Network (RNN) – a neural network architecture specialized in handling sequential data. Section 2.2 discusses transformer – another neural network architecture that achieves state-of-the-art performance for many text-oriented tasks. Section 2.3 discusses Neural Language Modelling (NLM) – a deep learning based approach to learn the probability distribution of a textual corpus. Section 2.4 illustrates neural machine translation – a deep neural network-based approach for automated translation. Section 2.5 describes embedding – a process of translating high-dimensional numerical data into a low-dimensional semantic representation. Section 2.6 introduces Abstract Syntax Tree (AST) – an abstract representation of the source code structure. Section 2.7 defines structure-based traversal that converts AST into a sequence of tokens preserving the structural information. Finally, Section 2.8 summarizes this chapter.

2.1 Recurrent Neural Network

Recurrent Neural Network (RNN) is a class of artificial neural networks that can process sequential or time series data. Unlike feed-forward neural networks, RNNs have a cyclic (hence recurrent) structure that allows them to use their internal state (a.k.a. memory) to store information from previous inputs and influence the current output. This memory enables them to model temporal dynamics and sequential dependencies in the data. RNNs are widely used for tasks such as natural language processing [68], [69], speech recognition [70], [71], and machine translation [47], [53], where the order and context of the input elements are essential. Several baselines in our studies use RNNs as a part of their technique [9], [53].

One of the main characteristics of RNNs is that they share parameters across different *time steps*, which reduces the number of parameters and improves generalization. However, this also poses challenges for learning, as RNNs need to propagate errors and gradients through time using an algorithm called Backpropagation Through Time (BPTT). BPTT can suffer from two problems: exploding gradients and vanishing gradients. Exploding gradients occur when the gradient becomes too large and causes numerical instability or divergence. Vanishing gradients occur when the gradient becomes too small and prevents effective learning or convergence.

2.1.1 Long Short-term Memory and Gated Recurrent Unit Models

Long Short-term Memory (LSTM) [72] and Gated Recurrent Unit (GRU) [73] are two types of RNNs that can minimize the exploding and vanishing gradient problems by using a gating mechanism that controls the information flow inside the network. The main idea is to introduce some components that can learn to selectively remember or forget the previous hidden state and update it with the current input. This way, the network can preserve the long-term dependencies and avoid the gradient from becoming too large or too small. In LSTM, these components are called the *cell state* and the *hidden state*, whereas in GRU uses only the *hidden state*. Both LSTM and GRU can handle long-term dependencies better than vanilla RNNs, but they are not perfect solutions [33], [74], [75]. They still have limitations, such as difficulty modelling very long sequences, computational complexity, and lack of interpretability [76]. Among the baselines of our studies, CommitGen [53] uses LSTM as a part of their technique, where DeepLineDP [9] uses GRU.

2.1.2 Attention Mechanism

Attention [33], [74] solves the problems of LSTM and GRU by reducing the dependency on the hidden state while encoding the entire input sequence. LSTM and GRU use a gating mechanism to control the information flow inside the network and preserve the long-term dependencies. However, they still need to rely on a single hidden state to summarize the input sequence, which might not be sufficient for remembering very long sequences and capturing the context and relevance of each input element. Attention, on the other hand, allows the model to access all the input states and estimates their importance for the current output. This way, attention can overcome the challenges of a single hidden state during input encoding. Attention mechanisms can be classified into different categories, such as self-attention [77], encoder-decoder attention (a.k.a. cross-attention) [33], [78], global attention [33], [75], and local attention [75]. Attention mechanisms have been widely used for various tasks that involve sequential data, such as machine translation [9], [47], [53], [75], natural language processing [68], [69], speech recognition [70], [71], and computer vision [74]. Several baselines in our study [9], [53] use the attention mechanism as a part of their techniques.

2.2 Transformers

Transformers are neural network architecture that relies on self-attention to encode and decode sequential data, such as natural language, source code or images. Transformers were first introduced by Vaswani *et al.* [30] as a novel way to overcome the limitations of RNNs, such as capturing long-range dependencies and parallelizing computation. Since then, transformers have achieved state-of-the-art results in various Natural Language Processing (NLP) tasks, such as machine translation [79], text summarization [80], question answering [81], [82], and natural language generation [67], [83], [84]. Our first study – Bugsplorer, uses two different transformer models in a hierarchy to estimate attention values for two levels of code elements: code token and code line. Our second study – Bugsplainer, uses a transformer model to generate explanations for the defective lines.

2.3 Neural Language Modeling

A statistical language model is a probability distribution over sequences of words [85]. Given a sequence of words, say of length L, the model assigns a probability to the whole sequence as follows.

$$P(W) = P(w_1 w_2 ... w_L) \tag{2.1}$$

A language model attempts to predict how frequently a phrase occurs within the natural use of a language. The estimation of the relative likelihood of different phrases can be used in many natural language processing tasks, especially ones that generate text as an output. For instance, language models can be used for code comment generation by predicting a word w_L given all the previous words in the comment before it [47] as follows.

$$w_L = \arg\max_{w_v \in V} P(w_v | w_{L-1} w_{L-2} ... w_1) \tag{2.2}$$

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

Neural language models use neural networks to capture the complex patterns and dependencies of natural language. Unlike traditional statistical language models (e.g., n-grams) that rely on counting word frequencies, neural language models learn distributed representations of words and sentences (e.g., word embedding) and use them to compute the probabilities of the next word, given the context [86]. Our first study – Bugsplorer – uses neural language modelling to predict the probability of a line being defective, given a source code document. Our second study – Bugsplainer – uses neural language modelling to generate explanations for defective source code.

2.4 Neural Machine Translation

Neural Machine Translation (NMT) is a deep neural network-based approach for automated translation [78]. In recent years, NMT has achieved rapid progress and has drawn the attention of both the research community and the practitioners. Generally, an NMT model is composed of two different blocks: *encoder* and *decoder*. The encoder accepts an input sequence and produces a numerical, intermediate representation of the input using Neural Language Modeling. Then, this intermediate representation is passed to the decoder. Based on this intermediate representation, the decoder generates the target sequence, one token at a time. While generating each token, all the previously generated tokens are also passed to the decoder. Such generation of tokens is known as *autoregressive process*, where the current output is based on all previously generated outputs [30]. In our second study, we use Transformer [30], [84], the state-of-the-art NMT model, as a part of Bugsplainer, to generate explanations for the defective source code.

2.5 Embedding

Embedding is a process of translating high-dimensional data (i.e., one-hot encoding) into low-dimensional numerical representations that capture the semantics or features of the data [31]. Embeddings make it easier for the machines to learn from large inputs by reducing the dimensionality and enabling numerical similarity measures [31], [87]. Both Bugsplorer and Bugsplainer use word embedding and positional embedding to represent the source code in our machine-learning models.

2.5.1 Word Embedding

Word embedding is a distributed representation of words in a vector space model where semantically similar words appear close to each other [31], [88]. An embedding function $E : \mathcal{W} \rightarrow \mathbb{R}^d$ takes an input word w in the domain $\mathcal{W}$ and produces its vector representation in a d-dimensional vector space [89]. The vector is distributed in the sense that a single value in the vector does not convey any meaning; rather, the vector as a whole represents the semantics of the input word. Word embedding has the potential to overcome many limitations of traditional vector representations, such as the sparse representation problem or the vocabulary mismatch issue [31].

2.5.2 Positional Embedding

Positional embedding is a way of incorporating positional information of the input into a model that uses self-attention, such as a transformer [30]. Without the positional information, the model cannot retain the order of the tokens in the input text and treats them as a bag of tokens. Positional embedding can be learnt during the training phase [90] or be predetermined (e.g., using sinusoidal functions) [30]. The most common forms of positional embedding are Absolute Positional Embedding (APE) and Relative Positional Embedding (RPE). APE assigns a vector to each input element based on its absolute position in the input sequence and combines it with the input token embedding. RPE represents the positional difference between each pair of tokens as a vector and incorporates it into the input embedding. RPE can capture long-range dependencies and handle variable-length inputs [91]. Bugsplorer uses positional embedding to incorporate the positional information into the input embedding.

```
name = "John Doe"
print("Hello " + name)
```

(a) An Example Code

```
Module(
    body=[
        Assign(
            targets=[
                Name(id='name', ctx=Store())],
            value=Constant(value='John Doe')),
        Expr(
            value=Call(
                func=Name(id='print', ctx=Load()),
                args=[
                    BinOp(
                        left=Constant(value='Hello '),
                        op=Add(),
                        right=Name(id='name', ctx=Load()))],
                keywords=[]))],
    type_ignores=[])
```

(b) Abstract Syntax Tree of The Code

Figure 2.1: An example of Abstract Syntax Tree

2.6 Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of texts (often source code) written in a formal language [92]. Each tree node denotes a construct occurring in the source code document, such as a variable, a function, a loop, and others. The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. ASTs are widely used in compilers to represent the structure of program code [92], and every major programming language defines its own AST [92]. Such a property makes AST ideal for use in any language-agnostic tool [47], [55]. Bugsplainer uses AST through Structure-Based Traversal (see Section 2.7) to capture the structure of the buggy source code. Figure 2.1 shows (a) an example code segment and (b) corresponding AST.

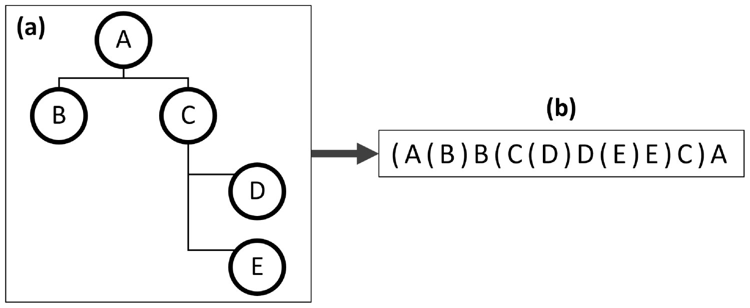


Figure 2.2: Structure-Based Traversal (SBT) – (a) an example tree, and (b) corresponding SBT sequence

2.7 Structure-Based Traversal

Traditional NMT models treat their input as sequential data (e.g., English language texts). However, source code is rich in structures (e.g., syntactic or data dependencies), which are essential to convey the semantics of the code. To leverage this structural information, several studies represent the tree structure into a sequence of code tokens. Then, they use this token sequence as the input to sequence-to-sequence models [47], [55]. Hu *et al.* [47] propose Structure-Based Traversal (SBT) converting an AST node into a token sequence that can preserve the structural information. Fig. 2.2 shows an example tree and its corresponding SBT sequence. We use the SBT algorithm to generate structurally rich sequences from source code segments (see Section 4.3.2, Algorithm 1 for details).

To generate the SBT sequence of a tree, we first use a pair of brackets to represent the tree structure and put the root node (i.e., `A`) behind both brackets, i.e., `(A)A`. Next, we traverse the sub-trees of the root node and put all root nodes of sub-trees into the brackets, i.e., `(A(B)B(C)C)A`. Recursively, we traverse each sub-tree until all tree nodes are traversed and the final sequence is obtained. For example, we get the following SBT sequence –`(A(B)B(C(D)D(E)E)C)A` – for the example tree in Fig. 2.2.

2.8 Summary

In this chapter, we introduced different terminologies and background concepts that would help one to follow the remainder of the thesis. We discussed RNNs and transformers – the state-of-the-art architectures of artificial neural networks for sequential data. We also discussed word embedding and positional embedding, which are widely used to represent textual data numerically. Finally, we discussed structure-based traversal that is used to represent AST into sequential data.

Chapter 3

Bugsplorer: Predicting Line-Level Defects by Capturing Code Contexts with Hierarchical Transformers

Software defects consume 40% of the total budget in software development and cost the global economy billions of dollars annually. Unfortunately, despite the use of many Software Quality Assurance (SQA) practices in software development (e.g., code review, continuous integration), software defects may still exist in the public release of a software product. Therefore, prioritizing SQA efforts for the vulnerable areas of the codebase is essential to ensure the high quality of a software release. In this chapter, we discuss our first study – Bugsplorer – a novel deep-learning technique aiming to reduce SQA costs by ranking the defect-prone lines.

The rest of this chapter is organized as follows. Section 3.1 introduces the study and highlights the novelty of our contribution. Section 3.2 illustrates the usefulness of our technique with a motivating example. Section 3.3 presents our proposed technique for predicting line-level defects by capturing code context with hierarchical transformers. Section 3.4 discusses our experimental design, datasets, and evaluation results. Section 3.5 introduces a working prototype of our study. Section 3.6 identifies possible threats to the validity of our work. Section 3.7 discusses our manual analysis exploring the strengths and weaknesses of our technique. Section 3.8 discusses the existing studies related to our research. Finally, Section 3.9 summarizes this study.

3.1 Introduction

A software bug (a.k.a. software defect) is an erroneous step, process, or data definition in a computer program that leads to unexpected program behaviours [1]. The resolution of bugs has been one of the major tasks of software development and maintenance. According to several studies, it consumes up to 40% of the total budget [4] and costs the global economy billions of dollars each year [3], [24]. Software Quality Assurance (SQA) practices play a critical role in preventing these defects. However,

```
name_str = ""
for name in names:
    name_str = name
name_str = sanitize(name_str)
```

Figure 3.1: An example of defective code

despite using many SQA practices in the development phase (e.g., code review, continuous integration), software defects may still exist in the official release of a software product [7], [8]. Interestingly, a recent study [9] suggests that only ≈ 3% lines of code from the whole release could lead to most of the defects. Therefore, prioritizing SQA efforts for the vulnerable areas of the codebase is essential to achieve the high quality of a software release.

Defect prediction has been a popular research topic for the last few decades [9], [20], [93]. It identifies defects in the software code during its development, which could be useful in improving the software quality before releasing the product to end users. It can also help prioritize the SQA efforts [9]. Software defects can be predicted at various abstraction levels of code such as module [15], [16], file [17], [18], method [19], and line [9], [20], [21]. Among them, line-level defect prediction provides the most fine-grained location of a software defect, which can reduce the effort to address the defect [9], [64].

The majority of the contemporary approaches for line-level defect prediction first train their machine learning models to predict the defective source code files [9], [20] or commits [21]. Then, if a file or commit is predicted as defective, they identify the tokens in the file that help explain the defects using various techniques (e.g., attention mechanism [30]). Finally, they mark the code lines containing many defect-explaining tokens as defective lines. However, such an approach poses two major challenges, as follows.

Existing models might not optimally represent code elements: In natural language texts, the semantics of a word is often determined by its context (e.g., surrounding words) [31]. Similarly, the surrounding tokens from both sides could influence the meaning and intent of a code token. For example, Fig. 3.1 shows a piece of defective code, where a code token – `name_str` – contains an erroneous value after the program execution. That is, inside the for loop, the variable `name` should be concatenated (i.e., `+=` operator) to `name_str` instead of being assigned (i.e., `=` operator). Therefore, the code token `name_str` is triggered to be buggy by another code token, "`=`", which appeared later. The intent of the token `name_str` is also influenced by the earlier tokens, such as the token `for`, by repeating the assignment multiple times. Such a phenomenon indicates that we need information on the surrounding tokens from both sides to represent a token optimally. However, the techniques used in existing study [9] (e.g., Recurrent Neural Network) can only focus on a single direction (a.k.a. unidirectional), which could be either earlier tokens or later tokens. Then, they concatenate two unidirectional representations of a token's context to generate a bidirectional representation. However, Reimers *et al.* [32] suggest that simple concatenation of two vectors might not produce an optimal representation for an input (e.g., a token or line).

Existing models might fail to capture the local context of a defect: During the training phase of the existing techniques [9], [20], [21], the attention values [33] or importance values [34] for the tokens are optimized for file-level defect prediction. In other words, these values are optimized to predict whether the whole file is defective or not. However, source code documents are often quite large, containing thousands of tokens, which could make them noisy. Therefore, the attention or importance values from existing models might fail to properly capture the local context of a software defect since, in a codebase, only 0.03-2.9% lines could be defective [9]. Thus, relying on these attention or importance values might not be sufficient to detect line-level defects accurately.

In this study, we propose – *Bugsplorer* – a novel deep-learning technique for line-level defect prediction. It leverages two transformer models in a hierarchical structure to estimate the attention values for two types of code elements: code tokens and code lines. Our solution can address the above challenges, which makes our work *novel*.

First, unlike existing techniques [9], [20], [21], Bugsplorer is directly trained for line-level defect prediction and thus can better capture the local context of a defect. Second, unlike sequential models, Bugsplorer can learn the representation of a code element by capturing its context from both earlier and later tokens simultaneously. Thus, our approach is better suited to predict line-level defects.

We train and evaluate Bugsplorer with two different benchmark datasets. The first dataset [64] consists of $\approx$ 230K Python source code documents from 24 GitHub repositories. The second dataset [20] consists of 32 software releases that span nine open-source Java software systems. We find that Bugsplorer can predict defective code lines with 26-68% higher accuracy than that of the state-of-the-art technique. It can also reduce the effort in finding defective lines by 72-81%. Using an ablation study, we further show that (a) optimizing deep learning models for line-level defect prediction and (b) generating bidirectional representations of code elements (e.g., tokens and lines) can significantly influence the performance of our technique.

We make the following contribution in this study.

(a) A novel technique – Bugsplorer, for line-level defect prediction leveraging hierarchically structured transformers.

(b) A large benchmark dataset [64] to evaluate line-level defect prediction.

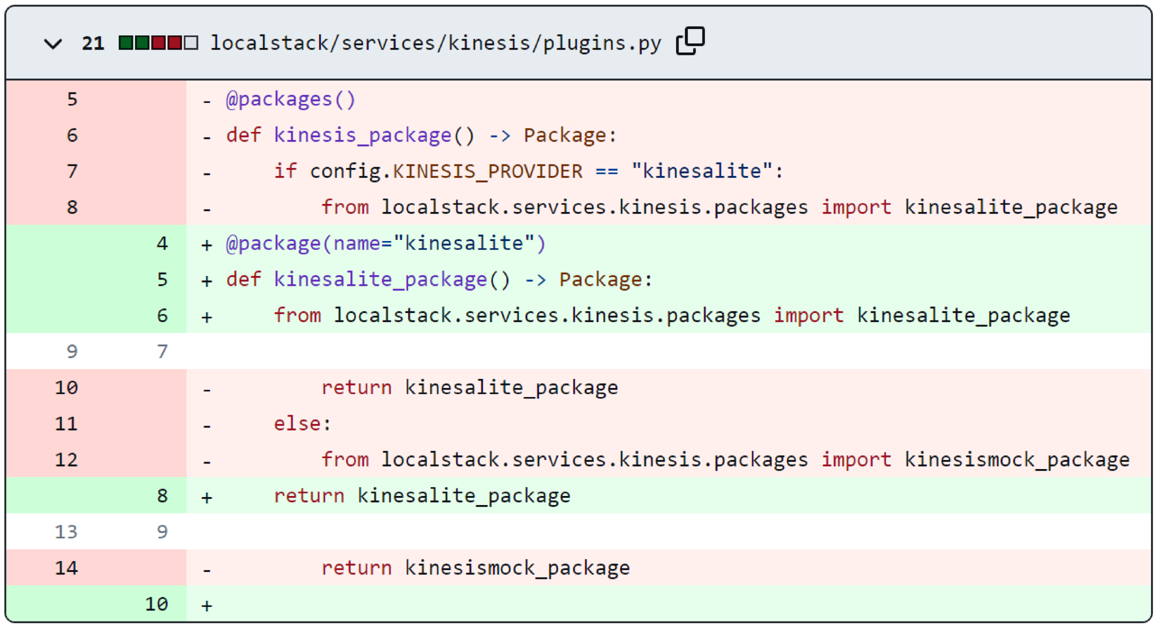
(c) A comprehensive evaluation and validation of the Bugsplorer technique in terms of both classification performance and cost-effectiveness using two different benchmark datasets of Python and Java software systems.

(d) A replication package (Appendix A.2) that includes our working prototype and other configuration details for the replication or third-party reuse.

3.2 Motivating Example

To demonstrate the capability of our technique – Bugsplorer, let us consider the example in Fig. 3.2. The code snippet is taken from the `ray-project/ray` repository at GitHub[1]. The buggy code attempts to return the driver for the *Amazon Kinesis*

[1]https://bit.ly/3N1NSOf



```
⌄ 21 🟩🟩🟥🟥⬜ localstack/services/kinesis/plugins.py
 5      - @packages()
 6      - def kinesis_package() -> Package:
 7      -     if config.KINESIS_PROVIDER == "kinesalite":
 8      -         from localstack.services.kinesis.packages import kinesalite_package
     4  + @package(name="kinesalite")
     5  + def kinesalite_package() -> Package:
     6  +     from localstack.services.kinesis.packages import kinesalite_package
 9   7
10      -         return kinesalite_package
11      -     else:
12      -         from localstack.services.kinesis.packages import kinesismock_package
     8  +     return kinesalite_package
13   9
14      -         return kinesismock_package
    10  +
```

Figure 3.2: Motivating example for Bugsplorer

service based on configuration. In particular, it returns the `kinesalite` driver if it is explicitly specified in the configuration and the `kinesismock` package otherwise. Here, the bug is that the `kinesismock` package should only be used during testing, but this function returns the package even when no configuration is available. Therefore, the defect can be identified in two places. The first one is on line 7, where the configuration is checked with an `if` condition. The second one is on line 10 and line 14, where an incorrect value is returned from the function. The state-of-the-art technique – DeepLineDP [9] – ranks these three lines beyond the 50th (line 7) and 80th (lines 10 and 14) percentiles. DeepLineDP is trained with a file-level defect prediction objective, and thus its focus is intuitively scattered over the whole file. As a result, it might fail to capture the local context of the defect. On the other hand, Bugsplorer can rank all of these defective lines within the first percentile. Since Bugsplorer is trained with a line-level defect prediction objective, it can focus more on individual lines while making a prediction. Thus, Bugsplorer can better pinpoint the defective lines and rank them higher in the list of suspicious lines.

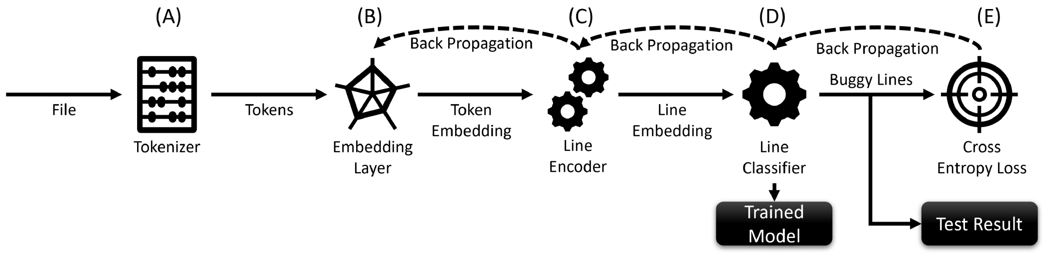


Figure 3.3: Schematic diagram of Bugsplorer

3.3 Methodology

Fig. 3.3 shows the schematic diagram of our proposed technique – *Bugsplorer* – for predicting software defects at the line level. We discuss different steps of our technique in detail as follows.

3.3.1 Pre-processing and Tokenization

Unlike many deep-learning (DL) models trained on code that treat source code documents as a stream of tokens [20], [21], [67], [94], we capture the hierarchical structure of source code documents (i.e., tokens forming lines and lines forming files). We split each source code document into lines and represented them as a list of strings, where each string denotes a source code line (Fig. 3.3, Step A). Then, we use a Byte-Pair Encoding (BPE) tokenizer [95] to convert each line into distinct tokens. BPE is a tokenizer that attempts to map a token to the largest possible sub-word in the vocabulary and falls back to smaller sub-words and even to a single letter in case of rare words. After tokenization, it encodes (i.e., maps) each token into a distinct integer from its vocabulary. Due to its fallback nature, BPE can encode any text unless it faces a completely new letter. In such a case, BPE uses a special unknown token to represent that letter. Unlike traditional rule-based tokenizers, BPE requires a training phase where it learns to represent the most common tokens as a single word and split rare tokens into sub-words or even letters.

After the encoding, we represent each source code document as an integer matrix of shape (L, T), where L is the maximum number of lines in a file and T is the maximum number of tokens in a line. Each cell in the matrix contains an integer value that denotes a token from the source code document. If a line has more tokens

```
class ScoreNames:
    ...
    def get_labels():
        return labels

class Scorer:
    score = None
```

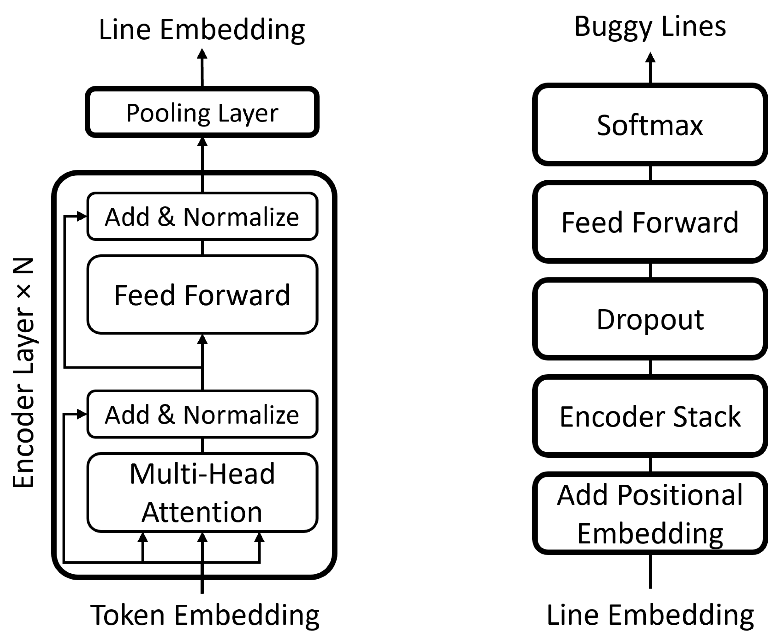
Figure 3.4: An example of structural distance

than T, then we truncate the remaining tokens from the line. If any line has fewer tokens than T, we fill the lines with a special padding token. If a file has more lines than L, then we split the file into multiple entries with N_O lines of overlap. For example, if a file has $2L - N_O$ lines, we make one split from line 1 to L and another from line $L - N_O + 1$ to line $2L - N_O$. On the contrary, if a file has fewer lines than L, we fill the files with lines containing only padding tokens.

3.3.2 Token Embedding Generation

Bugsplorer uses both word embedding and positional embedding to represent the source code tokens (Fig. 3.3, Step B). It starts with a word embedding layer that encodes each token into a vector representing the semantics of the token. The word embedding layer takes each source code document as an input and outputs a 3-dimensional matrix (L, T, d_{model}), where d_{model} is the size of a vector representing a token. Then, we pass this matrix to the positional embedding layer, which adds the positional information to the model. The positional embedding informs the model which token comes after which. In the original transformer model [30], positional embedding was statically defined as a sinusoidal wave. However, such a definition does not always reflect the structural distance between two tokens. For instance, let us consider the code example in Fig. 3.4. There the code tokens `labels` and `Scorer` belong to different class definitions. Therefore, even though they are only two tokens apart, their structural distance is way larger. Thus, to adapt to the structural aspects of source code documents, we learn and optimize the positional embedding of each token during the training phase. Finally, similar state-of-the-art transformer architectures (e.g., BERT [79], RoBERTa [90]) we sum both word embedding and positional embedding and pass the new matrix of shape (L, T, d_{model}) to the line encoder.

3.3.3 Line Embedding Generation



(a) Line Encoder (b) Line Classifier

Figure 3.5: Internal architecture of the line and line classifier

In this step, we pass the matrix representing a source code document (with semantic and positional information) to a transformer network. We call it the *line encoder* (Fig. 3.3, Step C). For each line, the line encoder outputs a d_{model}-dimensional vector representing the semantics of the line. Thus, the output of our line encoder is a matrix of shape $(L,\ d_{model})$.

Fig. 3.5a illustrates a high-level overview of the line encoder. The encoder stack has N identical layers. Each layer has two parts: multi-head self-attention and a position-wise fully connected feed-forward neural network. The attention network implements a "Scaled Dot-Product Attention" [30], defined as follows.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_{model}}})V \tag{3.1}$$

$$softmax(x \in X) = \frac{e^x}{\sum_{x \in X} e^x} \tag{3.2}$$

Here, Q, K, and V refer to query, key, and value vectors for each code line. Bugsplorer uses the same token embedding captured above (Section 3.3.2) to represent each token in these three vectors. That is, we query the line with all individual tokens in the line to find the most informative tokens. Then, during the back-propagation phase, each token learns to attend to all other tokens to determine their relative importance within the same line. We aimed to learn an optimized representation of each source code token for the objective – line-level defect prediction. The encoder stack outputs a matrix of shape (L, T, d_{model}). This means, at this stage, we still have a vector representation of size d_{model} for every token in the file. Interestingly, the vector representations at this stage know other tokens in the same line and their relative importance in predicting defective lines. Now, to generate a vector representation for each line, we pass the matrix of shape (L, T, d_{model}) to a feed-forward network to capture line-level representation, commonly known as the pooling layer. Unlike most CNN models that use a fixed pooling method (e.g., max pooling or average pooling), most transformer models (e.g., RoBERTa, T5) use a feed-forward network as the pooling layer. This layer takes the vectors representing all tokens in a line as input and produces a single vector representing the source code line. During the training, this layer learns to extract important information to detect defective code lines. Thus, for each file, the pooling layer outputs a matrix of shape (L, d_{model}), where each row is a vector representing the semantics of a line. This matrix is the final output of our line encoder.

3.3.4 Line Classification

The *line classifier* (Fig. 3.3, Step D) accepts the vector representation of each line and determines their defect-proneness. Fig. 3.5b illustrates a high-level overview of our Line Classifier module. It starts with a positional embedding layer that adds the positional information of *each line* to their line embedding. Similar to the positional embedding in the standalone Embedding Layer (i.e., Fig. 3.3, Step B), the positional embeddings of lines are also learned during the training phase. The positional embedding layer is followed by the same encoder stack as the line encoder. The encoder stack accepts the line embeddings as the input and outputs a new representation of the source code lines. In particular, the encoder stack applies self-attention to the whole source document. In this output, each line attends to every other line to determine their relative importance within the same document. Our goal was to find an optimized representation of each line by capturing not only their local but also global contexts. This encoder stack has the same structures and hyper-parameters as the line encoder; thus, the details were skipped for brevity. Then, the output of the encoder stack is passed to a feed-forward network via a dropout layer. This feed-forward network outputs two values for each line indicating whether the line is defective or not. Finally, we pass these values to a softmax layer, which performs a non-linear transformation to ensure the sum of two corresponding values is always 1. The softmax function is defined in Eq. 3.2. Finally, we have a matrix of shape (L, 2), indicating the probability of each line being defect-free and defective. During the testing phase, we use the output of the line classifier to compare with the ground truth.

3.3.5 Optimization

After every training run, we identify the number of mistakes the model made using a loss function. Then, an optimizer algorithm identifies which nodes are responsible for these mistakes and adjust their weight accordingly. The amount of adjustment is dictated by a hyper-parameter named learning rate. We use cross-entropy loss [96] and AdamW optimizer [97] (Fig. 3.3, Step E). The cross-entropy loss is defined as the number of bits needed to express the difference between two probability distributions. Mathematically, it is defined as

$$CE(x, y) = \frac{\sum_{c \in C} w_c (y_c \cdot log x_c + (1 - y_c) \cdot log(1 - x_c))}{\sum_{c \in C} w_c} \tag{3.3}$$

Here, C is the set of classes, w_c is the weight or relative importance of class c, and x and y are two probability distributions. In our case, x and y represent prediction and ground truth, respectively.

AdamW is an improvement over the more common Adam optimizer [98]. The main difference between AdamW and Adam is how they implement regularization (i.e., preventing the model from overfitting). AdamW enables a model to optimize some parameters while keeping the others unchanged. Such optimization has been shown to lead a model to faster convergence and improved generalization performance [97].

We also use a linear scheduler to reduce the learning rate over time. The convergence of deep-learning models depends heavily on the learning rate. A large learning rate may prevent reaching the minimum loss, while a small one slows down the training. This is particularly challenging for large models with millions of parameters. Therefore, it is a common practice to start with a moderately high learning rate (e.g., 5×10^{-4}) so that the model starts to learn fast and then to reduce it over time when the model reaches near the minimum loss value. Existing baseline models like RoBERTa [90] and CodeT5 [67] also used a linear scheduler to reduce their learning rate over time, which might justify our choice.

3.4 Experiment

We evaluate Bugsplorer with two large datasets based on 9 Java and 24 Python projects. We examine its classification performance as well as its ability to rank the defective lines higher. In particular, we use five appropriate metrics from the relevant literature – AuROC [99], Balanced Accuracy [100], Recall@Top20%LOC [9], [20], [21], Effort@Top20%Recall [9], [21], and Initial False Alarm [9], [20], [21]. We also examine whether optimizing the model for line-level defect prediction can improve its prediction accuracy. To place our work in the literature, we also compare our work with the existing state-of-the-art technique for line-level defect prediction. In our experiments, we thus answer four research questions as follows.

- **RQ$_1$**: How does Bugsplorer perform at line-level defect prediction in terms of classification performance and cost-effectiveness?

- **RQ$_2$**: How do (a) the bidirectional representation of code elements (tokens and lines) and (b) the optimization of the model to line-level defect prediction affect Bugsplorer's performance?

- **RQ$_3$**: How does the choice of transformer architecture affect the performance of Bugsplorer?

- **RQ$_4$**: Can Bugsplorer outperform the existing state-of-the-art technique in terms of classification performance and cost-effectiveness?

Table 3.1: Summary of the benchmark datasets

Dataset	**Defectors**	**LineDP**
# Files	213,419	73,395
# Defective Files	93,668 (44%)	4,092 (6%)
# Defect-Free Files	119,751 (56%)	69,303 (94%)
Defective Lines in Defective Files	4%	0.34%

3.4.1 Experimental Datasets

To evaluate Bugsplorer, we use a benchmark dataset of Java software systems – LineDP [20] and construct another dataset – Defectors [64] – with Python software systems. These two datasets are large enough to facilitate a comprehensive evaluation and diverse enough (e.g., 25 organizations, 18 domains, and two programming languages) to offer generalizability in findings. Table 3.1 provides the summary statistics of our benchmark datasets.

LineDP is a large dataset for line-level defect prediction containing 32 software releases from nine Java-based open-source software systems. Each release contains 731 – 8K files, 74K – 567K lines of code, and 58K – 621K code tokens. All bug reports were retrieved from the JIRA Issue Tracking System (ITS) for each system. Then, the authors collect the bug-fixing changes associated with each bug-reporting issue. They also used the SZZ algorithm [101] to identify defect-inducing changes from the bug-fixing changes. LeClair *et al.* [102] suggest that the training set should contain instances older than the testing set for an unbiased evaluation. Thus, we keep the last release for each software system (total of 9) for testing, the second last release for each system (total of 9) for validation, and the remaining early releases (total of 14) for training. This provides $\approx$ 19K files for training, $\approx$ 10K for validation, and $\approx$ 24K for testing.

Defectors is another dataset for line-level defect prediction that we constructed to achieve the generalizability of our work. Even though there exist several benchmark datasets for defect prediction, they are limited by several aspects. First, the performance of deep-learning models often scales with the size of their dataset [103], [104].

However, most of the existing datasets used in defect prediction might not be large enough [93] (e.g., Kamei *et al.* [105], McIntosh *et al.* [106]). Second, these datasets also suffer from the class imbalance problem containing only 5%-26% defective instances [93], [105], [106]. Such an imbalance could lead to sub-optimal performance with any deep-learning models. Third, these datasets were constructed either from a small number of projects [106] or the projects from a single organization [93], [107]. Such a choice limits the capability of the models to generalize their performances across different domains and organizations. Finally, most of the existing datasets are constructed from Java-based software systems.

To mitigate the challenges with existing datasets and to achieve generalizability in our findings, we constructed *Defectors* – a large-scale dataset containing both source code and their defective lines from 24 popular Python projects across 18 domains and 24 organizations. We carefully identify defective source code files and their code changes, following five levels of noise filtration recommended in the literature [93], [105], [106]. Our dataset contains $\approx$ 213K source code files ($\approx$ 93K defective and $\approx$ 120K defect-free). We discuss our dataset construction process as follows.

Project Selection

Most of the existing datasets used in defect prediction are constructed using Java projects. To diversify our datasets, we thus choose Python-based projects. Similar to existing studies [93], we sort all the Python repositories on GitHub in *descending order* using their star counts. Then, we manually investigate this ordered list of repositories sequentially to find mature and high-quality repositories. We also wanted to ensure that each repository contains sufficient bug-fix Pull Requests (PRs). The following steps summarize our repository selection process[2].

1. If a repository has less than 2000 PRs, we discard it considering it is not mature enough.

2. From a mature repository, we identify all the bug-related labels (e.g., bug, bugfix).

[2]Accessed: December 2, 2022

3. We find the PRs that contains one of these labels. We accept the repository if the number of such PRs is more than 100. We find 11 repositories with consistently labeled PRs.

4. If the number of bug-fix PRs is less than 100, we attempt to find the issues (a.k.a. bugs reports). In particular, we search for issues associated with one of these labels and have a linked pull request resolving the issue. If the number of such issues is more than 100, then we accept the repository. We find 12 repositories with consistent labeling and linked bug-fix PRs.

5. If the number of both PRs and issues are less than 100, we look for other consistent ways of labeling issues. In particular, we read the titles of the most recent 200 PRs and look for any consistent patterns. We find that the titles of bug-fix PRs from one project start with a specific keyword (i.e., fix). The titles of bug-fix PRs from another project *consistently* contain issue IDs from a different bug report management website. We accept these two projects for having such consistent patterns of labeling bug-fix PRs.

Following the steps above, we investigated the first 100 repositories from the ordered list (based on descending star count). From there, we ended up with a total of 25 projects from various organizations and domains. Later, in our quality filtration stage, we discard a repository for having only one bug-fix commit matching our criteria (see Section 3.4.1, *Bug Inducing Commit Filtration*). Table 3.2 enlists the remaining 24 projects along with their domain and bug report management system. Table 3.3 contains the ratio of defective and defect-free code changes in the projects.

Bug Fixing Commit Collection

During project selection, we identified the bug report management system of each repository. We collect the bug-fixing commits from the repositories using that information. From the 11 projects where bug-fix PRs are labeled with appropriate labels, we collect the labeled PRs. From the 12 projects where issue reports are labeled with appropriate labels, we first collect the issues and then collect their associated PRs. We collect $\approx$ 39K bug-fix PRs from the 23 projects using these two approaches. For the remaining two projects, we use slightly different approaches. `getsentry/sentry`

Table 3.2: Description of the projects used in Defectors

Project	**Domain**	**Bug Report Management***
Lightning-AI/lightning	deep-learning	labeled PR
ansible/ansible	automation	labeled PR
apache/airflow	automation	labeled issue
celery/celery	task queue, messaging	labeled PR
commaai/openpilot	autonomous driving	labeled PR
django/django	web framework	separate website
encode/django-rest-framework	web framework	labeled PR
explosion/spaCy	natural language processing	labeled PR
getredash/redash	data science	labeled PR
getsentry/sentry	logging	PR title
google/jax	deep learning	labeled issue
home-assistant/core	internet of things	labeled PR
huggingface/ transformers	deep learning	labeled issue
localstack/localstack	cloud, serverless	labeled issue
numpy/numpy	data science	labeled PR
pandas-dev/pandas	data science	labeled PR
psf/black	development tool	labeled issue
pypa/pipenv	development tool	labeled issue
python/cPython	programming language	labeled PR
python-poetry/poetry	development tool	labeled issue
ray-project/ray	machine learning, deep learning	labeled issue
scikit-learn/scikit-learn	machine learning	labeled issue
scrapy/scrapy	crawling, scraping	labeled issue
ultralytics/yolov5	deep learning, image processing	labeled issue
Total	**18 distinct domains**	

*labeled PR = bug fixing PRs are uniquely labeled, labeled issue = bug reporting issues are uniquely labeled, separate website = bug reports are managed in a separate website, PR title = bug fixing PR titles have unique pattern

Table 3.3: Number of defective, defect-free, and total source code documents used in Defectors

Project	**Defective**	**Defect-free**	**Total**
Lightning-AI/lightning	5,369 (48%)	5,793 (52%)	11,162
ansible/ansible	15,225 (35%)	28,340 (65%)	43,565
apache/airflow	4,788 (48%)	5,166 (52%)	9,954
celery/celery	805 (23%)	2,674 (77%)	3,479
commaai/openpilot	695 (44%)	885 (56%)	1,580
django/django	5,309 (47%)	5,934 (53%)	11,243
encode/django-rest-framework	269 (67%)	134 (33%)	403
explosion/spaCy	1,224 (50%)	1,231 (50%)	2,455
getredash/redash	170 (24%)	550 (76%)	720
getsentry/sentry	13,791 (37%)	23,694 (63%)	37,485
google/jax	443 (61%)	283 (39%)	726
home-assistant/core	21,535 (60%)	14,413 (40%)	35,948
huggingface/ transformers	658 (36%)	1,187 (64%)	1,845
localstack/localstack	925 (51%)	879 (49%)	1,804
numpy/numpy	466 (47%)	527 (53%)	993
pandas-dev/pandas	7,816 (50%)	7,766 (50%)	15,582
psf/black	228 (41%)	328 (59%)	556
pypa/pipenv	60 (50%)	61 (50%)	121
python/cPython	1,769 (24%)	5,507 (76%)	7,276
python-poetry/poetry	1,283 (57%)	949 (43%)	2,232
ray-project/ray	7,405 (42%)	10,336 (58%)	17,741
scikit-learn/scikit-learn	2,565 (51%)	2,434 (49%)	4,999
scrapy/scrapy	221 (52%)	204 (48%)	425
ultralytics/yolov5	649 (58%)	476 (42%)	1,125
Total	**93,668 (44%)**	**119,751 (56%)**	**213,419**

*The values in the Defective, Defect-free, and Total columns are before creating train, validation, and test splits

adopts a pattern where all bug-fix PRs start with the keyword – *fix*. Therefore, we collect the PRs with such a pattern. Finally, `django/django` uses a separate Issue

Tracking System (ITS) website[3] to manage their issues. In GitHub, its PRs contain corresponding issue IDs from the ITS. We first collect the closed bug reports from the ITS and then capture the corresponding PRs from GitHub. Once we have the bug-fix PRs, we collect their merge commits as the bug-fix commits. This way, we collect $\approx$ 51K bug-fix commits from the 25 projects.

Bug Inducing Commit Collection

In this step, we capture the bug-inducing commits (i.e., changes introducing a bug) from the bug-fix commits using the SZZ algorithm [101]. Most of the studies in defect prediction [93], [106]–[109] use the SZZ algorithm to identify the bug-inducing changes. We use an implementation of the SZZ algorithm by the PyDriller tool [110]. This implementation takes a bug-fix commit as the input and returns a list of commits that modify the defective lines in the input commit.

Bug Inducing Commits Filtration

The SZZ algorithm provides a considerable amount of false positives, i.e., identifies defect-free commits as defective commits [107]. Thus, we apply a series of filtration inspired by the literature [93], [105], [106] to minimize the number of false positives.

Filtration Using the Number of Linked Bug Inducing Commits: SZZ often links several bug-inducing commits to a single bug-fix commit. This suggests that a bug could occur due to non-coherent changes in hundreds or even thousands of files, which is impractical. Therefore, existing studies discard the bug-fix commits that are linked to too many bug-inducing commits [93], [105], [106]. Let *inducer-count* be the number of bug-inducing commits linked to a single bug-fix commit. Keshavarz *et al.* [93] suggest using Equation 3.4 as a threshold.

$$thresh(X) = mean(X) + std(X) \tag{3.4}$$

In this work, we use a threshold of 14, derived from the above equation, to filter out the noisy bug-fix commits.

[3]https://code.djangoproject.com

Filtration Using the Number of Linked Bug-fix Commits: Let *fixer-count* be the number of bug-fix commits linked to a single bug-inducing commit. If a bug-inducing commit has a *fixer-count* greater than one, it suggests that the commit induced multiple bugs in the project. Similar to *inducer-count*, we apply Equation 3.4 to *fixer-count* as well. Thus, we discard the bug-inducing commits with a *fixer-count* higher than 7 – derived from the above equation.

Filtration Using the Size of Changed Code: If a commit changes a large number of lines or files, it indicates that the commit might contain *tangled* changes. During our manual analysis, we found several commits that modified even up to 1,000 files. Often these commits indicate some administrative tasks, such as merging several related projects into a single repository. Therefore, existing studies [93], [106] filter out the large commits. Similarly, we filter out the bug-inducing commits that have more than 1000 changed lines or have touched more than 100 files.

Filtration Using the File Type: In this dataset, we focus specifically on Python source code. Such a language constraint makes performing static analysis on source code easy. It also helps us capture the structural information from source code (e.g., changed methods). Even though Python is the main language of all our projects, they contain a small fraction of non-Python files (e.g., configuration files). Thus, we filter out the commits that do not modify any Python file.

Filtration Using the Nature of Change: All the changes in a source code file might not be bug-inducing. For instance, comments or code formatting changes generally do not introduce new bugs. As done by existing studies [93], [108], [111], we thus discard such trivial changes. Trivial changes do not modify the abstract syntax tree (AST) of the source code. Therefore, to identify trivial changes, we compare the AST of the source code before the commit to the AST after the commit. If both ASTs are the same, the commit performs a trivial change and is thus discarded. Our implementation of this filtration is tolerant of syntax errors. That is, if the source code is not syntactically correct, our implementation will still generate partial AST for comparison.

After completing all these filtrations, we find that one project, namely – *freqtrade/freqtrade*, contains only one bug-fix commit. We thus discard the project and keep the remaining 24 repositories in our dataset.

Collecting and Sampling Defect-free Commits

We collect all the commits within the date range of the defective (i.e., bug-inducing) commits from the same project. Then, we separate the defective commits from the defect-free commits using the commit hashes. In software projects, defect-free commits often outnumber defective commits by a large margin. Therefore, we down-sample the defect-free commits to ensure a near 1:1 class ratio. We sample defect-free commits from each project with a 95% confidence level and a 5% margin of error. If a sample size is less than the number of defective commits, then we increase the sample size to achieve parity. Finally, we discard the defect-free commits that do not modify any Python file.

Construction of Training and Testing Data

We formalize our dataset by targeting line-level defect prediction from source code documents. Here, the input is the content of a file after the commit. If the commit is defective, the output is the list of added (i.e., defective) line numbers. Otherwise, the output is an empty list. We make train, validation, and test sets based on both random and timewise splitting approaches. In the random setting, we order the commits randomly and take 10,000 commits for testing, 10,000 for validation, and the remaining for training. In the timewise setting, we order the commits ascendingly based on their commit time. Then we take the last 10,000 commits for testing, the second last 10,000 for validation, and the remaining for training. This way, we train the model with older data and keep the latest data for evaluation and testing. The training splits maintain a near 1:1 ratio of defective and defect-free instances, whereas test and validation splits maintain the *original distribution*. In particular, we found that $\approx 7\%$ files in the codebase are defective and $\approx 4\%$ lines are defective in the defective files.

3.4.2 Evaluation Metrics

We evaluate Bugsplorer both as a classification and a retrieval technique. We use five different performance metrics from the relevant literature [9], [20], [63] to evaluate our technique. Among these metrics, AuROC and balanced accuracy evaluate the classification performance, while recall@top20%LOC, effort@top20%recall, and initial false alarm evaluate the cost-effectiveness of the technique.

Area under the Receiver Operating Characteristic

AuROC is a measurement of how well a model can discriminate between two classes. The Receiver Operating Characteristic (ROC) curve is the ratio between the True-Positive Rate (TPR) and the False-Positive Rate (FPR) [99]. AuROC is the area under this curve. Mathematically, it is defined as

$$\begin{aligned} AuROC &= \int_{t=0}^{1} TPR/FPR \\ &= \int_{t=1}^{1} \left(\frac{TP}{TP+FN} / \frac{FP}{TN+FP} \right) \end{aligned} \tag{3.5}$$

where t is the threshold to convert probability scores to binary classes, and TP, FN, TN and FP refer to True Positive, False Negative, True Negative, and False Positive instances, respectively.

Balanced Accuracy

Traditional accuracy measure is often biased toward the majority class [9]. Balanced accuracy mitigates this problem by putting equal weight on the True-Positive Rate (TPR) and the True-Negative Rate (TNR) [100]. Mathematically, it is defined as

$$\begin{aligned} BA &= (TPR + TNR) \,/\, 2 \\ &= \left(\frac{TP}{TP+FN} + \frac{TN}{TN+FP} \right) / \, 2 \end{aligned} \tag{3.6}$$

Recall@Top20%LOC

This metric measures the ratio between the number of defective lines in the top 20% suspicious lines (i.e., with high bug probability) and the total number of defective lines [9]. A value of 1 for Recall@Top20%LOC means that all defective lines can be found within the top 20% suspicious lines marked by a technique. Assuming all defective lines are distributed naturally, a random guessing model will achieve a score of 0.20 for this metric. A metric value higher than 0.20 indicates that one can find more defective lines with less effort by ranking defective lines higher in the list of suspicious lines. Mathematically it is defined as

$$\begin{aligned} R@20 &= \frac{\sum_{i=1}^{N_{20}} Q(i)}{N_{defective}} \\ Q(i) &= \begin{cases} 1, & \text{if } l_i \text{ is defective.} \\ 0, & \text{otherwise.} \end{cases} \end{aligned} \tag{3.7}$$

where N_{20} is 20% of the total number of lines, $N_{defective}$ is the total number of defective lines, and l_i is the line with i^{th} highest probability of being defective.

Effort@Top20%Recall

This metric measures the ratio between the number of suspicious lines we have to investigate to find 20% of the defective lines and the total number of ranked lines [9]. A value of 1 for Effort@Top20%Recall means that to find all defective lines, all the lines from the codebase need to be investigated as ranked by a technique. Assuming all defective lines are distributed naturally, a random guessing model will achieve a 0.20 score for this metric. A *lower* metric value indicates one needs to put less effort into finding the defective lines. Mathematically it is defined as

$$E@20 = \frac{NB_{20}}{N} \tag{3.8}$$

where NB_{20} is the number of lines to inspect to find 20% of the defective lines and N is the total number of ranked lines.

Initial False Alarm

The Initial False Alarm (IFA) metric is the ratio between the number of misclassifications before the first true-positive and the total number of instances. A lower value of IFA indicates that we have to put less effort into finding the defective lines.

3.4.3 Experiment Design and Hyper-Parameters

Tokenizer: We use a Byte-Pair Encoding (BPE) tokenizer that is pre-trained on GitHub CodeSearchNet [112] dataset. The dataset contains $\approx$ 6M code snippets accompanied by documentation. Since the tokenizer is trained on code corpus (versus natural language corpus), it encodes source code with 33-50% shorter length, compared to that of GPT2 [83] or RoBERTa [90] tokenizer.

Encoder: For the two encoder stack in Line Encoder and Line Classifier, we use RoBERTa [90] transformer architecture. We empirically find that RoBERTa performs better for our research problem compared to other similar models (see Section 3.4.4). We initialize learnable parameters of the encoder stack of the *Line Encoder* using CodeBERTa pre-trained model from huggingface[4]. Similar to our tokenizer, this model too is pre-trained with the CodeSearchNet dataset. We initialize the learnable parameters of our second network, the Line Classifier, from normally distributed random values with mean = 0 and standard deviation = 0.02.

Hyper-Parameters: The hyper-parameters we have used in our experiment can be divided into two categories as described below.

Bugsplorer-specific parameters: During our experimentations, we set the maximum number of tokens in a line to 16 (i.e., $T = 16$) as the threshold, which covers 90% lines in Defectors and 99% lines in the LineDP dataset. Therefore, the possibility of data loss due to the truncation is very low, which might justify our choice. We set the maximum number of lines in a file to 512 (i.e., $L = 512$) as the threshold. While splitting large files into multiple parts, we use 64 lines of overlap (i.e., $N_O = 64$). We make our train-validation-test datasets at the file level; thus, multiple splits of the same file reside in the same dataset. This way, we ensure that the overlapping between the splits does not affect our evaluation.

RoBERTa-specific parameters: We use 6 layers in the encoder stack (i.e., $N = 6$) in both the line encoder and line classifier. In the embedding layer, line

[4]https://huggingface.co/huggingface/CodeBERTa-small-v1

encoder, and line classifier, we use hidden states of size 768 (i.e., $d_{model} = 768$). We use 12 attention heads to parallelize our training process. During the training, we use the AdamW [97] optimizer with a learning rate of 5×10^{-5}. These values are inspired by state-of-the-art transformer models [67], [94], [113]. The detailed number of parameters and configurations of different components in Bugsplorer can be found in our replication package.

Hardware: Our experiments are run on two NVidia A100 GPUs with 40GB of memory each. We use batches of 16 files in each step (i.e., 16 files $\times$ 512 lines $\times$ 16 tokens $= 131,072$ tokens). The average model training time is two days for the Defectors dataset and one day for the LineDP dataset. The average evaluation time is $\approx$ 12 minutes for the Defectors dataset (i.e., $\approx$ 72 milliseconds per file) and $\approx$ 25 minutes for the LineDP dataset (i.e., $\approx$ 62 milliseconds per file).

Computation of Evaluation Metrics: We compute the metric values by counting all the lines from all the files. In other words, each line constitutes a sample in our experiment. While computing the metric values, we discard the overlapping lines. In particular, for each overlap of size $N_O = 64$, we take $N_O/2 = 32$ lines from the first split and the next 32 lines from the next split. The number of lines in the test set is $\approx$ 5.6M in the random split of Defectors, $\approx$ 5.8M in the timewise split of Defectors, and $\approx$ 4.5M in the LineDP dataset.

3.4.4 Evaluating Bugsplorer

Answering RQ_1 – Performance of Bugsplorer

In this experiment, we evaluate Bugsplorer using five metrics in two different aspects – classification and cost-effectiveness. Fig. 3.6 and Table 3.4 show the metric scores from our experiment.

First, we explore the performance of Bugsplorer using the random split of the Defectors dataset. For this dataset, Bugsplorer scores 0.77 for balanced accuracy and 0.83 for AuROC. Such values indicate a very good capability of distinguishing true positive instances (i.e., defective lines) from true negative instances (i.e., defect-free lines). In the case of cost-effectiveness metrics, Bugsplorer achieves 0.69 for

Table 3.4: Performance metric scores of Bugsplorer

Metric	**Defectors (Random Split)**	**Defectors (Timewise Split)**	**LineDP**	**LineDP (Cross-Project Split)**
BalAcc ↑	0.769	0.784	0.901	0.872
AuROC ↑	0.829	0.841	0.920	0.892
Recall@20% ↑	0.690	0.754	0.985	0.871
Effort@20% ↓	0.025	0.027	0.037	0.036
IFA ↓	0.000	0.000	0.006	0.004

* Up arrow (↑) indicates higher is better and down arrow (↓) indicates lower is better.

recall@20%LOC, which means with the help of our technique, an SQA engineer can find 69% defective lines by only investigating 20% lines of the codebase. Similarly, Bugsplorer archives 0.025 for effort@20%recall, which means to find the first 20% defective lines, an SQA engineer has to investigate only 2.5% lines from the codebase. Finally, an Initial False Alarm (IFA) score of ≈ 0.00 indicates a very minimal effort to find the defective lines.

Even though the random split of the Defectors dataset provides an overview of the capabilities of our technique, in the real world, Bugsplorer is meant to be trained on historical data and can predict future data [102]. The timewise split of the Defectors dataset and the LineDP dataset evaluates Bugsplorer in such scenarios. While the timewise split of the Defectors dataset evaluates Bugsplorer using the latest instances from all projects, the LineDP dataset evaluates Bugsplorer using the latest data from each project. Looking at the metric scores, we see Bugsplorer continues to perform well in timewise settings as well, interestingly, even better in some cases. For balanced accuracy, Bugsplorer achieves metric scores 0.78–0.90, and for AuROC, it achieves 0.84–0.92. Such scores indicate Bugsplorer also retains its capability of distinguishing in cross-project settings. Bugsplorer is cost-effective in timewise setting as well. For recall@20%LOC, it scores 0.99 for the LineDP dataset, which means Bugsplorer can help find nearly all defective lines by investigating only 20% of the suspicious lines. For the effort@20%recall metric, Bugsplorer scores 0.027–0.037, which means an SQA engineer has to investigate only 2.7%–3.7% suspicious lines to find 20% of the defective lines. Finally, an Initial False Alarm (IFA) score of $\approx 0.00 - 0.01$ indicates very

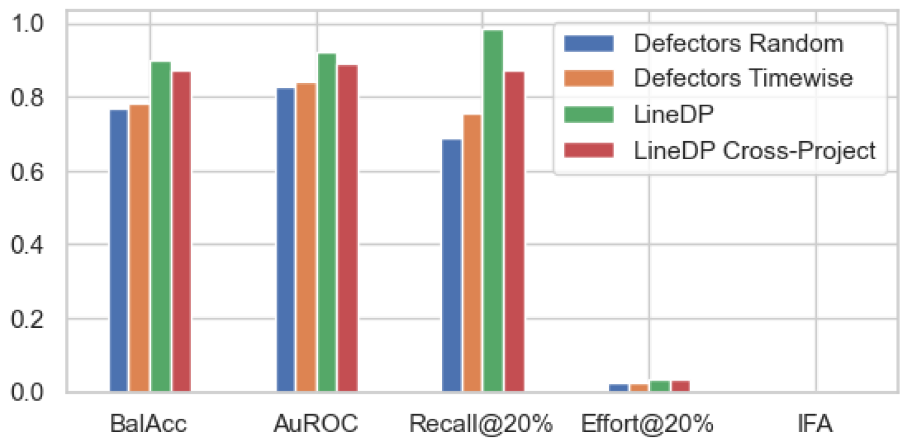


Figure 3.6: Automated metric scores of Bugsplorer

minimal overhead to find the defective lines Interestingly, even though ML models tend to perform better with randomly split data, the performance of Bugsplorer in both random and timewise split of the Defectors dataset is comparable. Such a phenomenon indicates the robustness of our technique at line-level defect prediction with unseen future data.

In practical scenarios, when integrating Bugsplorer into a new project, obtaining data specific to that project for retraining Bugsplorer may not always be feasible. As a result, we explore how the performance of Bugsplorer changes when employed in a cross-project context. This means the training, validation, and testing datasets encompass commits from entirely separate projects, ensuring mutual exclusivity. To avoid bias towards any certain project, we create nine variants of the LineDP dataset for cross-project settings. In each variant, we take one project for validation, one for testing, and the remaining eight projects for training. Such a setting ensures Bugsplorer is tested with each project. Then, we report the average score from each variant. From Table 3.4, we see that the performance of Bugsplorer in cross-project settings shows a mixed trend when compared to the performance for the original LineDP dataset (i.e., cross-release settings). For the balanced accuracy, AuROC, and recall@20%LOC metrics, the performance drops by 3%, 3%, and 12%, respectively. Still and all, the metric scores achieved by Bugsplorer in cross-project settings are very promising. Interestingly, for effort@20%recall and Initial False Alarm (IFA) metrics, Bugsplorer performs 3% and 33%, respectively. Thus, overall, Bugsplorer

Table 3.5: Effectiveness of Bi-directional Representation and Line-Level Optimization

Dataset	Metric	Bugsplorer	Bugsplorer$_{File}$	DeepLineDP
Defectors (Random Split)	**BA ↑**	0.769	0.603	0.610
	AuROC ↑	0.829	0.610	0.633
	Recall@20% ↑	0.690	0.320	0.324
	Effort@20% ↓	0.025	0.111	0.089
	IFA ↓	0.000	0.000	0.002
Defectors (Timewise Split)	**BA ↑**	0.784	0.628	0.561
	AuROC ↑	0.841	0.630	0.518
	Recall@20% ↑	0.754	0.380	0.281
	Effort@20% ↓	0.027	0.085	0.105
	IFA ↓	0.000	0.000	0.000
LineDP	**BA ↑**	0.901	0.605	0.538
	AuROC ↑	0.920	0.556	0.510
	Recall@20% ↑	0.985	0.251	0.224
	Effort@20% ↓	0.037	0.167	0.191
	IFA ↓	0.006	0.006	0.007

can significantly reduce costs even in cross-project settings.

Summary of RQ$_1$: Bugsplorer shows promising results for line-level defect prediction with a balanced accuracy up to 0.90 and an AuROC up to 0.92. It can also rank the first 20% of the defective lines from the codebase within the top 2-3% of its suspicious lines, which is promising.

Answering RQ$_2$ – Effectiveness of Bi-directional Representation of Code Contexts and Line-Level Optimization

In this experiment, we analyze the effectiveness of (a) generating bidirectional representations of code elements (e.g., tokens and lines) instead of concatenating two unidirectional representations and (b) line-level optimization during model training. To do so, we introduce a new variant of Bugsplorer – Bugsplorer$_{File}$, which is trained with the objective of file-level defect prediction. We compare (a) DeepLineDP and Bugsplorer$_{File}$ to determine the effectiveness of bidirectional representation and (b) Bugsplorer$_{File}$ and Bugsplorer to determine the effectiveness of line-level optimization during model training. Table 3.5 shows the performances of Bugsplorer, Bugsplorer$_{File}$, and DeepLineDP. Fig. 3.7 illustrates their performances using boxplots. Since, for certain metrics, increments are better, while for other decrements are better, we use the terms better performance or worse performance during our discussion. We also mark them using up-arrow and down-arrow in Table 3.5 respectively.

Bugsplorer$_{File}$ uses a transformer model to encode source code elements (e.g., tokens or lines), where DeepLineDP [9] uses a Recurrent Neural Network (RNN) model. Using a transformer model lets Bugsplorer$_{File}$ focus on surrounding tokens from both sides of a token simultaneously, leading to bidirectional representations of the lines. On the contrary, DeepLineDP generates two unidirectional representations of each line (one is left to right, and the other is right to left) and then concatenates them to generate a representation of the lines. Therefore, any difference in the performances of Bugsplorer$_{File}$ and DeepLineDP can be attributed to generating bidirectional representations. While comparing the performance between Bugsplorer$_{File}$ and DeepLineDP from Table 3.5, we see that in most cases, Bugsplorer$_{File}$ shows better performance. For the timewise split of Defectors, Bugsplorer$_{File}$ shows 12-35% better scores in balanced accuracy, AuROC, recall@20%LOC, and effort@20%recall metrics. Similarly, for the LineDP dataset, Bugsplorer$_{File}$ shows 9-15% better performance in all metrics. Finally, we see a mixed trend for the random split of Defectors. DeepLineDP shows a 1-3% better performance for balanced accuracy, AuROC, and recall@20%LOC metrics, which are marginally better. For the effort@20%recall metric, DeepLineDP archives 24% better performance (actual metric score reduced by only 0.022). Nonetheless, for the initial false alarm metric (lower is better), the score of DeepLineDP increased from ≈ 0.0 to 0.002. This means to find the first defective lines with DeepLineDP, one has to investigate 0.2% lines of the codebase, while the amount is $\approx 0\%$ for Bugsplorer. Given the evidence above, our choice of generating bidirectional representation using a transformer network might be justified.

While Bugsplorer$_{File}$ is trained with a file-level defect prediction objective, Bugsplorer is trained with a line-level defect prediction objective. Therefore, any difference in their performances can be attributed to their optimization level. Table 3.5

shows that Bugsplorer outperforms Bugsplorer$_{File}$ in nearly all metric scores across all datasets. For balanced accuracy, Bugsplorer shows 25-49% better performance, while for AuROC, the improvement is 33-65%. Such improvements imply that the line-level optimization during model training (i.e., Bugsplorer) leads to better classification performance with a strong capability of discriminating between defective and defect-free lines. In cost-effectiveness metrics, we see even bigger improvements. The line-level optimization in defect prediction achieves 98-292% better scores in terms of recall@20%LOC metric. Similarly, the effort@20%recall score is 68-78% better. Finally, the initial false alarm score is the same for both variants across all datasets. All these improvements in metric scores suggest that line-level optimization is a much better choice than file-level optimization during model training, which justifies our choice.

Summary of RQ$_2$: Both the bi-directional representation of code elements and the line-level defect prediction objective lead to better performance in our technique. Given all the evidence above, our choices regarding token representation and model optimization might be justified.

Answering RQ$_3$ – Impact of the Choice of Transformer Architecture on Bugsplorer

In this experiment, we investigate how our choice of the transformer architecture in the *encoder stack* affects the performance of Bugsplorer. In particular, we experiment with three popular transformer-based encoder architectures – RoBERTa [90], BERT [79], and T5 [84]. Among them, RoBERTa is the *default choice* of Bugsplorer. We choose these three architectures because of their extensive use in the software engineering domain and state-of-the-art performances with relevant benchmarks like CodeSearchNet [112] and CodeXGLUE [114]. To initialize the learnable parameters of Line Encoder (Section 3.3.3), we use CodeBERT [94] for BERT, CodeBERTa for RoBERTa, CodeT5 [67] for T5. All of these models were pre-trained with the CodeSearchNet dataset. The Line Encoder produces encoding for each line and passes it to the Line Classifier to predict whether a line is defective or not. We initialize the learnable parameters of Line Classifiers using normally distributed random values

Table 3.6: Performance of Bugsplorer with different transformer architectures

Dataset	**Metrics**	**RoBERTa**	**BERT**	**T5**
Defectors (Random Split)	**BA** ↑	0.769	0.769	0.709
	AuROC ↑	0.829	0.828	0.795
	Recall@20% ↑	0.690	0.690	0.572
	Effort@20% ↓	0.025	0.025	0.029
	IFA ↓	0.000	0.000	0.000
Defectors (Timewise Split)	**BA** ↑	0.784	0.778	0.710
	AuROC ↑	0.841	0.845	0.791
	Recall@20% ↑	0.754	0.754	0.577
	Effort@20% ↓	0.027	0.027	0.036
	IFA ↓	0.000	0.000	0.000
LineDP	**BA** ↑	0.901	0.849	0.909
	AuROC ↑	0.920	0.897	0.914
	Recall@20% ↑	0.985	0.871	0.995
	Effort@20% ↓	0.037	0.037	0.034
	IFA ↓	0.006	0.006	0.006

in all variants. Note that even though a T5 model contains both an encoder and a decoder, we use only the encoder part in our work. Table 3.6 shows the performance of Bugsplorer with these three transformer architectures. Since, for some metrics, increments are better while for other decrements are better, we use the terms better performance or worse performance, respectively, in our discussion. We also mark them using up-arrow and down-arrow in Table 3.6 respectively.

When comparing BERT with RoBERTa, there is no clear winner. In most cases, they achieve nearly the same performance. Even when their scores differ, the difference is only marginal in a few cases. In particular, for the random split of Defectors, both of them achieve the same scores for balanced accuracy, recall@20%LOC, effort@20%recall, and initial false alarm metrics. Only for the AuROC metric, RoBERTa shows 0.2% worse performance, which is marginal. For the timewise split of Defectors, the performance of RoBERTa varies from 0.5% worse to 0.8% better in balanced accuracy, AuROC, recall@20%LOC, and effort@20%recall metrics. For the initial false alarm metric, the score remains the same. For the LineDP dataset, RoBERTa shows 2-12% better performance in balanced accuracy, AuROC, and recall@20%LOC metrics. Both architectures achieve the same performance for effort@20%recall and initial false alarm metrics. Considering the trend in these metric scores, we see that the performance of RoBERTa is marginally better than that of BERT. Since the RoBERTa model is a successor of the BERT model, such a trend in their performances might be expected.

When comparing RoBERTa with T5, RoBERTa consistently performs better than T5 for both of the Defectors datasets but shows dissimilar patterns for the LineDP dataset. For the random split of Defectors, RoBERTa shows 4-17% better performance in balanced accuracy, AuROC, recall@20%LOC, and effort@20%recall metrics. For the timewise split of Defectors, RoBERTa consistently shows better performance (6-34%) for balanced accuracy, AuROC, recall@20%LOC, and effort@20%recall metrics. However, for the LineDP dataset, we see some mixed trends. RoBERTa shows 1% worse performance in balanced accuracy and recall@20%LOC metrics while achieving 1% better performance for the AuROC metric. Nonetheless, for the effort@20%recall metric, RoBERTa shows 9% worse performance. Finally, for the initial false alarm, both of the architectures perform the same across all datasets. Thus, T5 and RoBERTa show mixed performance trends in the LineDP dataset, whereas T5 consistently performs worse in the Defectors dataset. Since T5 is designed for both encoding and decoding, whereas RoBERTa is specialized for encoding, such performance differences among them might be explainable.

Summary of RQ$_3$: RoBERTa shows a mixed result when compared to T5, and comparable performance when compared to BERT. Such findings indicate that even though having two transformer models is essential in our technique, the choice of their architecture does not consistently impact Bugsplorer. It further confirms that the performance of Bugsplorer comes from its bidirectional representation and line-level optimization.

Answering RQ$_4$ – Comparison with the Existing Baseline Technique

In this research question, we compare Bugsplorer with the state-of-the-art technique for line-level defect prediction. In particular, we compare Bugsplorer with

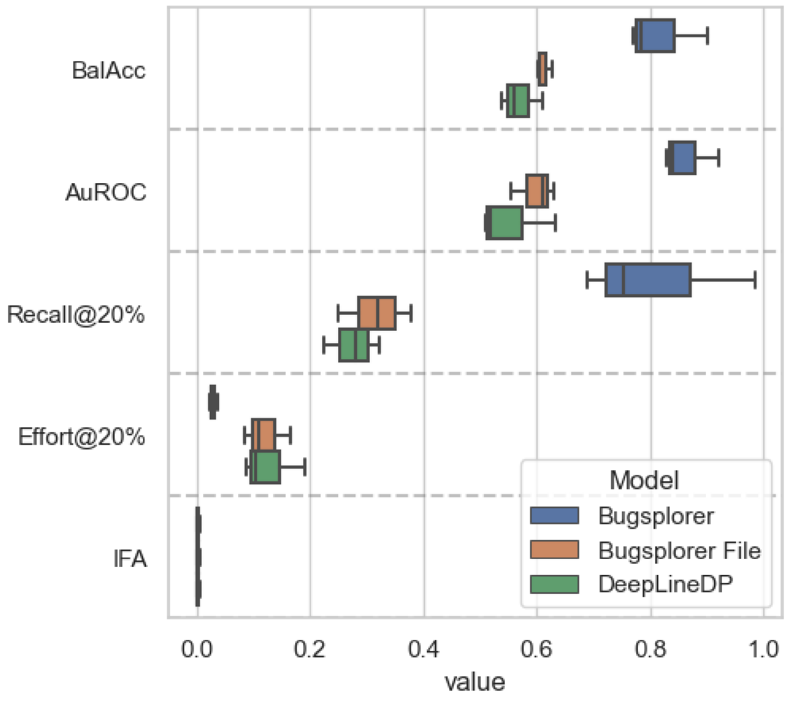


Figure 3.7: Effectiveness of Bidirectional Representation and Line-Level Optimization

the DeepLineDP technique [9]. Since, according to their paper, DeepLineDP outperforms all previous techniques, it can be considered the state-of-the-art technique for line-level defect prediction. We use the replication package from the original authors and evaluate DeepLineDP against our dataset for comparison. We investigate whether Bugsplorer can outperform it in terms of classification performance and cost-effectiveness.

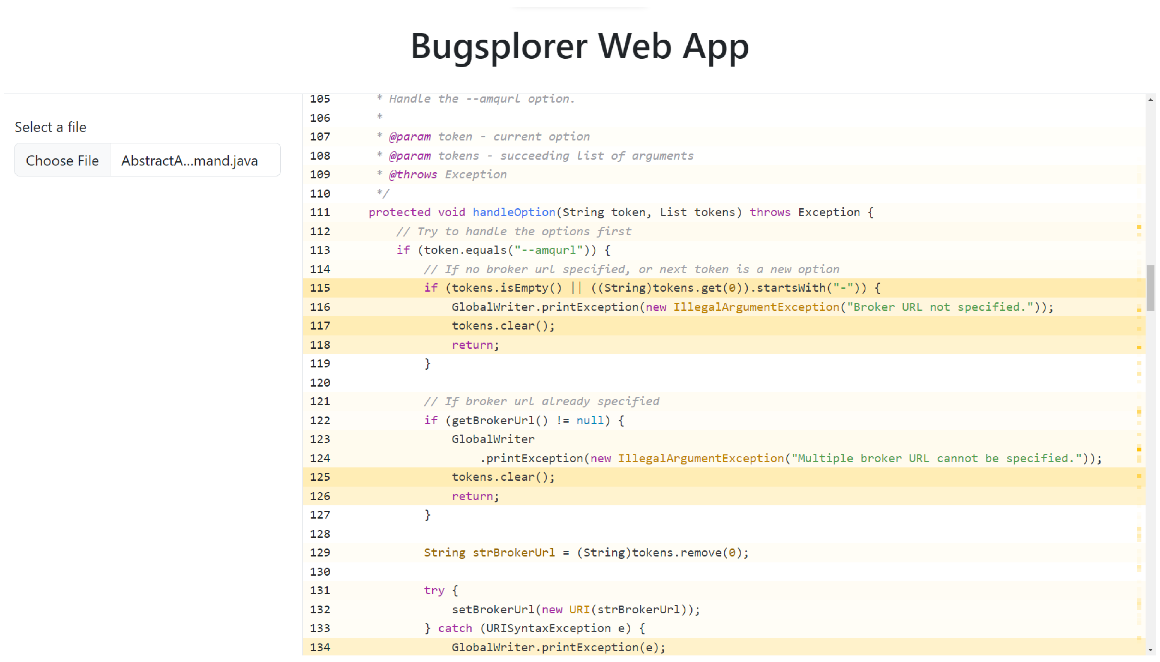
Table 3.5 compares the performance of Bugsplorer and DeepLineDP across different datasets. We see that Bugsplorer outperforms DeepLineDP across all datasets in all metrics. Fig. 3.7 compares their performance using boxplots. It further shows us that for any metric, the worst score of Bugsplorer is better than the best score of DeepLineDP. In the case of classification performance, Bugsplorer achieves 26-68% better performance for balanced accuracy and 31-80% better performance for AuROC. We also see a similar trend in cost-effectiveness. For the recall@20%LOC metric, Bugsplorer achieves 113-340% improved performance. For the effort@20%recall metric, Bugsplorer shows 72-81% improved performance, which indicates Bugsplorer can significantly reduce the effort needed to find defective lines in a codebase. Finally, for the initial false alarm metric, Bugsplorer shows 0-97% better performance. Thus, Bugsplorer has significantly better classification capability than the baseline while being more cost-effective at the same time.

Similar to Bugsplorer, DeepLineDP uses a hierarchical structure of neural networks. However, it uses two RNNs (inherently GRUs) to build the model, whereas Bugsplorer uses two transformer networks based on the RoBERTa architecture [90]. Due to a sequential architecture like RNN, DeepLineDP can represent a line only unidirectionally, either from left to right or right to left. Then it concatenates these two representations to make a bidirectional representation. On the contrary, Bugsplorer can directly make a bidirectional representation of a line via the Line Encoder (Section 3.3.3). Furthermore, during the training phase, Bugsplorer is optimized for line-level defect prediction, whereas DeepLineDP is optimized for file-level defect prediction. Both of these *novel contributions* (i.e., bidirectional representation and line-level optimization) are proven to be beneficial in RQ_2. Thus, Bugsploer's better performance over DeepLineDP is explainable.

Summary of RQ_4: Bugsplorer outperforms the state-of-the-art technique for line-level defect prediction. Bugsplorer is 26-68% more accurate in predicting the defective lines from source code. It can also reduce the effort in finding defective lines by 72-81%

3.5 Prototype

In this section, we discuss our web-based prototype for the Bugsplorer technique. It provides easy access to our technique to the end-users and provides various features to find defective lines. Fig. 3.8 shows the user interface of our tool. The prototype consists of three components, Web-based Front End, Application Server, and Defect Predictor, described below.



```
     * Handle the --amqurl option.
     *
     * @param token - current option
     * @param tokens - succeeding list of arguments
     * @throws Exception
     */
    protected void handleOption(String token, List tokens) throws Exception {
        // Try to handle the options first
        if (token.equals("--amqurl")) {
            // If no broker url specified, or next token is a new option
            if (tokens.isEmpty() || ((String)tokens.get(0)).startsWith("-")) {
                GlobalWriter.printException(new IllegalArgumentException("Broker URL not specified."));
                tokens.clear();
                return;
            }

            // If broker url already specified
            if (getBrokerUrl() != null) {
                GlobalWriter
                    .printException(new IllegalArgumentException("Multiple broker URL cannot be specified."));
                tokens.clear();
                return;
            }

            String strBrokerUrl = (String)tokens.remove(0);

            try {
                setBrokerUrl(new URI(strBrokerUrl));
            } catch (URISyntaxException e) {
                GlobalWriter.printException(e);
```

Figure 3.8: User Interface of the Prototype of Bugsplorer

3.5.1 Web-based Front End

The web-based front end serves as the user interface for Bugsplorer. It has a minimal design to help mitigate the learning curves of the users. It contains a file input, allowing users to select any Python or Java source code document. Upon selection, the prototype reads the source code document, identifies the programming language, and sends an HTTP request containing this information to the application server. The application server (described later) receives this request and returns a ranked list of the defective lines. Upon receiving, the front end identifies and highlights the most risky lines to the user. The intensity of the highlighting depends on how much vulnerable a particular source code line is. This helps the user to easily navigate to the most vulnerable part of the codebase and reduce the SQA cost along the process.

3.5.2 Application Server

The application server is implemented as a Facade [115] between two incompatible components – the web-based front end and the defect predictor. It receives the file content and the name of the programming language from the front end and captures the ranking of vulnerable lines from the defect predictor component (discussed later).

First, it performs the pre-processing on the source code as outlined in Section 3.3.1. During this pre-processing, the source code document is divided into lines, and each line is divided into tokens. Then, each token is encoded (i.e., mapped) to an integer value denoting that token. After the tokenization and encoding, it calls the appropriate defect predictor based on the provided programming language name. The defect predictor returns the probability of each source code line being defective. Based on these probability values, the application server generates the ranked list of vulnerable lines and returns them to the front end.

3.5.3 Defect Predictor

The defect predictor represents our Bugsplorer technique. It comes in two variants – Python and Java. The application server is responsible for calling the appropriate defect predictor based on the information provided by the front end. Both of these defect predictors work the same way, but their underlying models are trained on two different datasets. While the defect predictor for Python is trained on the Defectors [64] dataset, the defect predictor for Java is trained on the LineDP dataset [20]. As described in the Methodology (Section 3.3), it comprises a hierarchical structure of transformer networks containing Embedding, Line Encoder, and Line Classifier sub-components. Using these sub-components, the defect predictor analyzes the defect-proneness measure of each line from the source code document. These measures are then captured by the application server and sent to the front end of the prototype.

3.6 Threats To Validity

Threats to internal validity relate to experimental errors and biases. Re-implementation of the existing techniques could pose a threat. However, while implementing the DeepLineDP technique [9], we use the replication package provided by the authors. Possible errors in the implementation of our technique could also pose a threat. To avoid such errors, we carefully developed the technique with several rounds of revision followed by rigorous testing. Therefore, the threats to the internal validity posed by Bugsplorer might be minimal.

Threats to construct validity are factors that may affect how well a test or measure assesses what it is supposed to measure. We use five evaluation metrics to evaluate Bugsplorer in both classification and cost-effectiveness aspects. Given the severe class imbalance in datasets (less than 1% defective lines), we chose the metrics that are minimally affected by class imbalance. Furthermore, these metrics were also widely used by similar prior works [9], [20], [21]. The reliability of the datasets used is another threat since some bug-fix commits might make additional changes. To minimize the threat, we perform five levels of filtration suggested in the literature [93], [105] to ensure good quality of the commits. Since Bugsplorer only takes a single file as input, its capability of finding defects that span multiple files (e.g., incorrect API use) might pose a threat. However, Bugsplorer learns to predict defective lines based on previous mistakes. Thus, it could detect some of such defects if the training dataset contains similar instances. In other words, even though Bugsplorer accepts single-file input, it could identify defects related to external files. Nonetheless, we acknowledge that our technique might be limited in this regard.

Threats to external validity relate to the generalizability of our technique. We evaluate Bugsplorer using two datasets constructed from Python and Java software systems. These datasets contain 33 software systems in total. Furthermore, the software systems in the Python dataset – Defectors – are from 18 application domains and 24 organizations. Thus, our evaluation using these large and diverse datasets could mitigate the threats to external validity.

3.7 Manual Analysis

In this section, we perform a qualitative analysis to investigate in which scenarios Bugsplorer shines and in which it struggles. In particular, we categorized the predictions as false positives, false negatives, true positives, and true negatives. Then, we analyze 100 random samples from each category to find interesting patterns. We summarize our findings below.

False Positives

The most common pattern in this category is the use of long comments that look like code. In particular, more than half of our samples (52) have comments spanning three or more lines. Example 1 in Table 3.7 shows such a case where an IPython code example is added as a comment and spans eight lines. Embedding structural information with the source code [19], [113] might mitigate such issues. Another common pattern (Example 2) is the use of valid but rare syntax. Declaring a class within a class is a valid but rarely used Python syntax. Therefore, Bugsplorer might predict it as a defective line since it learns from common patterns in the training data. In the future, training the model with code examples from the official documentation of the programming language might mitigate the issue.

False Negatives

The most common pattern in the group is the code that depends on the environment. It is hard to know whether such code is defective or not just by looking at the code (a.k.a. extrinsic bug) [116]. Some common examples of such a pattern are reading environment variables, reading a file, or programs concerning operating systems or software versions (Example 3). Nearly one-fifth of our samples from this group (21) follow this pattern.

True Positives

An interesting finding is that Bugsplorer cannot only find bugs in various programming languages (e.g., Python or Java), but it also knows the common tools. For instance, Example 4 shows a git command that uses the `missing=print` option, which is added in version 2.22. Bugsplorer identifies the corresponding line as defective. The fixed version of that code[5] also reflects the issue. Another interesting finding is that Bugsplorer is precise in identifying blocks of defective lines (Example 5). Bugsplorer is good at identifying security vulnerabilities as well. Example 6 shows a case where the password is hardcoded, whereas it should be read from some configuration file. Bugsplorer was able to pinpoint the line containing this security vulnerability.

[5]https://bit.ly/3LnpaXj

Table 3.7: Examples of classification by Bugsplorer

Eg	Category	Code*

1	FP	`>>> from torch import Tensor`
		`>>> class ExampleModule(DeviceDtypeModuleMixin):`
		`...     def __init__(self, weight: Tensor): <---`
		`...         super().__init__()`
		`...         self.register_buffer('weight', weight)`
		`>>> _ = torch.manual_seed(0)`
		`>>> module = ExampleModule(torch.rand(3, 4))`
		`>>> module.weight #doctest: +ELLIPSIS`

2	FP	`class ApiKeyForm(forms.ModelForm):`
		`    allowed_origins = OriginsField(label=_('Allowed Domains'), required=False,`
		`    help_text=_('Separate multiple entries with a newline.'))`
		
		`    class Meta: <---`
		`        model = ApiKey`
		`    fields = ('label', 'scopes')`

3	FN	`    elif is_path:`
		`        if compat.PY2:`
		`            # Python 2`
		`            f = open(path_or_buf, mode) <---`
		`        elif encoding:`
		`            # Python 3 and encoding`
		`            f = open(path_or_buf, mode, encoding=encoding)`
		`        else:`
		`            # Python 3 and no explicit encoding`

4	TP	`# Now we need to find the missing filenames for the subpath we want.`
		`# Looking for this 'rev-list' command in the git --help? Hah.`
		`cmd = f"git -C {tmp_dir} rev-list --objects --all --missing=print -- {subpath}" <---`
		`ret = run_command(cmd, capture=True)`

5	TP	`try {`
		`  // delete done file`
		`  boolean deleted = operations.deleteFile(doneFileName); <---`
		`  log.trace("Done file: {} was deleted: {}", doneFileName, deleted); <---`
		`    if (!deleted) { <---`
		`    log.warn("Done file: " + doneFileName + " could not be deleted"); <---`
		`  }`
		`} catch (Exception e) {`
		`  handleException(e);`
		`}`

6	TP	`properties.setProperty("user","cloud");`
		`properties.setProperty("password","scape"); <---`

*A left arrow (`<---`) indicates the predicted buggy line.

True Negatives

Unfortunately, finding any pattern within this group is hard since it contains all the correctly classified defect-free code.

3.8 Related Work

In this section, we discuss the related work on defect prediction from the following four perspectives.

3.8.1 Defect Prediction at Different Levels of Granularity

Defect prediction has been a popular research topic for the last few decades. Defects can be predicted at different granularity levels such as module [15], [16], file [18], [117], method [19], [118], and commit [21], [63], [105], [106], [119]–[121]. Finding the actual lines of code that contain defects still consumes significant effort from developers. Two recent independent studies by Wan *et al.* [122] and Pornprasit *et al.* [9] showed that practitioners could benefit from fine-grained defect prediction such as line-level defect prediction. It can help developers focus their SQA efforts on the vulnerable parts of the source code.

3.8.2 Machine Learning Approaches for Defect Prediction

Machine learning approaches for defect prediction primarily rely on different metric scores to identify defective entities (e.g., file or commit). Kamei *et al.* [105] perform a large-scale study on change-level defect prediction using six open-source and five closed-source projects. They are the first to coin the phrase just-in-time (JIT) defect prediction. They proposed a total of 14 metric scores to predict defects at the file level with a logistic regression model. McIntosh *et al.* [106] conduct a time-series analysis on JIT defect prediction using two rapidly evolving projects. They extract 17 code properties and showed that the importance of these code properties in predicting the defective commits change over time. Jiang *et al.* [109] attempt to personalize defect prediction for different developers. They used bag-of-words and characteristics vector (i.e., count of each node type in AST) to predict the defects at the file level. Even though these works lay the ground for further defect prediction research, they are often limited by their coarse granularity and mediocre performance.

3.8.3 Deep Learning Approaches for Defect Prediction

Previous deep learning-based defect prediction models used various architectures to extract semantic and syntactic features from source code. Wang *et al.* [25] proposed a Deep Belief Network (DBN) architecture that represents a source code document using semantic features derived from the Abstract Syntax Tree (AST). Li et al. [26], [27] proposed a Convolutional Neural Network (CNN) architecture that learns the semantic and structural features of source code documents from the token sequences and the AST, Program Dependency Graph (PDG) and Data Flow Graph (DFG). Dam *et al.* [28] and Zou *et al.* [29] individually proposed a Long Short-term Memory (LSTM) architecture that can learn the semantic and syntactic features of source code documents from the token sequences and the CFG. However, these models only predicted defects at the file level, which is too coarse-grained. In contrast, our deep learning-based approach predicts defects at the line level and thus can identify defective lines of source code.

3.8.4 Line-Level Defect Prediction

Prior studies attempt to predict defects at the line level using various approaches, including static analysis [35]–[39]. However, static analysis tools produce too many false positive results [105] as well as false negative results [45]. In the last few years, line-level defect prediction with an explainable model has been popular. Wattanakriengkrai *et al.* [20] train a model to predict defects at the file level. Then, they attempt to explain the predictions of the model using LIME [34]. LIME provides importance values for each input (i.e., tokens), which, in turn, is used to find defective lines containing highly important tokens. Later, Pornprasit *et al.* [21] adapted this technique to identify defective lines from commit diffs. Recently, Pornprasit *et al.* [9] proposed DeepLineDP that trains a Gated Recurrent Unit (GRU) model [73] with attention mechanism [33] to predict defects at the file level. Then, they rank the lines with highly attended tokens as the candidate defective lines. The core limitation of these approaches is that their models learn with a file-level defect prediction objective. That is, these models originally learn to predict whether a file is defective or not. Therefore, their attention or importance values might not be targeted towards a buggy line and its surrounding lines but rather be scattered over the whole file. As a result, when these values are later used to predict defects at the line level, their performance could be sub-optimal. On the contrary, Bugsplorer directly learns to predict defects at the line level and thus can focus on a finer-grained context of each line.

3.9 Summary

To summarize, in this study, we propose a novel transformer-based technique to predict defects at the line level. Our evaluation with five performance metrics shows that Bugsplorer has a very good capability of predicting defective lines with 26-72% better accuracy than the state-of-the-art technique. It can rank the first 20% defective lines within the top 1-3% vulnerable lines of the codebase. Thus, Bugsplorer has the potential to significantly reduce SQA costs by ranking defective lines higher. While Bugsplorer identifies certain parts of the code as buggy, it could be less helpful without any meaningful explanation [23]. Thus, in Chapter 4, we propose a novel technique to generate natural language explanations for software defects by learning from a large corpus of defect-fixing commits. Our evaluation using three performance metrics shows that Bugsplainer can generate *understandable* and *good* explanations according to Google's standard and can outperform multiple baselines from the literature.

Chapter 4

Bugsplainer: Explaining Software Defects Leveraging Code Structures in Neural Machine Translation

Our first study in Chapter 3 proposes Bugsplorer that can predict defects at the line level accurately and cost-effectively. However, these predictions could be less helpful without any meaningful explanation about the defects [23]. In this study, we propose *Bugsplainer*, a transformer-based generative model that generates natural language explanations for software defects leveraging structural information and defective patterns from the source code. Our evaluation using three performance metrics shows that Bugsplainer can generate *understandable* and *good* explanations according to Google's standard and can outperform multiple baselines from the literature. We also conducted a developer study involving 20 participants where the explanations from Bugsplainer were found to be more accurate, more precise, more concise, and more useful than the baselines.

The rest of this chapter is organized as follows. Section 4.1 introduces the study by outlining current solution for the problem, addressing 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 defects leveraging code structure in neural machine translation. Section 4.4 discusses our experimental design, datasets, and evaluation results. Section 4.5 introduces a working prototype of our study. Section 4.6 identifies possible threats to the validity of our work. Section 4.7 discusses the existing studies related to our research. Finally, Section 4.8 summarizes this study.

4.1 Introduction

A software bug is an incorrect step, process, or data definition in a computer program that prevents the program from producing the correct result [1]. Bug resolution is one of the major tasks of software development and maintenance. According to several studies, it consumes up to 40% of the total budget [4] and costs the global economy billions of dollars each year [3], [24].

When a defect prediction tool predicts a software bug, the assigned developer attempts to understand the source code responsible for the bug and then corrects the code. Over the last five decades, there have been numerous approaches to automatically find the location of a bug [24], [123]. However, they often identify specific parts of the code as buggy without offering any meaningful explanation [23]. Developers are thus generally responsible for understanding a bug from the identified code before making any changes. Understanding bugs by looking at the code claims significant debugging time. Developers spend $\approx$ 50% of their time comprehending the code during software maintenance [10]. However, neither many studies attempt to explain the bugs in the source code to the developers, nor are they practical and scalable enough for industry-wide use [23], [24].

Explaining a bug in the software code is essential to fix the bug, but a highly challenging task. Many static analysis tools such as FindBugs [35], PMD, SonarLint, PyLint, and pyflakes [39] employ complex hand-crafted rules to detect the bugs and vulnerabilities in software code. They use pre-defined message templates to explain the bugs and vulnerabilities upon detection. Unfortunately, their utility could be limited due to their high false-positive results and the lack of actionable insights in their explanations [40]–[42]. In particular, their explanations are often too generic and unaware of the context due to their pre-defined, templated nature [44]. Thung *et al.* [45] also suggest that static analysis tools suffer from a large number of *false negative* results, which could leave the software systems vulnerable to bugs.

Unlike traditional, rule-based approaches (e.g., static analysis tools), explaining software bugs can be viewed as a translation task, where the buggy code is the source language and the corresponding explanation is the target language. In recent years, machine translation, especially neural machine translation (NMT) [31], has found numerous applications in several domains [79], [84]. NMT has also been used in different software engineering tasks including, but not limited to, code summarization [46], [47], code comment generation [124], [125], and commit message generation [53]–[56]. Traditional NMT models often consist of two items: *encoder* and *decoder*. The encoder first converts the words of the source language into an intermediate numeric representation. Then the decoder generates the target words one by one using the intermediate representation and previous words from the generated sequence [31]. However, explanation generation from the buggy source code using neural machine translation poses two major challenges.

Understanding the structures of source code: Natural language is loosely structured, which exhibits phenomena like ambiguity and word movement [31]. Word movement is the appearance of words in a sentence in different orders but still being grammatically correct. On the contrary, programming languages are more structured, syntactically restricted, and less ambiguous [60]. From the two programs having the same vocabulary, one could be buggy, and the other could be correct due to their structural differences (e.g., 4.3b, 4.3c). Thus, capturing and understanding code structures is essential to explaining the buggy code. Unfortunately, traditional NMT-based techniques often treat source code as a sequence of tokens and thus might fail to capture the structures of source code properly [61].

Understanding and detecting buggy code patterns: From a high-level perspective, NMT models translate words from the source language into words from the target language. However, to generate explanations from the buggy code, the model must accurately reason about the bug from the buggy code and its structures. Such reasoning is non-trivial and warrants the model to know buggy code patterns. Traditional NMT models might not be sufficient to tackle all these challenges due to their simplified assumptions about sequential inputs and outputs. According to Ray *et al.* [62], buggy code is less repetitive (a.k.a. *unnatural*) than regular code, which could exacerbate the above challenges.

In this study, we propose *Bugsplainer*, a novel transformer-based generative model that generates natural language explanations for software bugs by learning from a large corpus of bug-fix commits (i.e., commits that correct bugs). Our solution can address the above challenges, which makes our work *novel.* First, Bugsplainer can leverage code structures in explanation generation by applying structure-based traversal [47] to the buggy code. Second, we train Bugsplainer using both buggy source code and its corrected version, which helps the model to understand and detect buggy code patterns during its explanation generation for the buggy code.

We train and evaluate Bugsplainer with $\approx$ 150K bug-fix commits collected from

GitHub using three different metrics – BLEU [65], Semantic Similarity [66] and Exact Match. We find that the explanations from Bugsplainer are *understandable* and *good*. We compare our technique with four appropriate baselines – pyflakes [39], CommitGen [53], NNGen [56], and Fine-tuned CodeT5 [67]. Bugsplainer outperforms all four baselines in all metrics by a statistically significant margin. One major strength of Bugsplainer is understanding the structure of the code and buggy code patterns, where the baselines might be falling short. To further evaluate our work, we conducted a developer study involving 20 developers from six countries, where the identities of both our tool and the baselines were kept hidden. The study result shows that explanations from Bugsplainer are more accurate, more precise, more concise, and more useful compared to that of the baselines.

We thus make the following contributions in this study:

(a) A novel transformer-based technique, *Bugsplainer*, that can explain software bugs by leveraging the structural information and buggy code patterns from source code.

(b) A novel pre-training technique, namely – Discriminatory Pre-training, that is shown to be effective in generating better explanations.

(c) A benchmark dataset containing $\approx$ 150K instances of buggy code, corrected code, and corresponding explanations written by human developers. To the best of our knowledge, this is the first benchmark of its kind.

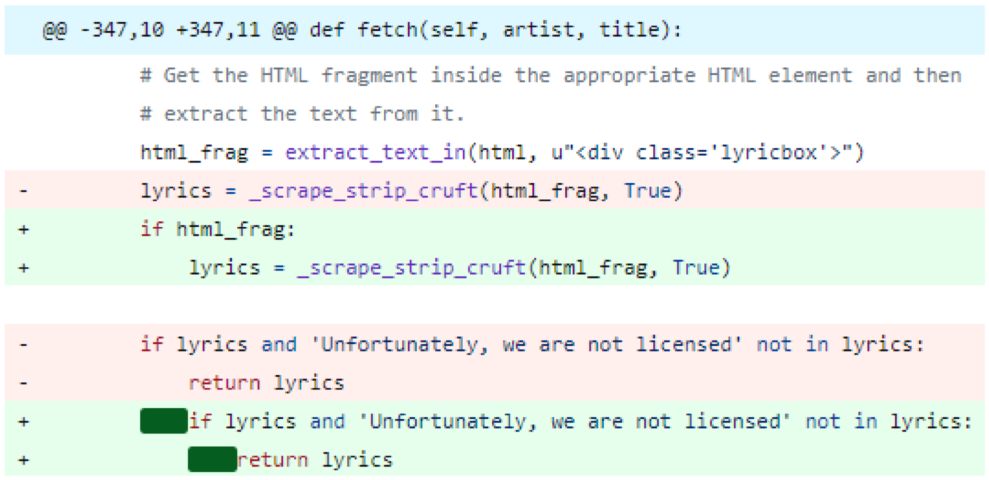
(d) A comprehensive evaluation and validation of the Bugsplainer technique using both popular performance metrics (e.g., BLEU score) and a developer study.

(e) A replication package that includes our working prototype, experimental dataset, and other configuration details for the replication or third-party reuse[1].

4.2 Motivating Example

To demonstrate the capability of our technique – *Bugsplainer*, let us consider the example in Fig. 4.1. The code snippet is taken from `beetbox/beets` repository at

[1]https://bit.ly/3H7R1aI



```
@@ -347,10 +347,11 @@ def fetch(self, artist, title):
           # Get the HTML fragment inside the appropriate HTML element and then
           # extract the text from it.
           html_frag = extract_text_in(html, u"<div class='lyricbox'>")
-          lyrics = _scrape_strip_cruft(html_frag, True)
+          if html_frag:
+              lyrics = _scrape_strip_cruft(html_frag, True)

-          if lyrics and 'Unfortunately, we are not licensed' not in lyrics:
-              return lyrics
+              if lyrics and 'Unfortunately, we are not licensed' not in lyrics:
+                  return lyrics
```

Figure 4.1: A motivating example for Bugsplainer

Table 4.1: Generated explanations for buggy code

Technique	**Generated Explanation**
Ground Truth	Fix a bug where the lyricswiki fetcher would try to unescape an empty (None) response and crash
CommitGen [53]	Small bug fix for error handling
NNGen [56]	fix UnicodeDecodeError with non-ASCII text
Fine-tunedCodeT5 [67]	Don't try to get lyrics if we are licensed
pyflakes [39]	*no error found*
Bugsplainer	fix crash when lyrics not found

GitHub[2]. The buggy code attempts to scrape the lyrics of a song from an HTML fragment. However, if the HTML fragment is empty, then the program crashes. Table 4.1 shows both developer's explanation for the buggy code (a.k.a., reference) and the explanations generated by different techniques, including Bugsplainer. We see that the explanations generated by CommitGen [53] (i.e., RNN-based technique) and NNGen [56] (i.e., Information Retrieval-based technique) are not helpful. On the other hand, the explanation from Fine-tuned CodeT5 [67] is not accurate as the bug has nothing to do with licensing. Pyflakes, a static analysis-based technique, does not provide any explanation since it could not detect the bug using its pre-defined rules. On the other hand, the explanation generated by our technique, Bugsplainer, is *accurate* as it expresses the same information as the ground truth and *precise* as it

[2]https://bit.ly/3PGnkzK

expresses no unnecessary information. Moreover, we see that in the fixed version of the code (Fig. 4.1), an `if` condition was used to check whether the HTML fragment exists (i.e., lyrics were found) or not, which reflects the solution implied by our explanation.

4.3 Methodology

Fig. 4.2 shows the schematic diagram of our proposed technique – *Bugsplainer* – for explaining software defects in natural language. We discuss different steps of our technique in detail as follows.

4.3.1 Extract Buggy and Bug-free AST Nodes from Commit

First, we construct Abstract Syntax Trees (ASTs) of both buggy and bug-free code using the information from a bug-fix commit. A bug-fix commit contains the bug-free version of the code while being connected to its parent commit containing the buggy version. From these two versions of the source code, we construct two different ASTs – the buggy AST (Step 1a, Fig. 4.2) and the bug-free AST (Step 1b, Fig. 4.2). A commit also references removed lines (i.e., buggy lines) and added lines (i.e., bug-fix lines). Using these line numbers, we extract the buggy nodes from the buggy AST (Step 2a, Fig. 4.2) and bug-free nodes from the bug-free AST (Step 2b, Fig. 4.2). If a multi-line expression touches these line numbers, we extract the whole expression node (Line 11, Algorithm 1). Besides the affected lines, the contextual information (e.g., surrounding lines) often provides useful clues about why the code was changed. Asaduzzaman *et al.* [126] suggest that three lines of code around a target line might be sufficient to capture the contextual information. While extracting the buggy and bug-free nodes, we thus also extract the nodes representing three lines above and below the changed lines in the code.

4.3.2 Generate diffSBT Sequence

In this step, we convert the buggy and bug-free AST nodes into diffSBT sequences (i.e., preserve structural information) using the diffSBT algorithm (Step 3, Fig. 4.2).

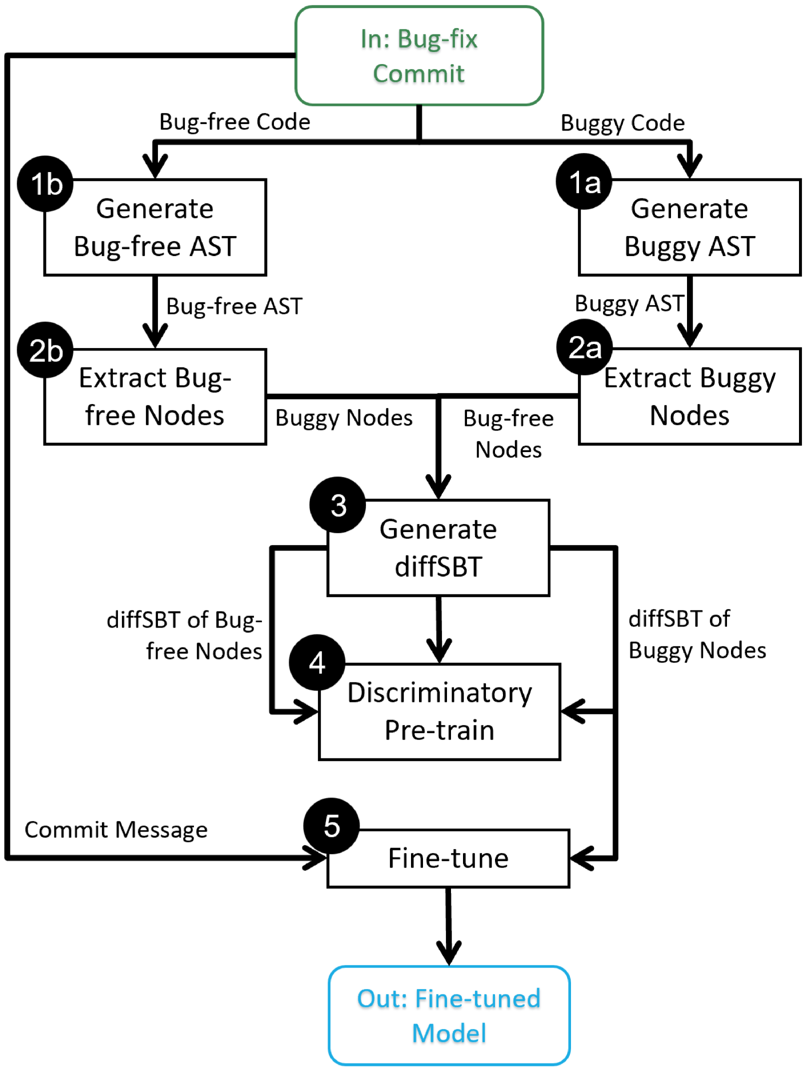


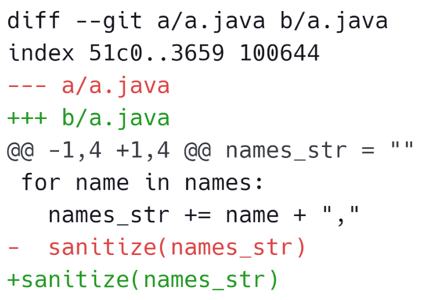
Figure 4.2: Schematic diagram of Bugsplainer

Algorithm [1] shows our algorithm – *diffSBT* – for structure-preserving sequence generation from commit diff, which is an adaptation of SBT algorithm by Hu *et al.* [47]. We create two versions of the diffSBT sequence. One of them contains both buggy and bug-free nodes to aid the discriminatory pre-training (see Section 4.3.3). The other contains only buggy nodes to aid the fine-tuning.

To illustrate the generation of diffSBT sequences from a commit diff, let us consider Fig. 4.3. Fig. 4.3a contains a bug-fix commit. The source code before submitting this commit was buggy (Fig. 4.3b), and the source code after the commit is bug-free (Fig. 4.3c). The bug is that the `sanitize()` function was called inside the `for` loop rather than outside the `for` loop. The buggy code resembles the fixed version

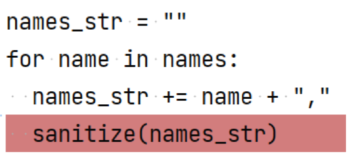
Algorithm 1 Generate diffSBT sequence from commit diff

```
1: function diffSBT(c)                                ▷ Generate diffSBT sequence for commit
2:     buggyAST ← BuildAST(c.buggyCode)
3:     bugfreeAST ← BuildAst(c.bugFreeCode)
4:     buggyNodes ← Intersections(buggyAST, c.removed)
5:     bugfreeNodes ← Intersections(bugfreeAST, c.added)
6:     return SBT(buggyNodes) + $\langle /s \rangle$ + SBT(bugfreeNodes)
7: end function
8: function Intersections(r, ln)
9:     nodes ← $\phi$                                  ▷ Initialize nodes with an empty list
10:    for all n in r do                               ▷ Get intersections for all nodes in $r$
11:        if IsInside(n, ln) or IsExpression(n) then
12:            Append(nodes, n)
13:        else if StartsInside(n, ln) then     ▷ Keep $n$ but prune the children outside $ln$
14:            n.children ← IntersectingChildren(n, ln)
15:            Append(nodes, n)
16:        else if EndsInside(n, ln) then ▷ Node $n$ starts before the $ln$. Return only the
   children of $n$ that intersect with $ln$.
17:            children ← IntersectingChildren(n, ln)
18:            Append(nodes, children)
19:        end if
20:    end for
21:    return nodes
22: end function
23: function IntersectingChildren(r, ln)
24:    children ← $\phi$
25:    for all n in r.children do
26:        if HasIntersection(n, ln) then
27:            node ← Intersections(ch, ln)
28:            Append(children, node)
29:        end if
30:    end for
31:    return children
32: end function
```



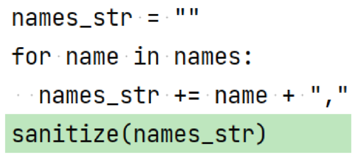
```
diff --git a/a.java b/a.java
index 51c0..3659 100644
--- a/a.java
+++ b/a.java
@@ -1,4 +1,4 @@ names_str = ""
 for name in names:
   names_str += name + ","
-  sanitize(names_str)
+sanitize(names_str)
```

(a) Bug-fix commit diff



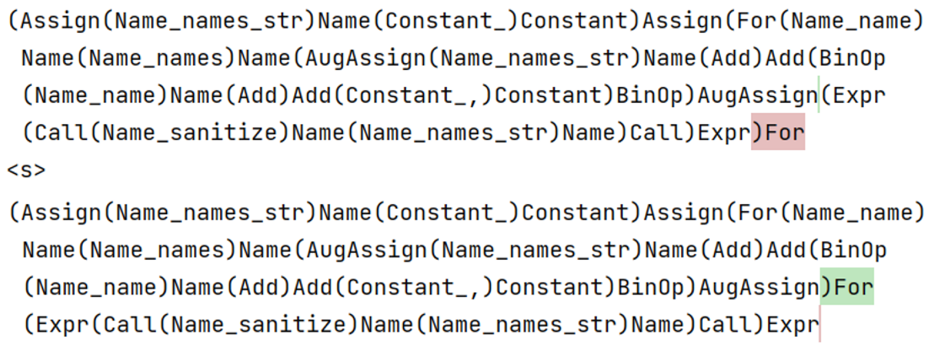
```
names_str = ""
for name in names:
  names_str += name + ","
  sanitize(names_str)
```

(b) Buggy code



```
names_str = ""
for name in names:
  names_str += name + ","
sanitize(names_str)
```

(c) Bug-free code



```
(Assign(Name_names_str)Name(Constant_)Constant)Assign(For(Name_name)
 Name(Name_names)Name(AugAssign(Name_names_str)Name(Add)Add(BinOp
 (Name_name)Name(Add)Add(Constant_,)Constant)BinOp)AugAssign(Expr
 (Call(Name_sanitize)Name(Name_names_str)Name)Call)Expr)For
<s>
(Assign(Name_names_str)Name(Constant_)Constant)Assign(For(Name_name)
 Name(Name_names)Name(AugAssign(Name_names_str)Name(Add)Add(BinOp
 (Name_name)Name(Add)Add(Constant_,)Constant)BinOp)AugAssign)For
 (Expr(Call(Name_sanitize)Name(Name_names_str)Name)Call)Expr
```

(d) diffSBT sequence for the buggy and bug-free code

Figure 4.3: An example of diffSBT sequence generation from commit diff

of the code as both code segments also have the same vocabulary. However, they differ by white spaces, as shown in the commit diff (Fig. 4.3a), which is a scope-related problem according to Python programming language. Thus, if the source code is considered a sequence of tokens without the structural information (as many studies [53], [56] do), the bug is hard to understand. Fig. 4.3d shows the diffSBT sequence for this bug-fix change. In the buggy version, we see that the `For` block closes at the end of the diffSBT sequence. On the contrary, in the bug-free version, the `For` block closes before the `Expr` block. Such placement ensures that the `Expr` block (i.e., `sanitize(name_str)`) is outside the `For` block. Thus, with the help of diffSBT, Bugsplainer can identify the difference in the structure of code, which could be useful in explaining the bug.

4.3.3 Train Bugsplainer

In Fig. 4.2, Steps 4-5 explain the training of Bugsplainer. Our training phase is divided into two steps – discriminatory pre-training and fine-tuning. In both steps, we use a RoBERTa tokenizer [90], pre-trained on GitHub CodeSearchNet dataset [112]. Due to its pre-train dataset (CodeSearchNet), this RoBERTa tokenizer has common code elements in its vocabulary, which can reduce the length of tokenized code sequence by 30%-45% [67]. We use this tokenizer to tokenize and encode the inputs (e.g., diffSBT sequence) and decode the outputs (e.g., commit message). In the following sections, we describe the training phase in detail.

Discriminatory Pre-training

Pre-trained language models have been found to be effective in improving many natural language understanding tasks (e.g., news title generation, question-answering) [79], [84]. During pre-training, a model acquires a general knowledge about a domain which allows it to *understand* the input (e.g., text, image) [84]. In natural language processing, pre-training is often performed in an unsupervised fashion (e.g., Word2Vec [127], missing token prediction [79]). However, many domains also use supervised pre-training (e.g., Multi-Task Learning [67], [84]). Bugsplainer uses both unsupervised and supervised pre-training to equip the model with a comprehensive understanding of the programming language and its bugs.

We use a pre-trained model – CodeT5 [67] – to perform our discriminatory pre-training with buggy and bug-free code. CodeT5 is a transformer model based on the Text to Text Transfer Transformer (T5) architecture [30], [84]. It has two versions – 60M parameters and 220M parameters. We use the 60M parameter version for Bugsplainer, which is pre-trained on GitHub CodeSearchNet data [112] for three unsupervised tasks. CodeSearchNet contains $\approx$ 6M methods written in popular programming languages accompanied by natural language documentation. Thus, the CodeT5 model has a significant understanding of programming and natural languages, making it an ideal choice for our pre-training task.

The pre-training with the CodeSearchNet dataset provides the model with general knowledge about programming and language syntax. However, to reason about a bug in the source code, the model should be able to differentiate between buggy and bug-free code. To equip the model with such a reasoning capability, we use diffSBT sequences of both buggy and bug-free AST nodes (Step 4, Fig. 4.2). We pre-train the Bugsplainer model to predict commit messages from the diffSBT sequences of the buggy and bug-free code. We refer to this pre-training step as *discriminatory pre-training* since Bugsplainer learns to discriminate between buggy and bug-free code. The diffSBT sequences for the buggy and bug-free code are separated by a special token (e.g., `</s>`). We hypothesize that the model can differentiate and attend to (i.e., selectively focus on) the changes in both sides of the separator token and generate the commit message (a.k.a. bug explanation) accordingly. Our experimental result reports the effectiveness of discriminatory pre-training in explaining software bugs (see Section 4.4.3).

Fine-tuning

Once the discriminatory pre-training is complete, we also train Bugsplainer to generate explanations from only buggy code. We take diffSBT sequences of only buggy code as the input and corresponding explanation (i.e., commit message) as the output. We pass both input and output to the RoBERTa tokenizer. Then, we fine-tune our pre-trained model from the previous phase to generate explanations from the diffSBT sequence of buggy code (Step 5, Fig. 4.2). The output of the fine-tuning step is the Bugsplainer model for bug explanation generation.

4.3.4 Generate Explanation

Once the training phase is complete, we test our model using the testing instances. Fig. 4.4 shows how Bugsplainer generates an explanation from buggy code. During the generation phase, Bugsplainer takes two inputs – the buggy code and the line numbers within the code that need an explanation. From the buggy code, Bugsplainer constructs the AST (Step 1, Fig. 4.4) and extracts the AST nodes that intersect with the given line numbers (Step 2, Fig. 4.4). Subsequently, Bugsplainer converts the intersecting nodes into a diffSBT sequence (Step 3, Fig. 4.4). Then, it tokenizes the diffSBT sequence using the same RoBERTa tokenizer and passes the tokens to the fine-tuned model (Step 4, Fig. 4.4). Finally, the fine-tuned model generates an explanation for the buggy code.

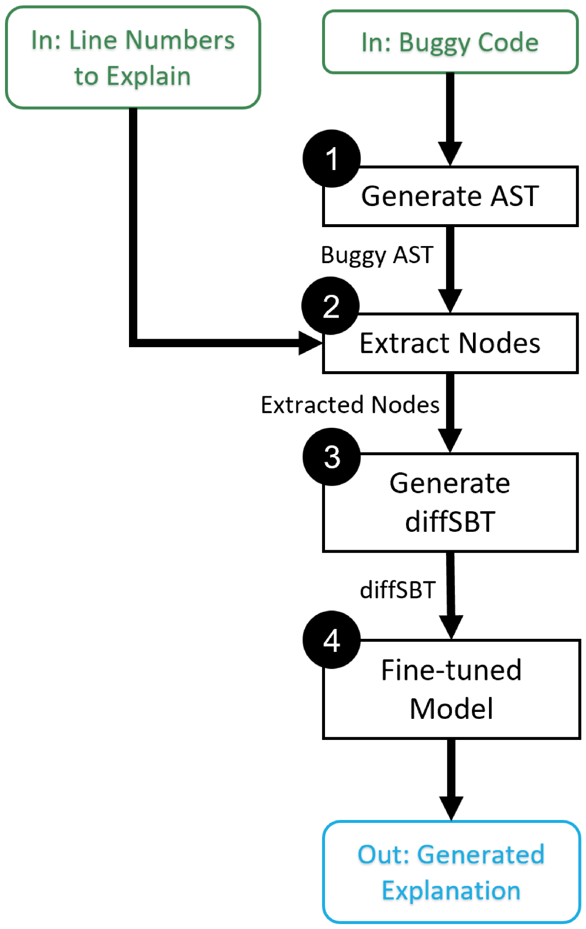


Figure 4.4: Explanation generation for buggy code

4.4 Experiment

We curate a large dataset of ≈ 150K bug-fix commits and evaluate Bugsplainer using three appropriate metrics from the relevant literature – BLEU score [65], Semantic Similarity [66], and Exact Match. To place our work in the literature, we compare our solution – Bugsplainer – with four relevant baselines. We also conduct a developer study to assess the quality of our automatically generated explanations (e.g., accuracy, usefulness) for software bugs. In our experiments, we thus answer four research questions as follows.

- **RQ$_1$**: How does Bugsplainer perform in explaining software bugs in terms of automatic evaluation metrics?
- **RQ$_2$**: How do (a) structural information and (b) discriminatory pre-training influence the performance of Bugsplainer in generating explanations for software bugs?

- **RQ$_3$**: Can Bugsplainer outperform the existing baseline techniques in terms of automatic evaluation metrics?

- **RQ$_4$**: How accurate, precise, concise, and useful are the explanations of Bugsplainer compared to baselines?

4.4.1 Dataset Construction

To conduct our experiments, we curate a dataset of ≈ 150K bug-fix commits from GitHub[3] using its REST API[4]. We discuss different steps of our dataset construction process as follows.

Repository Selection

First, we aim to find $\approx 10K$ Python repositories with high star counts to ensure high-quality commits. We choose Python since it is the second most popular programming language according to StackOverflow survey 2021[5] and the most wanted programming language to work with[6]. As GitHub's search API does not return more than $1K$ results from a single query, we use small buckets of star counts to renew our query contents. We found the 10,000th repository falls in the bucket of 300-399 stars. Thus, we collect all the repositories with a star count of $\geq$ 300, leading us to a total of 10,154 repositories.

Collection of Bug-fix Commits

We collected all the commits from the above repositories, which led to a total of ≈ 11.8M commits. Then, we attempt to find the bug-fix commits from them. Similar to previous studies [58], [128], we consider a commit as a *bug-fix* commit if it contains either 'fix' or 'solve' in its commit message. This filtration step led us to ≈ 1.4M bug-fix commits.

[3]Accessed: April 18, 2022
[4]https://docs.github.com/en/rest
[5]https://bit.ly/3cmooLv
[6]https://bit.ly/3PSFhLv

Filtration of Noisy Commits

To ensure the quality of the collected commits, we manually analyzed randomly sampled 500 commits from the above commit collection. We found seven machine-generated templates in the commit messages (e.g., "Merge branch X to master") that can be easily detected using appropriate regular expressions (see replication package for details). We remove these machine-generated templates from commit messages. If a commit message contains only machine-generated texts, then the whole commit is discarded from the dataset. We also note that Python repositories contain non-Python files (e.g., configuration files) and test scripts in their commits, which are out of the scope of this work. We thus keep the commits with at least one modified Python file (excluding test scripts).

Vaswani *et al.* [30] report that the complexity of transformer models increases quadratically with the length of input and output sequences. Therefore, we limit the maximum length of both commit diff and commit message. In particular, we retain such commits with $\leq$30 tokens in their commit message and $\leq$170 tokens in their commit diff. These limits cover >85% of both commit messages and commit diffs from the $\approx$ 1.4M bug-fix commits. Then, we remove commits with less than five tokens in their messages to discard trivial commits. We also keep only the commits with one diff hunk (i.e., change location) to avoid tangled commits (i.e., commits doing more than one task). After performing all these noise filtration steps, we end up with $\approx$ 180K bug-fix commits.

To determine the reliability of our constructed dataset, we performed a manual analysis using 385 commits. We randomly sample these commits from $\approx$ 180K commits above with a 95% confidence level and 5% error margin. We find that 92.1% of these commits are bug-fix and 5.2% are style-fix, indicating a negligible amount of noise (i.e., 2.7%) in our constructed dataset. Previous studies [58], [128] also use datasets with similar amounts of noise. Furthermore, manually filtering $\approx$ 180K commits was prohibitively costly or impractical, which possibly justifies our choice of using the current version of the dataset.

Embedding Structural Information

We generate a diffSBT sequence for each commit as described in Section 4.3.2. We first generate AST for both the buggy and bug-free code from the commit using the `ast` parser of Python 3[7]. After discarding syntactically incorrect programs, we find $\approx$ 150K diffSBT sequences.

Construction of Training and Testing Data

First, we randomly select 110K entries as the training dataset for both pre-training and fine-tuning steps. Second, we randomly split the remaining 40K entries into four sets that are allocated for validation and testing in both pre-training and fine-tuning steps. Thus, the training data are shared by both pre-training and fine-tuning steps, whereas the validation and testing data are not shared. Finally, we remove the part of the diffSBT sequence corresponding to the fixed version of the source code from the three fine-tuning splits (training, validation, and testing) since Bugsplainer aims to generate an explanation from the buggy code only.

4.4.2 Evaluation Metrics

To evaluate the explanations generated by Bugsplainer, we use three different metrics – BLEU score [65], Semantic Similarity [66], and Exact Match. Relevant studies [30], [67], [78], [84] frequently used these metrics, which justifies our choice. They are defined as follows.

Bi-lingual Evaluation of Understudy (BLEU)

BLEU score [65] is a widely used performance measure for NMT. It has been used in software engineering context as well [47], [53]–[56], [61]. It calculates the similarity between auto-generated and reference sequences in terms of their n-grams precision as follows.

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

Here, p_n is the ratio of overlapping n-grams from both generated and reference sequences and the total number of n-grams in the generated sentence, and w_n is the

[7]https://docs.python.org/3/library/ast.html

weight of the n-gram length. Following existing studies [47], [54], we use $N = 4$ and $w_n = 0.25$ for all $n \in [1, N]$. In other words, we compute the mean BLEU score for all n-gram lengths. The brevity penalty, BP, lowers the BLEU score if the candidate translation is too small. BP is defined as

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases} \tag{4.2}$$

Here, c is the length of the candidate translation, and r is the length of the reference sequence.

There exist several variations of the BLEU score. In our study, we use case-insensitive BLEU score with *add one smoothing* [129], which aligns the most with human judgement [54] among other variants of BLEU score.

Semantic Similarity

Although the BLEU score is widely adopted for evaluating machine translation, it does not take the meaning of the text into account. Haque *et al.* [66] conduct a human study to determine which metric better represents the perception of human evaluators. They find that Sentence-BERT encoder [32] with cosine similarity has the highest correlation with the human-evaluated similarity. Sentence-BERT provides a fixed-length numeric representation for any given text. As suggested by Haque *et al.* [66], we use `stsb-roberta-large`[8] pre-trained model to generate the embedding for the input text. We compute the Semantic Similarity as follows.

$$SemSim(ref, gen) = cos(sbert(ref), sbert(gen)) \tag{4.3}$$

where $sbert(x)$ is the numerical representation from Sentence-BERT for any input text x, ref is the reference explanation, and gen is the generated explanation.

Exact Match

We also use the Exact Match metric to evaluate our explanation. As the name suggests, Exact Match checks whether a generated explanation exactly matches the corresponding reference explanation. It is analogous to string equality checks in many programming languages, which are case-sensitive and space-sensitive.

[8]https://bit.ly/3dR9mxD

4.4.3 Evaluating Bugsplainer

Answering $\mathbf{RQ_1}$ – Performance of Bugsplainer

Table 4.2 shows the performance of Bugsplainer in terms of BLEU score, Semantic Similarity, and Exact Match. We run Bugsplainer five times with random initialization and report the average performance. The replication package contains the detailed results from our five runs.

When the dataset is split randomly into training, validation, and testing sets, Bugsplainer achieves a BLEU score of 33.15, which is considered as *understandable* to *good* translation according to Google's AutoML Translation documentation[9]. Explanations from Bugsplainer also have an average of 55.76% Semantic Similarity, which indicates a major semantic overlap with the explanations from developers. Finally, 22.37% of the explanations exactly match the reference explanations. To achieve an Exact Match with the reference, an NMT model warrants a substantial knowledge of the domain. All these statistics are highly promising and demonstrate the high potential of our technique in explaining software bugs.

Allamanis [130] report that overlap between training and testing datasets might lead to overestimating performance measurement. In our experiment design, we ensure no overlap between our training and testing datasets (see Section 4.4.1). However, we also use a pre-trained CodeT5 [67] model, which is pre-trained on millions of code snippets from thousands of repositories in CodeSearchNet dataset [112]. As a result, there might be an unavoidable overlap between pre-training and testing datasets. To ensure a fair evaluation, we thus discard the testing instances from CodeSearchNet repositories ($\approx$ 14% instances) to avoid any possibility of overlap and re-evaluate Bugsplainer. Table 4.2 shows that after discarding the overlapping repositories, Bugsplainer demonstrates a marginal improvement both in BLEU score and Semantic Similarity.

In the real world, when adopting Bugsplainer for a new project, data from the new project might not always be available to re-train Bugsplainer. Therefore, we investigate how the performance of Bugsplainer varies in a cross-project setting. That

[9]https://bit.ly/3wGpCIx

Table 4.2: Performance of Bugsplainer

Model	Dataset	BLEU	Semantic Similarity	Exact Match
Bugsplainer	Random split	33.15	55.76	22.37
	No CodeSearchNet Repository	34.53	56.67	19.55
	Cross-project	17.16	44.98	07.15
Bugsplainer 220M	Random split	33.87	56.35	23.50
	No CodeSearchNet Repository	35.59	57.29	20.74
	Cross-project	23.83	49.00	15.47

is, each of the training, validation, and testing datasets contains commits from mutually exclusive projects. From Table 4.2, we see that even though the performance of Bugsplainer decreases in a cross-project setting, it is still promising, especially in terms of the Semantic Similarity metric. We see that the BLEU score decreases by 48% whereas the Semantic Similarity decreases by only 19%. That is, in the cross-project setting, the generated explanations might express similar information but with different words. To verify the case, we manually compare a sample of 385 explanations from Bugsplainer (95% confidence level and 5% error margin) with the reference explanations. We find that a substantial amount of generated explanations express information either more precisely or with different phrases, which might cause the BLEU score to be low. For instance, for a particular bug[10], Bugsplainer generates *"Improve the message in IncompleteRead.__init__"*, whereas the reference is *"fixing incorrect message for IncompleteRead"*. Even though the generated explanation is accurate and more precise, it returns a BLEU score of only 11. Such a phenomenon also explains the low BLEU score and comparatively high Semantic Similarity score for the cross-project setting of Bugsplainer.

Recent studies suggest that increasing the model size can significantly improve the performance of deep learning models [67], [84], [90]. We thus were interested to see how the performance of Bugsplainer changes with an increased number of parameters. For this experiment, we train a 220M parameter variant of Bugsplainer

[10]https://bit.ly/3RoSpsT

Table 4.3: Performance of Bugsplainer by input length

#Words	**BLEU**	**Semantic Similarity**	**Exact Match**
$\# < 50$	32.05	54.90	17.84
$50 \leq \# < 100$	34.22	56.25	18.65
$100 \leq \# < 150$	34.72	56.99	21.10
$150 \leq \# < 200$	32.92	56.50	23.91

Table 4.4: Performance of Bugsplainer by the length of ground truth

#Words	**BLEU**	**Semantic Similarity**	**Exact Match**
$\# < 10$	35.75	56.32	22.72
$10 \leq \# < 20$	27.70	53.16	8.93
$20 \leq \# < 30$	21.62	52.03	1.26

and call it *Bugsplainer 220M*. Both variants share the same architecture (i.e., T5) but have different hyperparameters. The detailed hyperparameter values can be found in the replication package. Table 4.2 also shows the performance of Bugsplainer 220M in the random split and cross-project settings. We see improved performance in both cases, which aligns with the existing findings [67], [84]. Interestingly, in cross-project settings, Bugsplainer 220M achieves a big bump of $\approx 39\%$ in BLEU score and $\approx 117\%$ in Exact Match. Such a finding suggests that Bugsplainer 220M can generalize the acquired knowledge better across multiple projects than Bugsplainer.

Finally, we investigate how the performance of Bugsplainer is affected by the input and output length. Table 4.3 shows the metric scores categorized by the number of words in the input buggy code segments. The table shows no clear correlation between the input length and the performance. With increasing input length, the performance both increases and decreases. On the contrary, Table 4.4 shows that the performance of Bugsplainer tends to decrease with increasing ground truth lengths. Interestingly, the drop in Semantic Similarity is not as substantial as the BLEU score or Exact Match score. This suggests that even with increasing output length, Bugsplainer can provide semantically coherent explanations with the ground truth.

Summary of $\mathbf{RQ_1}$: Bugsplainer can generate bug explanations that are *understandable* and *good* according to Google's standard. It shows promising results not only in random split settings but also in cross-project settings. With a higher number of parameters, Bugsplainer can better generalize the acquired knowledge across multiple projects.

Table 4.5: Role of structural information and discriminatory pre-training

Model	BLEU	Semantic Similarity	Exact Match
Bugsplainer	**33.15**	**55.76**	**22.37**
Bugsplainer without structural information	30.78	53.74	15.42
Bugsplainer without discriminatory pre-training	30.32	53.51	16.62

Answering $\mathbf{RQ_2}$ – Role of structural information and discriminatory pre-training in Bugsplainer

In this experiment, we analyze the impact of structural information and discriminatory pre-training on bug explanation generation. We remove one of these two components from Bugsplainer and keep the rest as is. Such an experiment helps us understand the contribution of individual components toward Bugsplainer. Table 4.5 shows the performance of Bugsplainer for these two particular cases.

To analyze the impact of structural information, we use raw commit diff as input rather than diffSBT sequences. In the pre-train dataset, we keep the commit diff as is, while in the fine-tuning dataset, we remove the added lines (i.e., bug-free lines) from the commit diff. From Table 4.5, we see that the BLEU score of Bugsplainer reduces by 7.15% due to the absence of structural information. Interestingly, the Exact Match score also drops by 31.07%, which is significant. To analyze the impact of discriminatory pre-training, we use only fine-tuning dataset and avoid the pre-training step. In this experiment, we use the diffSBT sequences as input during the fine-tuning step. From Table 4.5, we see that the BLEU score of Bugsplainer reduces by 8.54% due to the absence of discriminatory pre-training. Interestingly, the Exact

Match score also drops by 25.70%, which is significant.

The significant performance drops due to the absence of structural information and discriminatory pre-training indicate their essential roles in Bugsplainer. Our technique also performs best when both items are incorporated.

Summary of $\mathbf{RQ_2}$: Both structural information and discriminatory pre-training have a major contribution to the performance of Bugsplainer. Furthermore, they are the most effective when they are used together.

Answering $\mathbf{RQ_3}$ – Comparison with existing baseline techniques

In this research question, we compare Bugsplainer with existing techniques from the literature and investigate whether Bugsplainer can outperform them in terms of various evaluation metrics. To the best of our knowledge, there exists no work that explains software bugs in natural language texts. Since the generation of commit messages is similar to explanation generation, we use state-of-the-art commit message generation techniques as our baseline. The main difference between commit message generation and explanation generation is the former takes both buggy and bug-free lines as input, whereas the latter takes only buggy lines as input. In particular, we compare Bugsplainer with three commit message generation techniques, namely – *CommitGen* [53], *NNGen* [56], and *Fine-tuned CodeT5* [67] and a static analysis tool *pyflakes* [39]. None of these existing approaches for commit message generation learns to differentiate between buggy and bug-free code. Thus, our approach has a better chance of generating meaningful explanations for the buggy code. We do not use another state-of-the-art NMT-based technique, ATOM [55], since they do not publish the code for a crucial part of their work due to commercial reasons [54]. As a result, ATOM cannot be evaluated using new datasets.

To generate error messages from *pyflakes*, a static analysis tool, we run pyflakes on the whole buggy source code. Once we have the error messages, we keep only the messages generated for the buggy lines. If we get multiple errors for the same data point, we keep them all and report the one with the highest automatic metric score (e.g., BLEU score).

CommitGen uses an NMT framework, namely nematus [131], which we use to

Table 4.6: Comparison of Bugsplainer with existing baseline techniques (Using five random runs)

Technique	**BLEU**	**Semantic Similarity**	**Exact Match**
pyflakes	0.49	5.68	0.00
CommitGen	9.94	35.39	1.04
NNGen	24.16	47.33	14.17
Fine-tuned CodeT5	26.19	54.52	8.85
Bugsplainer*	**32.90**	**55.22**	**18.14**

*The scores differ from the earlier tables due to five random runs

replicate the technique. The authors also provide the values of all the important parameters in their paper, which were carefully adopted in our replication.

According to a recent study [54], NNGen [56] is the state-of-the-art tool for generating commit messages. Being an IR-based technique, NNGen does not require any training phase. It solely depends on the K-Nearest Neighbours algorithm. NNGen first finds k most similar commit diffs from the training set using *bag-of-words* model (i.e., term frequency) and cosine similarity measure. Since the authors do not provide any details of their bag-of-words implementation, we use the `CountVectorizer` API of the scikit-learn library in our replication. As suggested in the paper, we set the value of k to 5. From the top-k commits, NNGen selects the message from the most similar commit (using the BLEU score) as the final translation.

The fine-tuned CodeT5 model has the same architecture and the same hyper-parameters as those of Bugsplainer. However, unlike Bugsplainer, it neither uses the structural information from the source code nor performs any discriminatory pre-training.

Table 4.6 compares Bugsplainer and four baselines in terms of BLEU score, Semantic Similarity, and Exact Match. The results in the table are the mean of five runs with different random initialization of the parameters. We see that Bugsplainer outperforms all the baselines in terms of all three metrics. Only Fine-tuned CodeT5 is comparable with Bugsplainer. Therefore, we perform the Mann-Whitney U rank test [132] to see whether their performances over the five runs differ significantly. We found that Bugsplainer performs significantly higher than Fine-tuned CodeT5. That is $p\text{-}value = 0.008 < 0.05$ and Cliff's $d = 1.0$ (*large*) for all three metrics.

CommitGen relies on specific patterns in commit messages that might be generated by machines [56]. However, we removed auto-generated messages to ensure a high-quality dataset (Section 4.4.1). CommitGen is also based on the LSTM architecture that performs poorly with long inputs [30]. Thus, the low scores of CommitGen are explainable.

According to Google's AutoML Translation documentation, a BLEU score between 20 and 29 indicates that the gist of a generated message is clear but has significant errors. NNGen reuses existing commit messages from the training set and thus cannot analyze the dynamic behavior of software programs. Thus, such errors in the translation are also explainable.

Thung *et al.* [45] report that static analysis tools produce a lot of false negatives. This means they often do not produce any output for potentially buggy code. Our experiment with *pyflakes* shows a similar result, generating error messages for only 7.70% cases. Therefore, its poor metric scores are also understandable.

Summary of RQ$_3$: Bugsplainer outperforms all four baselines in terms of three performance metrics. According to our statistical tests, our technique outperforms the closest competitor – Fine-tuned CodeT5 – by a statistically significant margin.

Answering RQ$_4$ – Evaluation of Bugsplainer using a developer study

The metric-based evaluation demonstrates the benefit of our technique in generating bug explanations. Nonetheless, to address the subjective nature of the task, we also conduct a developer study that further demonstrates the benefit of Bugsplainer in a practical setting. Given the reference explanations of a software bug (e.g., the message of a bug-fix commit), we ask the developers to assess how accurate, precise, concise, and useful the explanations are. During the study, *we anonymize the model names* to avoid bias.

Study participants: The target population of our study is English-speaking software engineers with experience in Python programming language. We invite our participants in two ways. First, we contact software companies with a history of participation in academic studies to contribute to this research. Second, we advertise

Table 4.7: Quality aspects of generated explanations

Quality	**Overview**
Accurate	It provides the same factual information as the reference.
Precise	It can pinpoint the issue in the code.
Concise	It is short and still conveys the whole message.
Useful	The provided information has the potential to fix the bug.

the study on the authors' social networks to reach potential participants and increase the diversity of samples. As of August 31, 2022, we have received 20 responses to our developer study. Existing studies [133], [134] on bug reproduction and bug fixing conducted surveys with ≈ 20 participants to evaluate their tools or to validate their hypothesis. Bug reproduction and fixing are closely related to our research. Therefore, we believe that 20 participants might be sufficient to evaluate our tool and its generated explanations with the acceptance of fellow researchers. The participants have professional software development experience of 1 to 10 years and bug-fixing experience of 1 to 7 years. All of them are familiar with Python programming language as well. Such experience makes them suitable candidates for our study.

Study setup: In the developer study, each participant worked with 15 bug-fix commits and spent 30 minutes on average. To select these examples for our developer study, we apply random sampling without replacement to the testing set. To avoid information overload, we take the examples that (1) do not have more than five changed lines or more than 15 word-tokens in a single line within the commit diff, and (2) do not require any project-specific knowledge to understand the bug. We take the first 15 randomly sampled examples matching these two criteria.

We ask the participants to assess the accuracy, precision, conciseness, and usefulness of the explanations from Bugsplainer and baselines with respect to the reference explanations. Table 4.7 provides our definitions for these aspects. The participants assess these four aspects using a five-point Likert scale, where 1 indicates strongly disagree, and 5 indicates strongly agree. Please note that *we anonymize the model names and do not show the participants which explanation comes from which model to avoid any potential bias.* We collect 300 data points (15 questions $\times$ 5 explanations $\times$ 4 aspects) from each participant.

Our survey has been rigorously reviewed and approved by the Dalhousie University

Research Ethics Board (REB file #: 2022-5980).

Table 4.8: Comparison of Bugsplainer with baselines using developer study

Quality	Model	Mean	Median	Mode	2nd Mode
Accurate	pyflakes	1.841	1	1	3
	CommitGen	2.176	2	1	2
	NNGen	2.653	3	1	4
	Fine-tuned CodeT5	2.842	3	4	3
	Bugsplainer	**4.074**	**4**	**5**	**4**
Precise	pyflakes	1.768	1	1	3
	CommitGen	1.884	2	1	2
	NNGen	2.529	2	1	2
	Fine-tuned CodeT5	2.772	3	4	2
	Bugsplainer	**3.891**	**4**	**4**	**5**
Concise	pyflakes	2.182	2	1	4
	CommitGen	2.350	2	1	2
	NNGen	2.881	3	4	1
	Fine-tuned CodeT5	3.044	3	4	3
	Bugsplainer	**3.974**	**4**	**4**	**5**
Useful	pyflakes	1.724	1	1	3
	CommitGen	1.960	2	1	2
	NNGen	2.576	2	1	4
	Fine-tuned CodeT5	2.728	3	4	1
	Bugsplainer	**3.923**	**4**	**4**	**5**

Study result and discussion: Table 4.8 summarizes our findings from the developer study. We note that the participants find the explanations from Bugsplainer to be the most *accurate*, most *precise*, most *concise*, and most *useful.* Based on the median and mode values, we see that the participants agree the most with explanations from Bugsplainer. Similar to our findings in RQ$_3$, according to the developers, the closest competitor of Bugspaliner is Fine-tuned CodeT5. According to the mode values, the developers agree with Fine-tuned CodeT5 in many cases. However, looking at the 2nd mode values, we see that the developers also strongly noticeably disagree with Fine-tuned CodeT5, making the mean agreement poor. We also perform the Wilcoxon Signed Rank test to check whether the developers' agreement

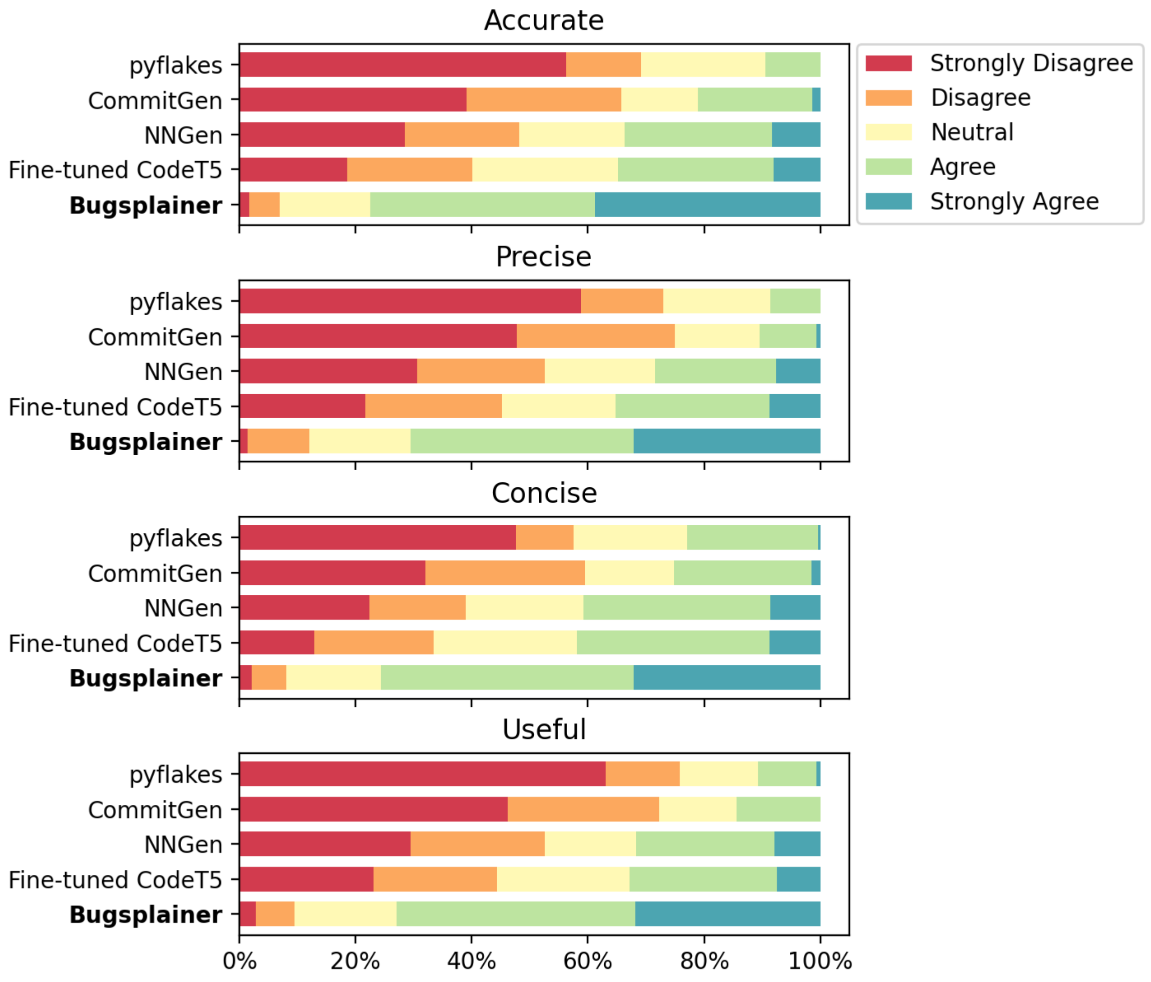


Figure 4.5: Comparison of Bugsplainer with the baselines using Likert scores

with Bugsplainer is significantly higher than that of Fine-tuned CodeT5. For accuracy, conciseness, precision, and usefulness, the p-values are $5.16e{-}23$, $2.74e{-}17$, $7.45e{-}18$ and $2.63e{-}23$ respectively; all are below the threshold of 0.05, which make the differences significant.

Fig. 4.5 shows the distribution of participants' agreement levels in different aspects. We see that the participants disagree with Bugsplainer very few times (highest 14% times in precision) with a substantial amount of agreement (highest 76% in accuracy). The developers strongly disagree with pyflakes and CommitGen nearly half the time. Such disagreement with pyflakes is explicable since it generates no error message for 13 out of 15 cases. However, CommitGen, even after explaining all cases, receives a high disagreement due to its generic and less informative explanations. The IR-based approach NNGen receives a similar amount of agreement to Fine-tuned CodeT5 but also much more disagreement. Being a retrieval-based technique, NNGen cannot adapt to dynamic and unseen code segments. Therefore, having a high disagreement in such cases might be explainable.

Summary of RQ$_4$: Professional developers with bug-fixing experience find the bug explanations from Bugsplainer to be accurate, precise, concise, and useful. Their preference levels for Bugsplainer over other baseline techniques are also significantly higher.

4.5 Prototype

In this section, we discuss Bugsplainer$_{\text{Web}}$ – a web-based prototype of our Bugsplainer technique. It enables easy access to our technique to end-users and provides various features to aid them in debugging. In particular, Bugsplainer$_{\text{Web}}$ provides the following features to explain software defects to the developers.

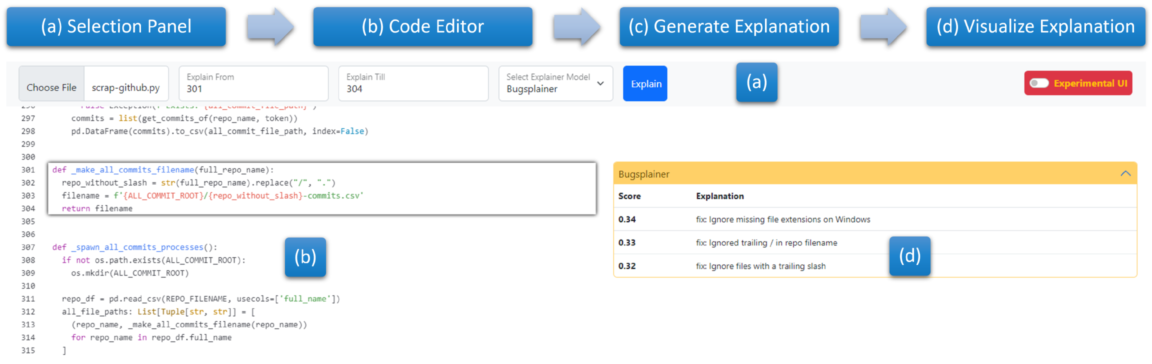
(a) Given a code segment, Bugsplainer can generate succinct explanations for the bug in the code.

(b) Provides three different variants of the explanation generation model – Bugsplainer, Bugsplainer 220M, and Fine-tuned CodeT5, and can help developers understand bugs from different perspectives.

(c) Facilitates code changes on-the-fly and allows one to check how the changes in the code affect the generated explanations.

(d) Provides two working modes – *production* and *experimental*, and allows a user to compare Bugsplainer with human written explanations and reason about the generated explanations.

Fig. 4.6 shows the user interface of our tool – Bugsplainer$_{\text{Web}}$. It is composed of four different components and has two different working modes. They are described as follows.



```python
    commits = list(get_commits_of(repo_name, token))
    pd.DataFrame(commits).to_csv(all_commit_file_path, index=False)


def _make_all_commits_filename(full_repo_name):
  repo_without_slash = str(full_repo_name).replace("/", ".")
  filename = f'{ALL_COMMIT_ROOT}/{repo_without_slash}-commits.csv'
  return filename


def _spawn_all_commits_processes():
  if not os.path.exists(ALL_COMMIT_ROOT):
    os.mkdir(ALL_COMMIT_ROOT)

  repo_df = pd.read_csv(REPO_FILENAME, usecols=['full_name'])
  all_file_paths: List[Tuple[str, str]] = [
    (repo_name, _make_all_commits_filename(repo_name))
    for repo_name in repo_df.full_name
  ]
```

Score	Explanation
0.34	fix: Ignore missing file extensions on Windows
0.33	fix: Ignored trailing / in repo filename
0.32	fix: Ignore files with a trailing slash

Figure 4.6: User interface of Bugsplainer$_{\text{Web}}$

4.5.1 Components

Bugsplainer$_{\text{Web}}$ is built using four different components – (a) selection panel, (b) code editor, (c) explanation generator, and (d) explanation visualizer. Among these four components, the explanation generator resides in a back-end server, whereas the remaining three reside in the front-end application. The following sections discuss these four components and their offered features.

Selection Panel

Fig. 4.6a shows the selection panel – the entry-point of Bugsplainer$_{\text{Web}}$. It contains a file selector which allows the user to choose a source code file from the local machine's file system. Since, at this stage, Bugsplainer$_{\text{Web}}$ only supports Python files (i.e., files with `.py` extension), the file selector filters out non-python files. Upon selection, Bugsplainer$_{\text{Web}}$ reads the contents of the file and loads them into the code editor (see details in Section 4.5.1).

Besides the file selector, the selection panel contains two input fields to indicate the starting and ending line numbers of the code segment that needs an explanation. It also contains a drop-down menu that allows one to select from three different models for explanation generation. These models are designed using deep learning technology and are responsible for generating explanations against the given code segment. The core difference among these models is their learned parameters and their analysis of the buggy source code segment.

Once the necessary inputs are provided, and the selections are made, the user can click the *Explain* button to see the explanations for the selected buggy code segment.

Upon clicking, the front-end application sends a request containing the selected code segment to the back-end server. Then, the back-end server generates and sends the top three explanations against the code segment back to the front-end application. Finally, the explanations are visualized to the user for review.

Code Editor

Fig. 4.6b shows the code editor of Bugsplainer$_{\text{Web}}$. Once a file is chosen from the selection panel, the code editor shows the contents of the file. From there, the user can choose one or more lines of code for explanation. This choice works synchronously with the selection panel. That is, choosing one or more lines in the code editor will update the line numbers in the selection panel and vice versa. For the user's convenience, the code editor highlights previously explained code segments as well as the current ones. Such highlighting not only prevents a user from generating duplicate explanations but also helps her quickly track back to previously explained code segments. The user can also change the code on-the-fly and receive an explanation for the change code. Comparing the explanations for different versions of the code can help the user better understand the underlying bug.

Explanation Generator

In Bugsplainer$_{\text{Web}}$, a back-end server is responsible for generating the explanations. It provides two different HTTP APIs – to access the list of deep learning models and to get explanations for a source code segment.

Bugsplainer$_{\text{Web}}$ offers three different deep learning models for explanation generation – Bugsplainer, Bugsplainer 220M, and Fine-tuned CodeT5. Among them, Bugsplainer and Bugsplainer 220M use structural information (e.g., abstract syntax tree) to generate the explanations. Therefore, they can explain a bug even if it lies in the structure of the code. On the contrary, Fine-tuned CodeT5 treats the source code as natural language text. Depending on the context, these different deep learning models can focus on various aspects of the bug in generating their explanations, which can help the user understand a bug comprehensively.

The front-end application sends an HTTP request to the back-end explanation generator upon clicking the *Explain* button in the selection panel. The request contains the source code, starting and ending line numbers of the selected code segment, and the name of the chosen deep learning model. If the selected model is either Bugsplainer or Bugsplainer 220M, then the server captures the structural information from the source code in an encoded form. First, it converts the whole source code into an Abstract Syntax Tree (AST) and recursively prunes the AST nodes that are not a part of the selected code segment. Second, the explanation generator module converts the partial AST into a token sequence using an algorithm, namely *diffSBT*. diffSBT captures structural information from the partial AST by traversing the remaining nodes and edges.

Once the diffSBT sequence is generated from the selected code segment, they are passed to the deep learning model. Then the model generates explanations for the bug in the code with a confidence score. Finally, the back-end server returns the top three explanations along with their confidence scores to the front-end application.

Explanation Visualizer

The Explanation Visualizer component (Fig. 4.6d) is responsible for visualizing the explanations received from the back-end server in a user-friendly manner. It groups all the received explanations based on the location of the source code segment. Since there can be several explanations for the same code segment from different models, it might get cluttered. As a solution, we keep all the explanations collapsed by default. When the user hovers on (i.e., moves the cursor on) an explanation group or corresponding source code segment, the explanations are shown along with their confidence score.

To differentiate between different explanation groups, we color the groups cyclically with six different colors. These colors contrast with each other and the background according to the Web Content Accessibility Guidelines (WCAG)[11]. We also use a set of colors that color-blind persons can distinguish [135].

[11]https://www.w3.org/TR/WCAG

4.5.2 Working Modes

Bugsplainer$_{\text{Web}}$ can work in two different modes – *production* and *experimental*. A user can choose either mode using a toggle button on the top right corner of the application (Fig. 4.6). The production mode is the default mode where Bugsplainer$_{\text{Web}}$ delivers automatically generated explanations for the buggy code as described in the above sections.

In the experimental mode, instead of selecting a file from the machine's file system, the user can select a file from a pre-defined set provided by Bugsplainer$_{\text{Web}}$. Each file from the set has a pre-defined bug location with human-written explanations associated with it. Upon selecting a file, along with the file contents, Bugsplainer$_{\text{Web}}$ shows the corresponding human-written explanations. It also highlights the buggy code segments connected to these human-written explanations. Then, the user can generate explanations for any code segment and compare them against the human-written explanations. Such comparison not only helps the user grow confidence in Bugsplainer$_{\text{Web}}$ but also helps reason about why Bugsplainer$_{\text{Web}}$ explains a bug in a certain way.

4.6 Threats To Validity

Threats of internal validity relate to experimental errors and biases. Re-implementation of the existing baseline techniques could pose a threat. However, our implementation of NNGen [56] is based on the well-known k-nearest neighbor algorithm. CommitGen [53] uses a framework and reports all important hyperparameters. We use the same framework and the reported parameters for our implementation. For pyflakes [39], we use the officially provided package and follow the official documentation. Fine-tuned CodeT5 adopts the same model architecture as that of Bugsplainer. Thus, the threats related to replication might be mitigated. We also repeat our experiments five times and compare the performance with that of baselines to mitigate any bias due to random trials.

Threats to external validity relate to the generalizability of our work. Even though Bugsplainer is evaluated using only Python code, the underlying algorithm is language-agnostic and can be easily adapted to any traditional programming language. The use of metric-based evaluation might also pose a threat to the real-world usability of our approach [54], [66]. To mitigate this threat, we also conduct a developer study involving 20 participants from six different countries. As the developer study suggests, our bug explanations were also found to be accurate and useful in real-world scenarios.

The performance of Bugsplainer might depend on the precision of defect prediction tools (e.g., Bugsplorer). To minimize this dependency, Bugsplainer accepts a range of lines containing both buggy lines and their surrounding lines as input during explanation generation. However, we do not indicate which lines among them contain a bug. Thus, precise localization of the bug either by developers or by existing tools might not be necessary to generate explanations using our tool.

4.7 Related Work

4.7.1 Explanation of Software Bugs

Software claim 50% of development time and cost the global economy billions of dollars every year [3], [24]. While there have been numerous approaches to finding or repairing software bugs, neither many approaches attempt to explain the bugs in the source code to the developers, nor are they practical and scalable enough for industry-wide use [23], [24]. Several tools (e.g., FindBugs [35], PMD [36], SonarLint [37], PyLint [38], and pyflakes [39]) attempt to explain bugs using static analysis. Unfortunately, their utility could be limited due to their high false-positive results and lack of meaningful, actionable explanations [40]–[42]. Furthermore, their explanations can be too generic and limited by their templated natures [44]. Recent studies suggest complementing these messages with rule graphs [136], assertive error explanation [40], and interactive feedback from developers [137]. However, static analysis tools are always likely to be restricted to a fixed set of rules.

Besides the static analysis, there have been several attempts to explain a program's behaviors, failed tests, bug-fixing patches, undocumented code, and intelligent behaviors. Ko *et al.* [137] design an interrogative debugging system for the Alice programming environment [138] where a novice learner can inquire why a program behaves unexpectedly or why it does not show an expected behavior. Lim *et al.* [139] later suggest that these why and why not questions are essential to improve a user's understanding or perception of an intelligent system. Zhang *et al.* [140] explain a failed test case by automatically performing failure-correcting edits (e.g., replacement of identifiers' values) and synthesizing a code comment from them. Tabaei Befrouei *et al.* [141] use program execution traces to explain concurrency bugs. Later, Bragaglio *et al.* [142] remove irrelevant information from the execution traces to understand the cause of the unexpected behavior. Rahman *et al.* [143] also explain the quality concerns of a piece of code using relevant comments mined from a popular programming Q&A site – Stack Overflow. Liang *et al.* [144] investigate what should be included in a patch explanation, such as expected program behaviors or a high-level summary of code changes. Recently, Pornprasit *et al.* [145] also explained why the changed code can be defect-prone by visualizing how specific local rules are satisfied by a change. Although all these studies and approaches are relevant and are a source of our inspiration, they might be restricted to only specific problem contexts (e.g., Alice programming language [138], failed tests [140]) or certain types of bugs (e.g., concurrency bugs [141]). Besides, their explanation might not always contain what needs to be done to correct the buggy code.

Unlike these traditional approaches, Bugsplainer is not restricted to any specific context or bugs. Besides, it can generate explanations that resemble that of humans and are accurate, precise, concise, and useful (see Section 4.4.3 for details).

4.7.2 Translation of Source Code into Texts

Our work is also related to code translation into natural language texts. Existing approaches translate code into the natural language to generate code comments, review comments, and commit messages.

Earlier works on code comment generation utilize hand-crafted templates [146], [147] and Information Retrieval (IR) [148], [149], while recent works more depend on learning-based approaches [47], [61], [150]. Wei *et al.* [151] combine both IR and NMT in comment generation. Recently, Mastropaolo *et al.* [152] used the Text-To-Text Transfer Transformer (T5) to perform several tasks, including code comment generation. Their comments explain what happens in the code rather than what makes it buggy. On the contrary, Bugsplainer not only explains the buggy code but also suggests useful information to correct the bug in the code (e.g., Table 4.1).

To generate code review comments, Tufano *et al.* [124] make use of the CodeT5 [67] model with Stack Overflow data and fine-tune their model with pairs of function and review comment pairs from GitHub. Later, Hong *et al.* [125] use Gestalt Pattern Matching (GPM) to mine candidate review comments of similar methods from a large corpus of source code repositories. However, both approaches treat source code as regular texts (e.g., a sequence of tokens) overlooking the structures. On the other hand, Bugsplainer leverages the structures of code through diffSBT sequences and learns to differentiate between buggy and bug-free code as a part of explanation generation.

To generate commit messages, several studies adopt an attention mechanism with RNN [46], [53], [55], [153] and leverage structural information [55], [153]. Xu *et al.* [153] jointly model the semantic representation and structural representation of code changes where they substitute identifier names with placeholders in the code. While they treat structure and semantics as different information, Bugsplainer embeds the structural information within the code using diffSBT. Liu *et al.* [55] capture both ASTs of code changes where they convert each AST into path sequence. However, converting each change from the AST into its own path might lead to long, redundant sequences, which could hurt the translation performance. Surprisingly, Liu *et al.* [56] show that a simple IR-based approach, NNGen, has promising capability in commit message generation due to its repetitive nature. NNGen is closely related to ours due to their nature of translation. We thus compare Bugsplainer with them using experiments, and the details can be found in Section 4.4.3.

4.8 Summary

While Section 3 predicts defects at the line level, this study – Bugsplainer – complements the former with useful and precise explanations. Bugsplainer learns from thousands of bug-fix commits and leverages structural information and defective patterns from the source code to generate an explanation for a defect. Our evaluation using three performance metrics shows that Bugsplainer can generate understandable and good explanations according to Google's standard and outperform multiple baselines from the literature. We also conducted a developer study involving 20 participants where the explanations from Bugsplainer were found to be more accurate, more precise, more concise, and more useful than the baselines.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Software defects not only claim precious development time but also cost billions of dollars every year. Predicting these defects at the line level and explaining them to the developers can significantly reduce the costs of software quality assurance. Unfortunately, existing techniques for line-level defect prediction might fail to capture the local context of a software defect. Furthermore, explaining the root cause of software defects is an open problem that warrants significant investigation. In this thesis, we propose two novel tools to predict defects at the line level and explain their root causes using natural language texts, respectively. Our first tool, *Bugsplorer*, is a novel transformer-based technique that predicts defects at the line level. It leverages a hierarchical structure of transformer networks to estimate the attention values for two types of code elements: code tokens and code lines. We train and evaluate Bugsplorer with two benchmark datasets and determine its classification performance and cost-effectiveness. We find that Bugsplorer can predict defective code lines with 26-68% higher accuracy than that of the state-of-the-art model. It can also reduce the effort in finding defective lines by 72-81%. Our second tool, *Bugsplainer*, is a novel neural machine translation technique that generates explanations for defective code segments. Our technique leverages both structural information and defective code patterns from source code and employs neural machine translation with a transformer model to generate the explanations for defects. We evaluate Bugsplainer using three metrics (i.e., BLEU score, Semantic Similarity, and Exact Match) where our technique outperforms the baselines. We also conducted a developer study involving 20 participants, and our explanations were found to be more accurate and useful compared to the baselines.

5.2 Future Work

There are several research avenues that can be pursued in the future. We discuss possible future works from each of our studies as follows.

5.2.1 Bugsplorer

Code Structure: Our manual analysis with Bugsplorer reveals that embedding structural information to the source code might prevent the model from being confused with long comments resembling code. To enhance Bugsplorer's robustness against rarely used syntax, it can be trained on examples from the official documentation of programming languages.

Model Architecture: Experimenting with different model architectures is a crucial aspect of future work. Currently, Bugsplorer uses RoBERTa [90] as its transformer architecture, which limits the maximum number of lines in a source code document (i.e., $L = 512$). However, newer transformer architectures like TransformerXL [91] eliminate this limitation and can handle variable-length inputs. Another approach worth considering is implementing the entire model using decoders (e.g., GPT) instead of encoders. This would enable the model to predict defective lines autoregressively (see Section 2.4) and handle inputs of any length. It provides more flexibility and freedom for the model's predictions.

5.2.2 Bugsplainer

Code Structure: Future works might investigate how to encode the structural information from source code in a more compact and efficient format and how to better leverage the structural differences between buggy and bug-free code. They might also experiment with different types of structural information, such as Data Flow Graph (DFG) [154], Control Flow Graph (CFG) [155], and Program Dependency Graph (PDG) [156]. This might help us better understand the underlying semantics of software bugs.

Transfer Learning: Another direction for future works could be transferring or reusing the parameters learned by defect prediction for explanation generation. Our experiments with Bugsplorer show that it can achieve reasonable performance with the T5 encoder model (see Section 3.4.4, RQ_3). Since Bugsplainer also uses the T5 architecture for explanation generation, these two techniques might be able to share their knowledge.

Bibliography

[1] Institute of Electrical and Electronics Engineers, "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990.

[2] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 602–614.

[3] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, *Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers*, 2013.

[4] R. Glass, "Frequently forgotten fundamental facts about software engineering," *IEEE Software*, vol. 18, no. 3, pp. 112–111, 2001.

[5] S. Matteson, *Report: Software failure caused $1.7 trillion in financial losses in 2017*, 2018. [Online]. Available: `https://tek.io/2FBNlf2i` (visited on 01/18/2022).

[6] F. Brandy, *Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year*, 2013. [Online]. Available: `https://goo.gl/okoj21` (visited on 01/18/2022).

[7] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Investigating code review practices in defective files: An empirical study of the qt system," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, IEEE, 2015, pp. 168–179.

[8] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Revisiting code ownership and its relationship with software quality in the scope of modern code review," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 1039–1050.

[9] C. Pornprasit and C. K. Tantithamthavorn, "DeepLineDP: Towards a deep learning approach for line-level defect prediction," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 84–98, 2022.

[10] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" In *2012 34th International Conference on Software Engineering (ICSE)*, IEEE, 2012, pp. 255–265.

[11] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," *IEEE Transactions on Software Engineering*, vol. 45, no. 2, pp. 111–147, 2017.

[12] J. Pachouly, S. Ahirrao, K. Kotecha, G. Selvachandran, and A. Abraham, "A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools," *Engineering Applications of Artificial Intelligence*, vol. 111, p. 104 773, 2022.

[13] Z. M. Zain, S. Sakri, and N. H. A. Ismail, "Application of deep learning in software defect prediction: Systematic literature review and meta-analysis," *Information and Software Technology*, p. 107 175, 2023.

[14] Y. Zhao, K. Damevski, and H. Chen, "A systematic survey of just-in-time software defect prediction," *ACM Computing Surveys*, vol. 55, no. 10, pp. 1–35, 2023.

[15] L. Gong, G. K. Rajbahadur, A. E. Hassan, and S. Jiang, "Revisiting the impact of dependency network metrics on software defect prediction," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 5030–5049, 2021.

[16] X. Yu, K. E. Bennin, J. Liu, J. W. Keung, X. Yin, and Z. Xu, "An empirical study of learning to rank techniques for effort-aware defect prediction," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2019, pp. 298–309.

[17] J. Jiarpakdee, C. K. Tantithamthavorn, and J. Grundy, "Practitioners' perceptions of the goals and visual explanations of defect prediction models," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, IEEE, 2021, pp. 432–443.

[18] J. Chen *et al.*, "Software visualization and deep transfer learning for effective software defect prediction," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 578–589.

[19] T. Shippey, D. Bowes, and T. Hall, “Automatically identifying code features for software defect prediction: Using ast n-grams,” *Information and Software Technology*, vol. 106, pp. 142–160, 2019.

[20] 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.

[21] C. Pornprasit and C. K. Tantithamthavorn, “JITLine: A simpler, better, faster, finer-grained just-in-time defect prediction,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, IEEE, 2021, pp. 369–379.

[22] S. Herbold, “On the costs and profit of software defect prediction,” *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2617–2631, 2019.

[23] P. S. Kochhar, X. Xia, D. Lo, and S. Li, “Practitioners’ expectations on automated fault localization,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 165–176.

[24] 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.

[25] S. Wang, T. Liu, and L. Tan, “Automatically learning semantic features for defect prediction,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 297–308.

[26] J. Li, P. He, J. Zhu, and M. R. Lyu, “Software defect prediction via convolutional neural network,” in *2017 IEEE international conference on software quality, reliability and security (QRS)*, IEEE, 2017, pp. 318–328.

[27] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, “Improving bug detection via context-based code representation learning and attention-based neural networks,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, 2019.

[28] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 67–85, 2018.

[29] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "Vuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2019.

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

[31] D. Jurafsky and J. H. Martin, "Vector semantics and embeddings," in *Speech and Language Processing*, 3rd ed. Prentice-Hall, Inc., 2022, pp. 1–17.

[32] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 3982–3992.

[33] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[34] M. T. Ribeiro, S. Singh, and C. Guestrin, ""Why should I trust you?" Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 1135–1144.

[35] B. Pugh and A. Loskutov, *FindBugs™ - Find Bugs in Java Programs*, 2021. [Online]. Available: `http://findbugs.sourceforge.net/` (visited on 01/18/2022).

[36] PMD Team, *PMD Source Code Analyzer*, 2021. [Online]. Available: `https://pmd.github.io/` (visited on 01/18/2022).

[37] SonarSource, *SonarLint — Free and Open Source Code Quality & Security IDE Extension*, 2021. [Online]. Available: `https://www.sonarlint.org/` (visited on 01/18/2022).

[38] Python Code Quality Authority, *Pylint - code analysis for Python*, 2021. [Online]. Available: `https://pylint.org/` (visited on 01/18/2022).

[39] A. Sottile, *pyflakes: A simple program which checks Python source files for errors*, 2022. [Online]. Available: `https://github.com/PyCQA/pyflakes` (visited on 01/18/2022).

[40] T. Barik, "How should static analysis tools explain anomalies to developers?" In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 1118–1120.

[41] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" In *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 672–681.

[42] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2007, pp. 1–8.

[43] M. Yan, X. Zhang, L. Xu, H. Hu, S. Sun, and X. Xia, "Revisiting the correlation between alerts and software defects: A case study on myfaces, camel, and cxf," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, IEEE, vol. 1, 2017, pp. 103–108.

[44] M. Nachtigall, M. Schlichtig, and E. Bodden, "A large-scale study of usability criteria addressed by static analysis tools," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 532–543.

[45] F. Thung, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu, "To what extent could we detect field defects? an extended empirical study of false negatives in static bug-finding tools," *Automated Software Engineering*, vol. 22, pp. 561–602, 2015.

[46] P. Loyola, E. Marrese-Taylor, J. Balazs, Y. Matsuo, and F. Satoh, "Content aware source code change description generation," in *Proceedings of the 11th International Conference on Natural Language Generation*, 2018, pp. 119–128.

[47] 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.

[48] D. Gros, H. Sezhiyan, P. Devanbu, and Z. Yu, "Code to comment "translation": Data, metrics, baselining & evaluation," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2020, pp. 746–757.

[49] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019. [Online]. Available: `https://proceedings.neurips.cc/paper/2019/file/e52ad5c9f751f599492b4f087ed7ecfc-Paper.pdf`.

[50] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[51] J. Austin *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[52] M. Chen *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[53] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2017, pp. 135–146.

[54] W. Tao *et al.*, "On the evaluation of commit message generation models: An experimental study," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2021, pp. 126–136.

[55] S. Liu, C. Gao, S. Chen, L. Y. Nie, and Y. Liu, "ATOM: Commit message generation based on abstract syntax tree and hybrid ranking," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1800–1817, 2020.

[56] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, “Neural-machine-translation-based commit message generation: How far are we?” In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 373–384.

[57] 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.

[58] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, “An empirical study on learning bug-fixing patches in the wild via neural machine translation,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[59] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, “Sequencer: Sequence-to-sequence learning for end-to-end program repair,” *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[60] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, “A survey of machine learning for big code and naturalness,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[61] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, “Summarizing source code using a neural attention model,” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.

[62] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, “On the “naturalness” of buggy code,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 428–439.

[63] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, “Deepjit: An end-to-end deep learning framework for just-in-time defect prediction,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, IEEE, 2019, pp. 34–45.

[64] P. Mahbub, O. Shuvo, and M. M. Rahman, "Defectors: A large, diverse python dataset for defect prediction," in *Proceeding of The 20th International Conference on Mining Software Repositories*, 2023, pp. 393–397.

[65] 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.

[66] 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.

[67] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[68] Y. Ren *et al.*, "Fastspeech: Fast, robust and controllable text to speech," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[69] Y. Ren *et al.*, "Fastspeech 2: Fast and high-quality end-to-end text to speech," in *International Conference on Learning Representations*, 2020.

[70] G. Saon, Z. Tüske, D. Bolanos, and B. Kingsbury, "Advancing rnn transducer technology for speech recognition," in *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2021, pp. 5654–5658.

[71] A. Amberkar, P. Awasarmol, G. Deshmukh, and P. Dave, "Speech recognition using recurrent neural networks," in *2018 international conference on current trends towards converging technologies (ICCTCT)*, IEEE, 2018, pp. 1–4.

[72] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[73] K. Cho *et al.*, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[74] K. Xu *et al.*, "Show, attend and tell: Neural image caption generation with visual attention," in *International conference on machine learning*, PMLR, 2015, pp. 2048–2057.

[75] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[76] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[77] Z. Lin *et al.*, "A structured self-attentive sentence embedding," *arXiv preprint arXiv:1703.03130*, 2017.

[78] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[79] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[80] J. Zhang, Y. Zhao, M. Saleh, and P. Liu, "Pegasus: Pre-training with extracted gap-sentences for abstractive summarization," in *International Conference on Machine Learning*, PMLR, 2020, pp. 11 328–11 339.

[81] Y. Lan and J. Jiang, "Query graph generation for answering multi-hop complex questions from knowledge bases," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 969–974.

[82] A. Roberts, C. Raffel, and N. Shazeer, "How much knowledge can you pack into the parameters of a language model?" *arXiv preprint arXiv:2002.08910*, 2020.

[83] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[84] 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.

[85] U. Kamath, J. Liu, and J. Whitaker, *Deep learning for NLP and speech recognition*. Springer, 2019, vol. 84.

[86] J. Ive, "Natural language processing: A machine learning perspective by yue zhang and zhiyang teng," *Computational Linguistics*, vol. 48, no. 1, pp. 233–235, 2022.

[87] O. Levy, Y. Goldberg, and I. Dagan, "Improving distributional similarity with lessons learned from word embeddings," *Transactions of the association for computational linguistics*, vol. 3, pp. 211–225, 2015.

[88] 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.

[89] 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 and Symposium on the Foundations of Software Engineering*, 2019, pp. 964–974.

[90] Y. Liu *et al.*, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[91] Z. Dai, Z. Yang, Y. Yang, J. G. Carbonell, Q. Le, and R. Salakhutdinov, "Transformer-xl: Attentive language models beyond a fixed-length context," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 2978–2988.

[92] R. Harper, "Abstract syntax," in *Practical Foundations for Programming Languages*, 2nd ed. Cambridge University Press, 2016, pp. 3–11. DOI: 10.1017/CBO9781316576892.003.

[93] H. Keshavarz and M. Nagappan, "Apachejit: A large dataset for just-in-time defect prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 191–195.

[94] Z. Feng *et al.*, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[95] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 1715–1725.

[96] C. Dwork *et al.*, "The mathematics of information coding, extraction, and distribution," *The IMA Volumes in Mathematics and its applications*, vol. 107, p. 82, 1999.

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

[98] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[99] C. Ferri, J. Hernández-Orallo, and R. Modroiu, "An experimental comparison of performance measures for classification," *Pattern recognition letters*, vol. 30, no. 1, pp. 27–38, 2009.

[100] R. J. Urbanowicz and J. H. Moore, "Exstracs 2.0: Description and evaluation of a scalable learning classifier system," *Evolutionary intelligence*, vol. 8, pp. 89–116, 2015.

[101] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[102] A. LeClair and C. McMillan, "Recommendations for datasets for source code summarization," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019, pp. 3931–3937.

[103] X. Zhu, C. Vondrick, C. C. Fowlkes, and D. Ramanan, "Do we need more training data?" *International Journal of Computer Vision*, vol. 119, no. 1, pp. 76–92, 2016.

[104] A. N. Çayır and T. S. Navruz, "Effect of dataset size on deep learning in voice recognition," in *2021 3rd International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, IEEE, 2021, pp. 1–5.

[105] Y. Kamei *et al.*, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[106] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 560–560.

[107] Y. Fan, X. Xia, D. A. Da Costa, D. Lo, A. E. Hassan, and S. Li, "The impact of mislabeled changes by szz on just-in-time defect prediction," *IEEE transactions on software engineering*, vol. 47, no. 8, pp. 1559–1586, 2019.

[108] D. A. Da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2016.

[109] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Ieee, 2013, pp. 279–289.

[110] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 908–911.

[111] S. Kim, T. Zimmermann, K. Pan, E. James Jr, *et al.*, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM international conference on automated software engineering (ASE'06)*, IEEE, 2006, pp. 81–90.

[112] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

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

[114] S. Lu *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[115] W. Pree, *Design patterns for object-oriented software development.* ACM Press/Addison-Wesley Publishing Co., 1995.

[116] G. Rodriguez-Perez, M. Nagappan, and G. Robles, "Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the openstack project," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1400–1416, 2020.

[117] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *2010 IEEE international conference on software maintenance*, IEEE, 2010, pp. 1–10.

[118] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *2012 34th international conference on software engineering (ICSE)*, IEEE, 2012, pp. 200–210.

[119] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *Journal of Systems and Software*, vol. 150, pp. 22–36, 2019.

[120] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2823–2862, 2019.

[121] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 518–529.

[122] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang, “Perceptions, expectations, and challenges in defect prediction,” *IEEE Transactions on Software Engineering*, vol. 46, no. 11, pp. 1241–1266, 2018.

[123] M. M. Rahman, F. Khomh, S. Yeasmin, and C. K. Roy, “The forgotten role of search queries in ir-based bug localization: An empirical study,” *Empirical Software Engineering*, vol. 26, no. 6, p. 116, 2021.

[124] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, “Towards automating code review activities,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 163–174.

[125] Y. Hong, C. Tantithamthavorn, P. Thongtanunam, and A. Aleti, “CommentFinder: a simpler, faster, more accurate code review comments recommendation,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 507–519.

[126] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, “Cscc: Simple, efficient, context sensitive code completion,” in *2014 IEEE International Conference on Software Maintenance and Evolution*, IEEE, 2014, pp. 71–80.

[127] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” *arXiv preprint arXiv:1301.3781*, 2013.

[128] M. Fischer, M. Pinzger, and H. Gall, “Populating a release history database from version control and bug tracking systems,” in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, IEEE, 2003, pp. 23–32.

[129] C.-Y. Lin and F. J. Och, “Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics,” in *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, 2004, pp. 605–612.

[130] M. Allamanis, “The adverse effects of code duplication in machine learning models of code,” in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 143–153.

[131] R. Sennrich *et al.*, “Nematus: A toolkit for neural machine translation,” in *Proceedings of the Software Demonstrations of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, 2017, pp. 65–68.

[132] H. B. Mann and D. R. Whitney, “On a test of whether one of two random variables is stochastically larger than the other,” *The annals of mathematical statistics*, pp. 50–60, 1947.

[133] A. J. Ko and B. A. Myers, “Debugging reinvented: Asking and answering why and why not questions about program behavior,” in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 301–310.

[134] R. F. Silva, C. K. Roy, M. M. Rahman, K. A. Schneider, K. Paixao, and M. de Almeida Maia, “Recommending comprehensive solutions for programming tasks by mining crowd knowledge,” in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, IEEE, 2019, pp. 358–368.

[135] B. Wong, “Points of view: Color blindness,” *Nature Methods*, vol. 8, no. 6, pp. 441–441, Jan. 2011, ISSN: 1548-7105. DOI: `10.1038/nmeth.1618`. [Online]. Available: `https://doi.org/10.1038/nmeth.1618`.

[136] L. N. Q. Do and E. Bodden, “Explaining static analysis with rule graphs,” *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 678–690, 2020.

[137] A. J. Ko and B. A. Myers, “Designing the whyline: A debugging interface for asking questions about program behavior,” in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2004, pp. 151–158.

[138] C. M. University, *Alice – Tell Stories. Build Games. Learn to Program.* 2021. [Online]. Available: `https://www.alice.org/` (visited on 01/18/2022).

[139] B. Y. Lim, A. K. Dey, and D. Avrahami, “Why and why not explanations improve the intelligibility of context-aware intelligent systems,” in *Proceedings of the SIGCHI conference on human factors in computing systems*, 2009, pp. 2119–2128.

[140] S. Zhang, C. Zhang, and M. D. Ernst, "Automated documentation inference to explain failed tests," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, IEEE, 2011, pp. 63–72.

[141] M. Tabaei Befrouei, C. Wang, and G. Weissenbacher, "Abstraction and mining of traces to explain concurrency bugs," *Formal Methods in System Design*, vol. 49, pp. 1–32, 2016.

[142] M. Bragaglio, N. Donatelli, S. Germiniani, and G. Pravadelli, "System-level bug explanation through program slicing and instruction clusterization," in *2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC)*, IEEE, 2021, pp. 1–6.

[143] M. M. Rahman, C. K. Roy, and I. Keivanloo, "Recommending insightful comments for source code using crowdsourced knowledge," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, 2015, pp. 81–90.

[144] J. Liang, Y. Hou, S. Zhou, J. Chen, Y. Xiong, and G. Huang, "How to explain a patch: An empirical study of patch explanations in open source projects," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2019, pp. 58–69.

[145] C. Pornprasit, C. Tantithamthavorn, J. Jiarpakdee, M. Fu, and P. Thongtanunam, "Pyexplainer: Explaining the predictions of just-in-time defect models," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 407–418.

[146] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 279–290.

[147] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 101–110.

[148] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*, IEEE, 2010, pp. 35–44.

[149] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2013, pp. 562–567.

[150] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International conference on machine learning*, PMLR, 2016, pp. 2091–2100.

[151] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: Exemplar-based neural comment generation," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 349–360.

[152] A. Mastropaolo *et al.*, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 336–347.

[153] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, "Commit message generation for source code changes," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, 2019, pp. 3975–3981.

[154] Kavi, Buckles, and Bhat, "A formal definition of data flow graph models," *IEEE Transactions on computers*, vol. 100, no. 11, pp. 940–948, 1986.

[155] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[156] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

Appendix A

Complementary Materials

A.1 Published Papers

- **Parvez Mahbub**, Ohiduzzaman Shuvo, and M. Masudur Rahman. Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation. In Proceeding of *The 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023)*, pp. 12, Melbourne, Australia, May 2023 (To appear).
- **Parvez Mahbub**, Ohiduzzaman Shuvo, and M. Masudur Rahman. Defectors: A Large, Diverse Python Dataset for Defect Prediction. In Proceeding of *The 20th International Conference on Mining Software Repositories (MSR 2023)*, pp. 5, Melbourne, Australia, May 2023 (To appear).

A.2 Replication Packages

Bugsplorer: https://github.com/parvezmrobin/bugsplorer-replication-package
Bugsplainer: https://github.com/parvezmrobin/bugsplainer-replication-package

A.3 Prototypes

Bugsplorer: https://github.com/parvezmrobin/bugsplorer-webapp
Bugsplainer: https://github.com/parvezmrobin/bugsplainer-webapp

A.4 Demo Video

Bugsplorer: https://youtu.be/mrp5W-WR5Jk
Bugsplainer: https://youtu.be/xga-ScvULpk

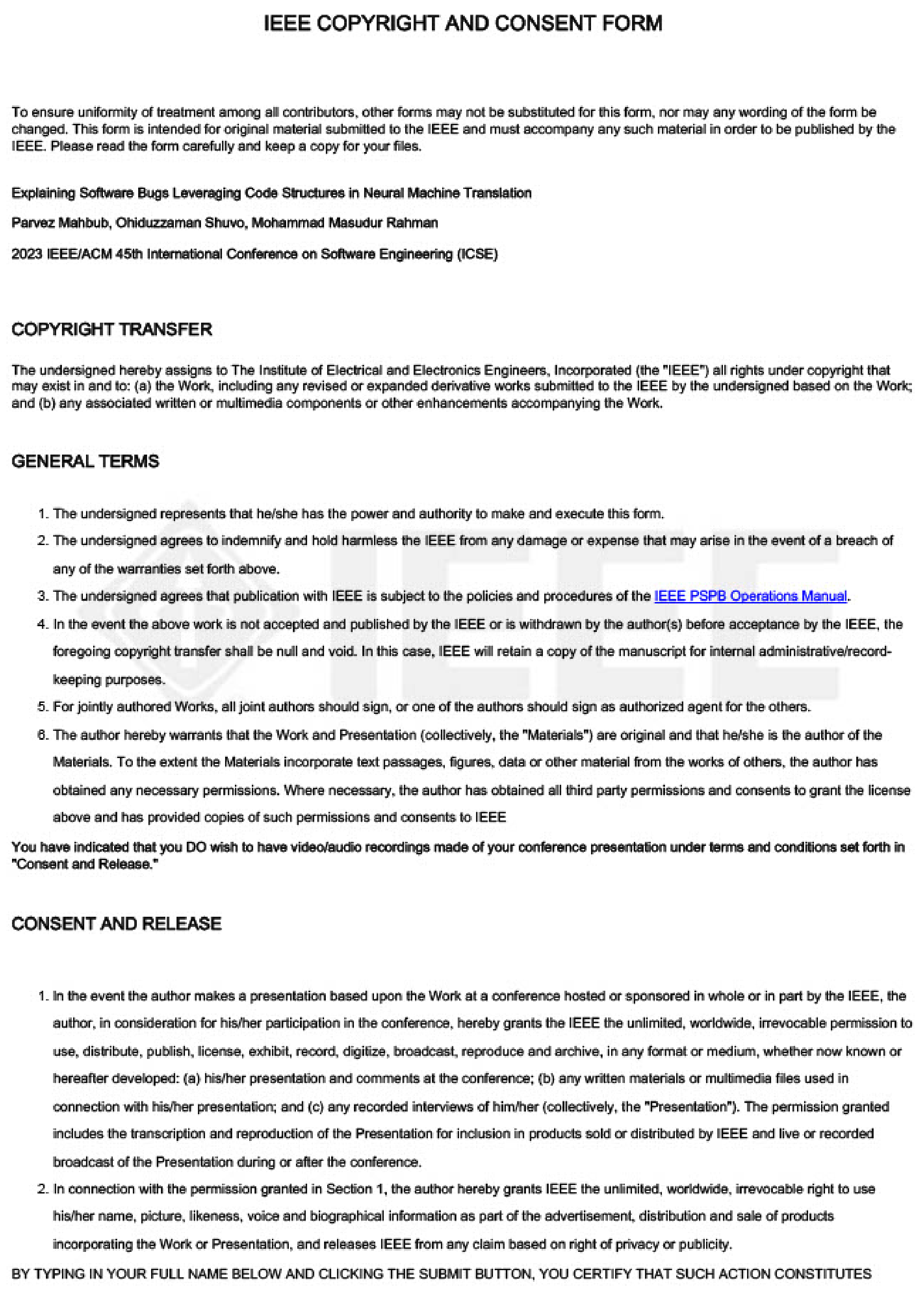
Appendix B

Copyright Release

B.1 Bugsplainer in ICSE 2023

Parvez Mahbub, Ohiduzzaman Shuvo, and M. Masudur Rahman. *Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation.* In Proceeding of The 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023), pp. 640-652, Melbourne, Australia, May 2023.

Following is the copyright agreement with Institute of Electrical and Electronics Engineers.



IEEE COPYRIGHT AND CONSENT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation

Parvez Mahbub, Ohiduzzaman Shuvo, Mohammad Masudur Rahman

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)

COPYRIGHT TRANSFER

The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

GENERAL TERMS

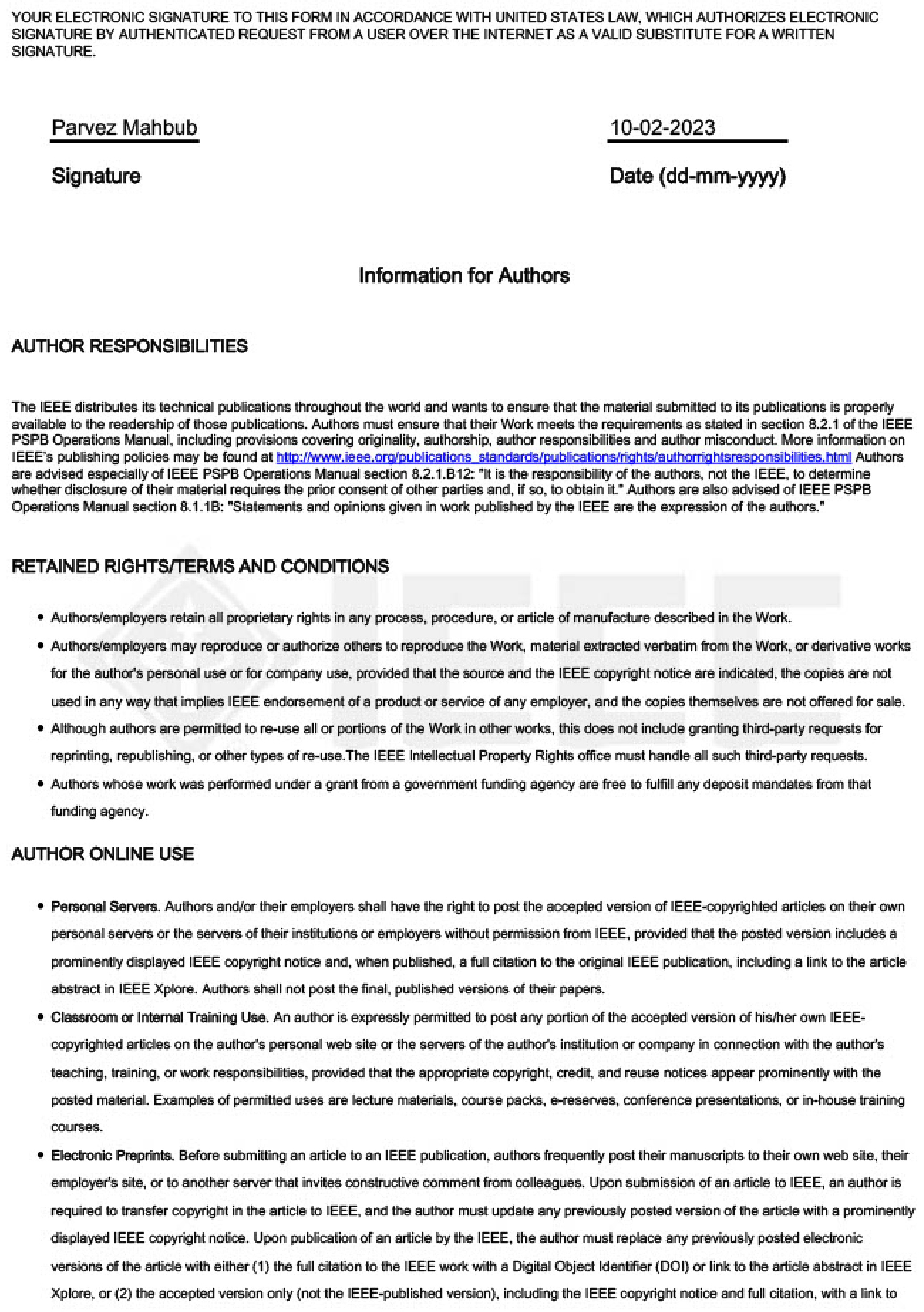
1. The undersigned represents that he/she has the power and authority to make and execute this form.
2. The undersigned agrees to indemnify and hold harmless the IEEE from any damage or expense that may arise in the event of a breach of any of the warranties set forth above.
3. The undersigned agrees that publication with IEEE is subject to the policies and procedures of the IEEE PSPB Operations Manual.
4. In the event the above work is not accepted and published by the IEEE or is withdrawn by the author(s) before acceptance by the IEEE, the foregoing copyright transfer shall be null and void. In this case, IEEE will retain a copy of the manuscript for internal administrative/record-keeping purposes.
5. For jointly authored Works, all joint authors should sign, or one of the authors should sign as authorized agent for the others.
6. The author hereby warrants that the Work and Presentation (collectively, the "Materials") are original and that he/she is the author of the Materials. To the extent the Materials incorporate text passages, figures, data or other material from the works of others, the author has obtained any necessary permissions. Where necessary, the author has obtained all third party permissions and consents to grant the license above and has provided copies of such permissions and consents to IEEE

You have indicated that you DO wish to have video/audio recordings made of your conference presentation under terms and conditions set forth in "Consent and Release."

CONSENT AND RELEASE

1. In the event the author makes a presentation based upon the Work at a conference hosted or sponsored in whole or in part by the IEEE, the author, in consideration for his/her participation in the conference, hereby grants the IEEE the unlimited, worldwide, irrevocable permission to use, distribute, publish, license, exhibit, record, digitize, broadcast, reproduce and archive, in any format or medium, whether now known or hereafter developed: (a) his/her presentation and comments at the conference; (b) any written materials or multimedia files used in connection with his/her presentation; and (c) any recorded interviews of him/her (collectively, the "Presentation"). The permission granted includes the transcription and reproduction of the Presentation for inclusion in products sold or distributed by IEEE and live or recorded broadcast of the Presentation during or after the conference.
2. In connection with the permission granted in Section 1, the author hereby grants IEEE the unlimited, worldwide, irrevocable right to use his/her name, picture, likeness, voice and biographical information as part of the advertisement, distribution and sale of products incorporating the Work or Presentation, and releases IEEE from any claim based on right of privacy or publicity.

BY TYPING IN YOUR FULL NAME BELOW AND CLICKING THE SUBMIT BUTTON, YOU CERTIFY THAT SUCH ACTION CONSTITUTES



YOUR ELECTRONIC SIGNATURE TO THIS FORM IN ACCORDANCE WITH UNITED STATES LAW, WHICH AUTHORIZES ELECTRONIC SIGNATURE BY AUTHENTICATED REQUEST FROM A USER OVER THE INTERNET AS A VALID SUBSTITUTE FOR A WRITTEN SIGNATURE.

Parvez Mahbub	10-02-2023
Signature	**Date (dd-mm-yyyy)**

Information for Authors

AUTHOR RESPONSIBILITIES

The IEEE distributes its technical publications throughout the world and wants to ensure that the material submitted to its publications is properly available to the readership of those publications. Authors must ensure that their Work meets the requirements as stated in section 8.2.1 of the IEEE PSPB Operations Manual, including provisions covering originality, authorship, author responsibilities and author misconduct. More information on IEEE's publishing policies may be found at http://www.ieee.org/publications_standards/publications/rights/authorrightsresponsibilities.html Authors are advised especially of IEEE PSPB Operations Manual section 8.2.1.B12: "It is the responsibility of the authors, not the IEEE, to determine whether disclosure of their material requires the prior consent of other parties and, if so, to obtain it." Authors are also advised of IEEE PSPB Operations Manual section 8.1.1B: "Statements and opinions given in work published by the IEEE are the expression of the authors."

RETAINED RIGHTS/TERMS AND CONDITIONS

- Authors/employers retain all proprietary rights in any process, procedure, or article of manufacture described in the Work.
- Authors/employers may reproduce or authorize others to reproduce the Work, material extracted verbatim from the Work, or derivative works for the author's personal use or for company use, provided that the source and the IEEE copyright notice are indicated, the copies are not used in any way that implies IEEE endorsement of a product or service of any employer, and the copies themselves are not offered for sale.
- Although authors are permitted to re-use all or portions of the Work in other works, this does not include granting third-party requests for reprinting, republishing, or other types of re-use.The IEEE Intellectual Property Rights office must handle all such third-party requests.
- Authors whose work was performed under a grant from a government funding agency are free to fulfill any deposit mandates from that funding agency.

AUTHOR ONLINE USE

- **Personal Servers.** Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore. Authors shall not post the final, published versions of their papers.
- **Classroom or Internal Training Use.** An author is expressly permitted to post any portion of the accepted version of his/her own IEEE-copyrighted articles on the author's personal web site or the servers of the author's institution or company in connection with the author's teaching, training, or work responsibilities, provided that the appropriate copyright, credit, and reuse notices appear prominently with the posted material. Examples of permitted uses are lecture materials, course packs, e-reserves, conference presentations, or in-house training courses.
- **Electronic Preprints.** Before submitting an article to an IEEE publication, authors frequently post their manuscripts to their own web site, their employer's site, or to another server that invites constructive comment from colleagues. Upon submission of an article to IEEE, an author is required to transfer copyright in the article to IEEE, and the author must update any previously posted version of the article with a prominently displayed IEEE copyright notice. Upon publication of an article by the IEEE, the author must replace any previously posted electronic versions of the article with either (1) the full citation to the IEEE work with a Digital Object Identifier (DOI) or link to the article abstract in IEEE Xplore, or (2) the accepted version only (not the IEEE-published version), including the IEEE copyright notice and full citation, with a link to

B.2 Defectors in MSR 2023

Parvez Mahbub, Ohiduzzaman Shuvo, and M. Masudur Rahman. *Defectors: A Large, Diverse Python Dataset for Defect Prediction.* In Proceeding of The 20th International Conference on Mining Software Repositories (MSR 2023), pp. 393-397, Melbourne, Australia, May 2023.

Following is the copyright agreement with Institute of Electrical and Electronics Engineers.

IEEE COPYRIGHT AND CONSENT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

Defectors: A Large, Diverse Python Dataset for Defect Prediction

Parvez Mahbub, Ohiduzzaman Shuvo, Mohammad Masudur Rahman

2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)

COPYRIGHT TRANSFER

The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

GENERAL TERMS

1. The undersigned represents that he/she has the power and authority to make and execute this form.
2. The undersigned agrees to indemnify and hold harmless the IEEE from any damage or expense that may arise in the event of a breach of any of the warranties set forth above.
3. The undersigned agrees that publication with IEEE is subject to the policies and procedures of the IEEE PSPB Operations Manual.
4. In the event the above work is not accepted and published by the IEEE or is withdrawn by the author(s) before acceptance by the IEEE, the foregoing copyright transfer shall be null and void. In this case, IEEE will retain a copy of the manuscript for internal administrative/record-keeping purposes.
5. For jointly authored Works, all joint authors should sign, or one of the authors should sign as authorized agent for the others.
6. The author hereby warrants that the Work and Presentation (collectively, the "Materials") are original and that he/she is the author of the Materials. To the extent the Materials incorporate text passages, figures, data or other material from the works of others, the author has obtained any necessary permissions. Where necessary, the author has obtained all third party permissions and consents to grant the license above and has provided copies of such permissions and consents to IEEE

You have indicated that you DO wish to have video/audio recordings made of your conference presentation under terms and conditions set forth in "Consent and Release."

CONSENT AND RELEASE

1. In the event the author makes a presentation based upon the Work at a conference hosted or sponsored in whole or in part by the IEEE, the author, in consideration for his/her participation in the conference, hereby grants the IEEE the unlimited, worldwide, irrevocable permission to use, distribute, publish, license, exhibit, record, digitize, broadcast, reproduce and archive, in any format or medium, whether now known or hereafter developed: (a) his/her presentation and comments at the conference; (b) any written materials or multimedia files used in connection with his/her presentation; and (c) any recorded interviews of him/her (collectively, the "Presentation"). The permission granted includes the transcription and reproduction of the Presentation for inclusion in products sold or distributed by IEEE and live or recorded broadcast of the Presentation during or after the conference.
2. In connection with the permission granted in Section 1, the author hereby grants IEEE the unlimited, worldwide, irrevocable right to use his/her name, picture, likeness, voice and biographical information as part of the advertisement, distribution and sale of products incorporating the Work or Presentation, and releases IEEE from any claim based on right of privacy or publicity.

BY TYPING IN YOUR FULL NAME BELOW AND CLICKING THE SUBMIT BUTTON, YOU CERTIFY THAT SUCH ACTION CONSTITUTES

YOUR ELECTRONIC SIGNATURE TO THIS FORM IN ACCORDANCE WITH UNITED STATES LAW, WHICH AUTHORIZES ELECTRONIC SIGNATURE BY AUTHENTICATED REQUEST FROM A USER OVER THE INTERNET AS A VALID SUBSTITUTE FOR A WRITTEN SIGNATURE.

Parvez Mahbub	15-03-2023
Signature	**Date (dd-mm-yyyy)**

Information for Authors

AUTHOR RESPONSIBILITIES

The IEEE distributes its technical publications throughout the world and wants to ensure that the material submitted to its publications is properly available to the readership of those publications. Authors must ensure that their Work meets the requirements as stated in section 8.2.1 of the IEEE PSPB Operations Manual, including provisions covering originality, authorship, author responsibilities and author misconduct. More information on IEEE's publishing policies may be found at http://www.ieee.org/publications_standards/publications/rights/authorrightsresponsibilities.html Authors are advised especially of IEEE PSPB Operations Manual section 8.2.1.B12: "It is the responsibility of the authors, not the IEEE, to determine whether disclosure of their material requires the prior consent of other parties and, if so, to obtain it." Authors are also advised of IEEE PSPB Operations Manual section 8.1.1B: "Statements and opinions given in work published by the IEEE are the expression of the authors."

RETAINED RIGHTS/TERMS AND CONDITIONS

- Authors/employers retain all proprietary rights in any process, procedure, or article of manufacture described in the Work.
- Authors/employers may reproduce or authorize others to reproduce the Work, material extracted verbatim from the Work, or derivative works for the author's personal use or for company use, provided that the source and the IEEE copyright notice are indicated, the copies are not used in any way that implies IEEE endorsement of a product or service of any employer, and the copies themselves are not offered for sale.
- Although authors are permitted to re-use all or portions of the Work in other works, this does not include granting third-party requests for reprinting, republishing, or other types of re-use.The IEEE Intellectual Property Rights office must handle all such third-party requests.
- Authors whose work was performed under a grant from a government funding agency are free to fulfill any deposit mandates from that funding agency.

AUTHOR ONLINE USE

- **Personal Servers**. Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore. Authors shall not post the final, published versions of their papers.
- **Classroom or Internal Training Use.** An author is expressly permitted to post any portion of the accepted version of his/her own IEEE-copyrighted articles on the author's personal web site or the servers of the author's institution or company in connection with the author's teaching, training, or work responsibilities, provided that the appropriate copyright, credit, and reuse notices appear prominently with the posted material. Examples of permitted uses are lecture materials, course packs, e-reserves, conference presentations, or in-house training courses.
- **Electronic Preprints.** Before submitting an article to an IEEE publication, authors frequently post their manuscripts to their own web site, their employer's site, or to another server that invites constructive comment from colleagues. Upon submission of an article to IEEE, an author is required to transfer copyright in the article to IEEE, and the author must update any previously posted version of the article with a prominently displayed IEEE copyright notice. Upon publication of an article by the IEEE, the author must replace any previously posted electronic versions of the article with either (1) the full citation to the IEEE work with a Digital Object Identifier (DOI) or link to the article abstract in IEEE Xplore, or (2) the accepted version only (not the IEEE-published version), including the IEEE copyright notice and full citation, with a link to

B.3 Bugsplainer Tool in ICSME 2023

Parvez Mahbub, M. Masudur Rahman, Ohiduzzaman Shuvo, and Avinash Gopal. *Bugsplainer: Leveraging Code Structures to Explain Software Bugs with Neural Machine Translation.* In Proceeding of The 39th IEEE International Conference on Software Maintenance and Evolution (ICSME 2023), pp. 5, Bogota, Columbia, October 2023 (to appear).

Following is the copyright agreement with Institute of Electrical and Electronics Engineers.

IEEE COPYRIGHT AND CONSENT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

Bugsplainer: Leveraging Code Structures to Explain Software Bugs with Neural Machine Translation

Parvez Mahbub, Mohammad Masudur Rahman, Ohiduzzaman Shuvo, Avinash Gopal

2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)

COPYRIGHT TRANSFER

The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

GENERAL TERMS

1. The undersigned represents that he/she has the power and authority to make and execute this form.
2. The undersigned agrees to indemnify and hold harmless the IEEE from any damage or expense that may arise in the event of a breach of any of the warranties set forth above.
3. The undersigned agrees that publication with IEEE is subject to the policies and procedures of the IEEE PSPB Operations Manual.
4. In the event the above work is not accepted and published by the IEEE or is withdrawn by the author(s) before acceptance by the IEEE, the foregoing copyright transfer shall be null and void. In this case, IEEE will retain a copy of the manuscript for internal administrative/record-keeping purposes.
5. For jointly authored Works, all joint authors should sign, or one of the authors should sign as authorized agent for the others.
6. The author hereby warrants that the Work and Presentation (collectively, the "Materials") are original and that he/she is the author of the Materials. To the extent the Materials incorporate text passages, figures, data or other material from the works of others, the author has obtained any necessary permissions. Where necessary, the author has obtained all third party permissions and consents to grant the license above and has provided copies of such permissions and consents to IEEE

You have indicated that you DO wish to have video/audio recordings made of your conference presentation under terms and conditions set forth in "Consent and Release."

CONSENT AND RELEASE

1. In the event the author makes a presentation based upon the Work at a conference hosted or sponsored in whole or in part by the IEEE, the author, in consideration for his/her participation in the conference, hereby grants the IEEE the unlimited, worldwide, irrevocable permission to use, distribute, publish, license, exhibit, record, digitize, broadcast, reproduce and archive, in any format or medium, whether now known or hereafter developed: (a) his/her presentation and comments at the conference; (b) any written materials or multimedia files used in connection with his/her presentation; and (c) any recorded interviews of him/her (collectively, the "Presentation"). The permission granted includes the transcription and reproduction of the Presentation for inclusion in products sold or distributed by IEEE and live or recorded broadcast of the Presentation during or after the conference.
2. In connection with the permission granted in Section 1, the author hereby grants IEEE the unlimited, worldwide, irrevocable right to use his/her name, picture, likeness, voice and biographical information as part of the advertisement, distribution and sale of products incorporating the Work or Presentation, and releases IEEE from any claim based on right of privacy or publicity.

BY TYPING IN YOUR FULL NAME BELOW AND CLICKING THE SUBMIT BUTTON, YOU CERTIFY THAT SUCH ACTION CONSTITUTES

YOUR ELECTRONIC SIGNATURE TO THIS FORM IN ACCORDANCE WITH UNITED STATES LAW, WHICH AUTHORIZES ELECTRONIC SIGNATURE BY AUTHENTICATED REQUEST FROM A USER OVER THE INTERNET AS A VALID SUBSTITUTE FOR A WRITTEN SIGNATURE.

Parvez Mahbub	23-08-2023
Signature	**Date (dd-mm-yyyy)**

Information for Authors

AUTHOR RESPONSIBILITIES

The IEEE distributes its technical publications throughout the world and wants to ensure that the material submitted to its publications is properly available to the readership of those publications. Authors must ensure that their Work meets the requirements as stated in section 8.2.1 of the IEEE PSPB Operations Manual, including provisions covering originality, authorship, author responsibilities and author misconduct. More information on IEEE's publishing policies may be found at http://www.ieee.org/publications_standards/publications/rights/authorrightsresponsibilities.html Authors are advised especially of IEEE PSPB Operations Manual section 8.2.1.B12: "It is the responsibility of the authors, not the IEEE, to determine whether disclosure of their material requires the prior consent of other parties and, if so, to obtain it." Authors are also advised of IEEE PSPB Operations Manual section 8.1.1B: "Statements and opinions given in work published by the IEEE are the expression of the authors."

RETAINED RIGHTS/TERMS AND CONDITIONS

- Authors/employers retain all proprietary rights in any process, procedure, or article of manufacture described in the Work.
- Authors/employers may reproduce or authorize others to reproduce the Work, material extracted verbatim from the Work, or derivative works for the author's personal use or for company use, provided that the source and the IEEE copyright notice are indicated, the copies are not used in any way that implies IEEE endorsement of a product or service of any employer, and the copies themselves are not offered for sale.
- Although authors are permitted to re-use all or portions of the Work in other works, this does not include granting third-party requests for reprinting, republishing, or other types of re-use.The IEEE Intellectual Property Rights office must handle all such third-party requests.
- Authors whose work was performed under a grant from a government funding agency are free to fulfill any deposit mandates from that funding agency.

AUTHOR ONLINE USE

- **Personal Servers**. Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore. Authors shall not post the final, published versions of their papers.
- **Classroom or Internal Training Use.** An author is expressly permitted to post any portion of the accepted version of his/her own IEEE-copyrighted articles on the author's personal web site or the servers of the author's institution or company in connection with the author's teaching, training, or work responsibilities, provided that the appropriate copyright, credit, and reuse notices appear prominently with the posted material. Examples of permitted uses are lecture materials, course packs, e-reserves, conference presentations, or in-house training courses.
- **Electronic Preprints.** Before submitting an article to an IEEE publication, authors frequently post their manuscripts to their own web site, their employer's site, or to another server that invites constructive comment from colleagues. Upon submission of an article to IEEE, an author is required to transfer copyright in the article to IEEE, and the author must update any previously posted version of the article with a prominently displayed IEEE copyright notice. Upon publication of an article by the IEEE, the author must replace any previously posted electronic versions of the article with either (1) the full citation to the IEEE work with a Digital Object Identifier (DOI) or link to the article abstract in IEEE Xplore, or (2) the accepted version only (not the IEEE-published version), including the IEEE copyright notice and full citation, with a link to

YOUR ELECTRONIC SIGNATURE TO THIS FORM IN ACCORDANCE WITH UNITED STATES LAW, WHICH AUTHORIZES ELECTRONIC SIGNATURE BY AUTHENTICATED REQUEST FROM A USER OVER THE INTERNET AS A VALID SUBSTITUTE FOR A WRITTEN SIGNATURE.

Parvez Mahbub	23-08-2023
Signature	**Date (dd-mm-yyyy)**

Information for Authors

AUTHOR RESPONSIBILITIES

The IEEE distributes its technical publications throughout the world and wants to ensure that the material submitted to its publications is properly available to the readership of those publications. Authors must ensure that their Work meets the requirements as stated in section 8.2.1 of the IEEE PSPB Operations Manual, including provisions covering originality, authorship, author responsibilities and author misconduct. More information on IEEE's publishing policies may be found at http://www.ieee.org/publications_standards/publications/rights/authorrightsresponsibilities.html Authors are advised especially of IEEE PSPB Operations Manual section 8.2.1.B12: "It is the responsibility of the authors, not the IEEE, to determine whether disclosure of their material requires the prior consent of other parties and, if so, to obtain it." Authors are also advised of IEEE PSPB Operations Manual section 8.1.1B: "Statements and opinions given in work published by the IEEE are the expression of the authors."

RETAINED RIGHTS/TERMS AND CONDITIONS

- Authors/employers retain all proprietary rights in any process, procedure, or article of manufacture described in the Work.
- Authors/employers may reproduce or authorize others to reproduce the Work, material extracted verbatim from the Work, or derivative works for the author's personal use or for company use, provided that the source and the IEEE copyright notice are indicated, the copies are not used in any way that implies IEEE endorsement of a product or service of any employer, and the copies themselves are not offered for sale.
- Although authors are permitted to re-use all or portions of the Work in other works, this does not include granting third-party requests for reprinting, republishing, or other types of re-use.The IEEE Intellectual Property Rights office must handle all such third-party requests.
- Authors whose work was performed under a grant from a government funding agency are free to fulfill any deposit mandates from that funding agency.

AUTHOR ONLINE USE

- **Personal Servers**. Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore. Authors shall not post the final, published versions of their papers.
- **Classroom or Internal Training Use.** An author is expressly permitted to post any portion of the accepted version of his/her own IEEE-copyrighted articles on the author's personal web site or the servers of the author's institution or company in connection with the author's teaching, training, or work responsibilities, provided that the appropriate copyright, credit, and reuse notices appear prominently with the posted material. Examples of permitted uses are lecture materials, course packs, e-reserves, conference presentations, or in-house training courses.
- **Electronic Preprints.** Before submitting an article to an IEEE publication, authors frequently post their manuscripts to their own web site, their employer's site, or to another server that invites constructive comment from colleagues. Upon submission of an article to IEEE, an author is required to transfer copyright in the article to IEEE, and the author must update any previously posted version of the article with a prominently displayed IEEE copyright notice. Upon publication of an article by the IEEE, the author must replace any previously posted electronic versions of the article with either (1) the full citation to the IEEE work with a Digital Object Identifier (DOI) or link to the article abstract in IEEE Xplore, or (2) the accepted version only (not the IEEE-published version), including the IEEE copyright notice and full citation, with a link to