Beyond Natural Language Processing: Advancing Software Engineering Tasks through Code Structure

by

Zishuo Ding

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

Waterloo, Ontario, Canada, 2024

Examining Committee Membership

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

External Examiner:	Alexander Serebrenik Full Professor Department of Mathematics and Computer Science Eindhoven University of Technology, The Netherlands
Supervisor:	Weiyi Shang Associate Professor Department of Electrical and Computer Engineering University of Waterloo
Internal Members:	Derek Rayside Associate Professor Department of Electrical and Computer Engineering University of Waterloo
	Ladan Tahvildari Professor Department of Electrical and Computer Engineering University of Waterloo
Internal-External Member:	Shane McIntosh Associate Professor David R. Cheriton School of Computer Science University of Waterloo

Author's Declaration

Abstract

Machine learning-based approaches have been widely used to address natural language processing (NLP) problems. Considering the similarities between natural language text and source code, researchers have been working on the application of NLP techniques to code-related tasks. However, it is crucial to acknowledge that source code and natural language are different by their natures. For example, source code is highly structured and executable; while NLP techniques may not understand the structure of source code. As a result, applying NLP techniques directly may not yield optimal results, and effectively adapting these techniques to suit software engineering tasks remains a significant challenge.

To tackle this challenge, in this thesis, we focus on two important intersections between the source code and natural language text: (1) learning and evaluating distributed code representations (i.e., code embeddings), which plays a fundamental role in numerous software engineering tasks, especially in the era of deep learning, and (2) improving the textual information in logging statements (i.e., logging texts), which record useful information (i.e., logs) to support various software engineering activities.

For distributed code representations, we first conduct a comprehensive survey of existing code embedding techniques. This survey encompasses techniques borrowed from NLP, as well as those specifically tailored for source code. We also identify six downstream software engineering tasks to evaluate the effectiveness of the learned code embeddings. Moreover, based on our analysis of existing code embedding techniques, we propose a novel approach to learn more generalizable code embeddings in a task-agnostic manner. This approach represents source code as graphs and leverages Graph Convolutional Networks to learn code embeddings that exhibit greater generalizability.

For the textual information in logging statements, we propose to improve the current logging practices from two aspects: (1) proactively suggesting the generation of new logging texts: we propose automated deep learning-based approaches that generate logging texts by translating the related source code into short textual descriptions; (2) retroactively analyzing existing logging texts: we make the first attempt to comprehensively study the temporal relations between logging and its corresponding source code, which is later successfully used to detect anti-patterns in existing logging statements.

Based on the experimental results on the subject systems, we anticipate that our work can offer valuable suggestions and support to developers, aiding them in the effective utilization of NLP techniques for software engineering tasks.

Acknowledgements

I would like to take this opportunity to extend my deep gratitude to all those who have supported me throughout my Ph.D. journey.

First, I would like to thank my advisor, Dr. Weiyi Shang, for his guidance, encouragement, and constructive suggestions not only on my research but also on my career development. I particularly appreciate his supervising style, which starts with meticulous guidance, seamlessly transitions into providing higher-level advice, and ultimately affords full flexibility and support for exploring new directions. My research was only possible due to his invaluable help and inspirational guidance.

I would also like to sincerely thank my examination committee members, Dr. Derek Rayside, Dr. Ladan Tahvildari, Dr. Shane McIntosh, and Dr. Alexander Serebrenik for dedicating their valuable time to review my work and provide invaluable insights.

In addition, I want to thank all of my lab mates and collaborators, for their support and encouragement, also for the wonderful moments we worked and enjoyed together.

Finally, I want to express my deepest gratitude to my parents. I would never have reached this point without their boundless love and unwavering support. Having them together with me is the greatest source of pride in my life.

Dedication

This is dedicated to all my teachers. Thank you for imparting your knowledge to us. In the walk of every student's life, you are the motivation, the creativity behind our ideas, and the source of our knowledge.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine learning-based approaches have been widely used to address natural language processing (NLP) problems. Considering the similarities between natural language and programming language, researchers have been working on applying techniques from NLP to deal with code [8]. For instance, Feng et al. [44] propose CodeBERT, which directly employs the BERT model from NLP to source code to deal with software engineering tasks, such as natural language code search and code documentation generation. Nevertheless, the direct application of NLP techniques may not yield optimal results, given the fundamental differences between source code and natural language text. For example, code is executable and has formal syntax, which may not be effectively captured by techniques designed primarily for natural language tasks. Consequently, optimizing these NLP techniques to better adapt to software engineering tasks remains a challenging task.

In recent times, researchers have proposed different approaches to encoding code-specific properties. For example, Guo et al. [53] further enhance CodeBERT to create GraphCodeBERT by leveraging the data flow information of source code. Among the recent advancements in these techniques, we focus on two research areas: (1) **distributed code representations**, and (2) **the textual information in logging statements**, which we believe are two important intersections between the natural language and source code.

Distributed code representations, also called **code embeddings**[1], project code tokens into a low-dimensional semantic space, where each token is represented by a vector of real numbers. In the era of deep learning, code embeddings play a fundamental role in various software engineering (SE) tasks, including automatic program repair [29, 178, 185], software vulnerability prediction [57, 144], and code clone detection [21].

[1]Throughout this thesis, unless explicitly specified, by code embeddings we often refer to distributed code representations.

Researchers have explored several approaches for representing source code into vectors in SE tasks. Some have directly applied word embedding techniques from NLP to source code to produce representations for code tokens. While this direct application of NLP techniques provides satisfactory results, it still comes with limitations, as programming languages have unique properties (e.g., code is executable and has formal syntax.) that may not be fully captured by existing NLP techniques. Therefore, the optimization of word embedding techniques to encode the unique information in the source code remains a challenge [8].

The textual information in logging statements is another important intersection between the source code and natural language text. Figure 1.1 provides an example logging statement from HBase, which contains a textual description (i.e., logging text[2]), "`Failed to create dir`". The logging texts are usually written by developers to record system execution behaviors. Logging statements execute and produce useful information (i.e., logs) while systems are running. These logs are used in various software engineering activities, such as failure diagnosis and system monitoring [13, 89, 197].

```
LOG.warn("Failed to create dir {}", dst)
```

Figure 1.1: An example of logging statement from HBase.

Extensive prior research has demonstrated that writing appropriate logging statements is an important yet challenging task [24, 157, 199, 202]. This challenge is particularly evident when it comes to determining and refining the texts within logging statements [23, 59]. Therefore, more research efforts on logging texts are needed to develop techniques that can better support developers in their daily tasks of writing proper logging statements.

Given the significance of code embeddings and logging texts in software engineering activities, as well as the aforementioned challenges, this thesis focuses on these two areas as a first step. On this basis, we aim to conduct studies and propose approaches to offer valuable suggestions and support to developers, aiding them in the effective utilization of NLP techniques for software engineering tasks.

Chapter organization. The rest of this chapter is organized as follows: We present the research hypothesis in Section 1.1. Section 1.2 gives an overview and the organization of the thesis. In Section 1.3, we discuss the contributions of this thesis.

[2]Throughout this thesis, unless explicitly specified, by logging texts we often refer to the textual information in logging statements.

1.1 Research Hypothesis

Our research hypothesis is as follows:

> Given the similarities between source code and natural language text, natural language processing techniques can be applied directly to source code. However, recognizing the inherent differences between code and natural language, the advancement of code-specific approaches holds the potential for further enhancing downstream Software Engineering tasks.

The goal of this thesis is to investigate how code-specific properties influence the performance of SE tasks and to explore effective strategies for optimizing NLP techniques to encode such features extracted from code. We will validate our research hypothesis through two parts: **Part I - Learning and evaluating distributed code representations** and **Part II - Improving the textual information in logging statements**.

1.2 Thesis Overview

Figure 1.2 provides an overview of the work presented in this thesis, which is divided into two parts based on the research hypothesis.

Part I focuses on the learning and evaluating of distributed code representations. Detailedly,

In Chapter 2, we first give a background, which discusses the training context (Figure 1.2(a)) and some representative techniques used for learning code embeddings (Figure 1.2(b)). We then present an overview of the recent research that is related to the field of distributed code representations as well as their limitations.

In Chapter 3, we revisit the use of pre-trained code embeddings for downstream software engineering tasks. In this chapter, we start off by conducting a comprehensive survey of existing code embedding techniques, and then we talk about the identified six downstream SE tasks (Figure 1.2(c)) that are used to assess the effectiveness of using pre-trained code embeddings in SE tasks. We also propose a framework, StrucTexVec, which uses a two-step unsupervised training strategy to incorporate

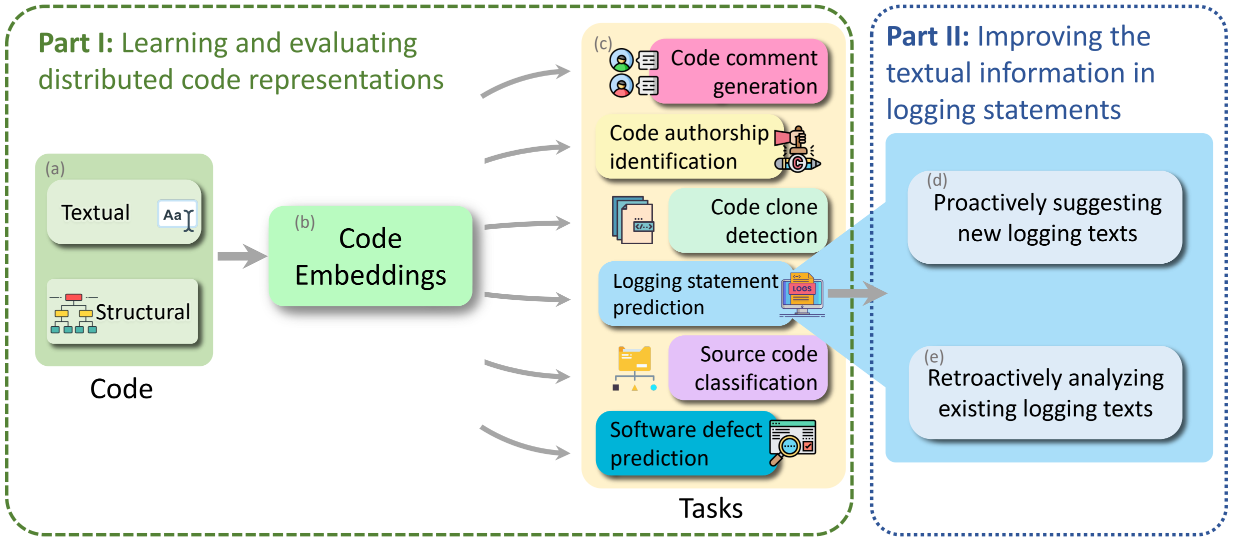

Part I: Learning and evaluating distributed code representations
(a)
Textual
Structural
Code
(b)
Code Embeddings
(c)
Code comment generation
Code authorship identification
Code clone detection
Logging statement prediction
Source code classification
Software defect prediction
Tasks
Part II: Improving the textual information in logging statements
(d)
Proactively suggesting new logging texts
(e)
Retroactively analyzing existing logging texts

Figure 1.2: An overview of the work presented in this thesis.

the textual and structural information of the code (Figure 1.2(a)). Finally, we present our results: (1) different embedding techniques can result in diverse performance for some SE tasks; (2) using well-pre-trained embeddings usually improves the performance of SE tasks; (3) the structural context has a non-negligible impact on improving the quality of code embeddings.

In Chapter 4, we propose our approach to learning generalizable code embeddings that can benefit different downstream SE tasks. Based on the findings in Chapter 3 (e.g., the lack of generalizability of existing embeddings, and the significance of the structural information from the source code), we propose GraphCodeVec, which represents the source code as graphs and leverages the Graph Convolutional Networks to capture the structural information. We find that GraphCodeVec outperforms all the baselines in five out of the six downstream tasks and its performance is relatively stable across different tasks and datasets.

Following our exploration of fundamental code embeddings, we shift our research focus to improving the textual information in logging statements, which constitutes Part II in Figure 1.2. Detailedly,

In Chapter 5, we discuss two types of research related to improving the current logging practices: (1) automated logging suggestions and (2) empirical studies on software logging. These align well with our proactive suggestions for new logging texts

(Figure 1.2(d)) and retroactive analysis of existing logging texts (Figure 1.2(e)), respectively.

In Chapter 6, we present our work of proactively suggesting the generation of new logging texts (Figure 1.2(d)). We first formulate this problem into a neural machine translation task and propose LoGenText, an automated approach that generates logging texts by translating the related source code into short textual descriptions. Furthermore, we extend LoGenText to LoGenText-Plus by incorporating the syntactic templates of the logging texts. Different from LoGenText, LoGenText-Plus decomposes the logging text generation process into two stages and tries to solve the problem with a coarse-to-fine strategy. Both the quantitative and human evaluations indicate the superiority of our proposed approaches.

In Chapter 7, we discuss our research on retroactively analyzing existing logging texts (Figure 1.2(e)). Specifically, we make the first attempt to comprehensively study the temporal relations between logging and its corresponding source code. Then, based on these findings, we propose a tool named TempoLo to automatically detect the issues of temporal inconsistencies between logging and code. This work lays the foundation for describing temporal relations between logging and code and demonstrates the potential for a deeper understanding of the relationship between logging and code.

Finally, in Chapter 8, we conclude the thesis and discuss future directions.

1.3 Thesis Contributions

This work presents several novel contributions aimed at providing valuable guidance and support to developers in optimizing NLP techniques for use in software engineering (SE) tasks.

- First, we revisit and extend previous work by evaluating more embedding techniques across a wider variety of downstream SE tasks. We find that models using pre-trained embeddings can perform better than models without pre-trained embeddings, and there does not exist an embedding technique that consistently outperforms others across all or even the majority of the tasks. Moreover, the experimental results show that the structural information has a non-negligible impact on the quality of pre-trained code embeddings.

- Then, based on our analysis of existing code embedding techniques (i.e., the generalization issues of the existing techniques and the stronger impact of the structural information), we propose to represent the source code as graphs and leverage the Graph Convolutional Networks to capture the structural information, aiming to learn more generalizable code embeddings in a task-agnostic manner. Our results show that our proposed approach generally outperforms the baseline approaches and can produce more stable results.

- Third, following the distributed code representation, we shift our research focus to improving the textual information in logging statements, as it is another important intersection between the source code and natural language text. We make efforts to proactively suggest the generation of new logging texts. In this part, we formulate the automatic logging text generation problem as a neural machine translation task, where we identify the source input and target output, and confirm the validity of our findings in our research questions that leverage the quantitative metrics. Moreover, we also show the effectiveness of incorpoarating the structural information using the multi-encoders architecture.

- Finally, we make the first attempt to model the temporal relationships between the existing logging texts and their corresponding code. We optimize the temporal relations from NLP and propose the logical and semantic temporal relations for source code. We also derive rules to detect the logical and semantic temporal relations between logging and code, as well as rules to detect logging-code temporal inconsistencies. The rules are further implemented as a tool, which has been successfully used to detect the temporal issues and thus improve the quality of the logging activities.

Part I

Learning and Evaluating Distributed Code Representations

Chapter 2

Background and Related Work

In this chapter, we discuss the background related to **distributed code representations**. We then talk about the prior research that proposes and applies code embeddings in software engineering tasks.

2.1 Background

In this section, we present a background of existing code embedding techniques. We begin by introducing the training context, followed by an examination of the code embedding techniques evaluated in this thesis.

2.1.1 Training Context

In this part, we introduce two types of training contexts for code embeddings (1) textual context, which is the plain text of the source code, and (2) structural context, which refers to the abstract syntax trees (ASTs) of the source code.

Textual context Like natural languages, programming languages are repetitive and predictable [61], and thus researchers [19, 43, 166] consider source code as plain text and directly apply existing word embedding techniques to source code. More specifically, to make source code suitable for embedding training, the source code is usually tokenized into a sequence of tokens [19, 43, 166]. For example, the following source code[1] in Figure 2.1,

[1]https://commons.apache.org/proper/commons-io/javadocs/api-2.5/src-html/org/apache/commons/io/filefilter/AndFileFilter.html

after tokenization, is converted to a sequence of code tokens, shown in Figure 2.2. The

```
public boolean accept(final File file) {
        if (this.fileFilters.isEmpty()) {
            return false;
        }
        for (final IOFileFilter fileFilter : fileFilters) {
            if (!fileFilter.accept(file)) {
                return false;
            }
        }
        return true;
}
```

Figure 2.1: Code snippet from Apache Commons project.

generated token sequence is used as the training corpus and finally fed into embedding techniques.

```
public boolean accept ( final File file ) { ... file ) ) { return false ; } } return true ; }
```

Figure 2.2: Code tokens after tokenization.

Structural context Another representation of source code is the abstract syntax tree (AST). AST represents source code with a tree structure. Figure 2.3 is the AST representation of the code snippet in Figure 2.1, where the leaf nodes of the tree are the tokens from the source code, while the non-leaf nodes are a set of AST node types that provide the syntax structure of the code. Due to its ability to capture not only the lexical information but also the syntactic structure of source code, AST has proven to be useful in a wide range of software engineering tasks, including code embeddings [11, 169, 204]. For example, Alon et al. [11] take the AST nodes as input and train a path-attention network for generating code embeddings.

2.1.2 Embedding Learning Techniques

With the rapid development of deep learning in SE applications, various distributed code representation techniques have been proposed, which can be categorized into two broad categories (1) non-contextual embeddings (e.g., Word2vec, GloVe, etc.), which learn unique

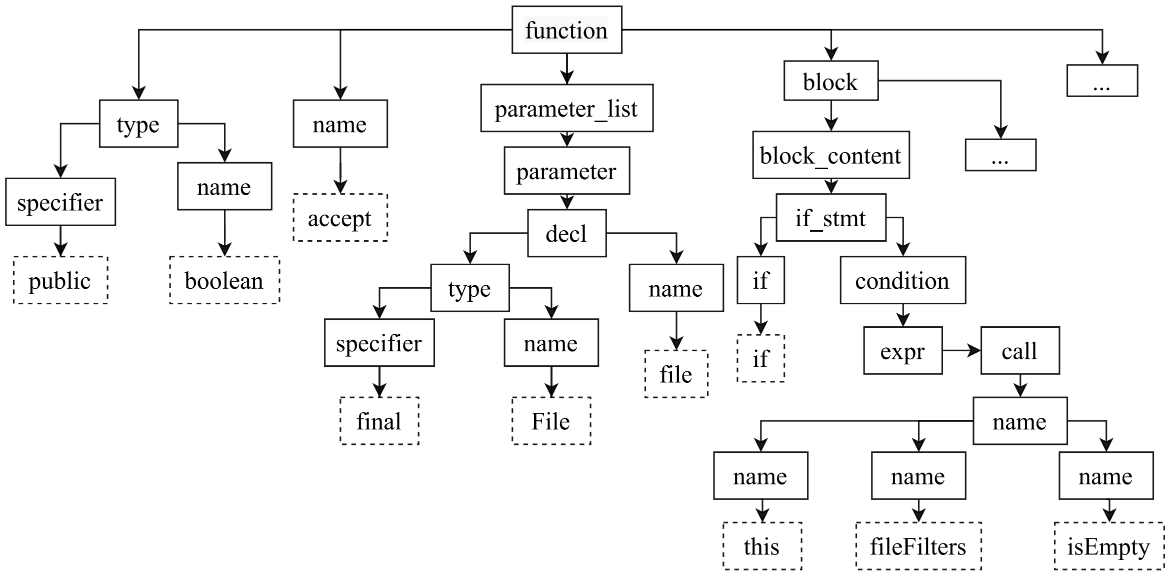


Figure 2.3: The AST of the code snippet from Figure 2.1. The leaf nodes of the tree are the tokens from the source code, while the non-leaf nodes are a set of AST node types that provide the syntax structure of the code.

fixed representations for tokens in the vocabulary without considering the meanings of tokens in different contexts, and (2) contextual embeddings (e.g., CodeBERT and CuBERT), which are generally obtained from the transformer-based models and the representations of tokens are adjusted based on different contexts. In this section, we first introduce the two categories and then describe the existing embedding learning techniques (i.e., Word2vec, GloVe, fastText, code2vec, CodeBERT, and CuBERT) that are evaluated in this work. Table 2.1 summarizes these techniques in more detail.

2.1.2.1 Non-contextual embeddings

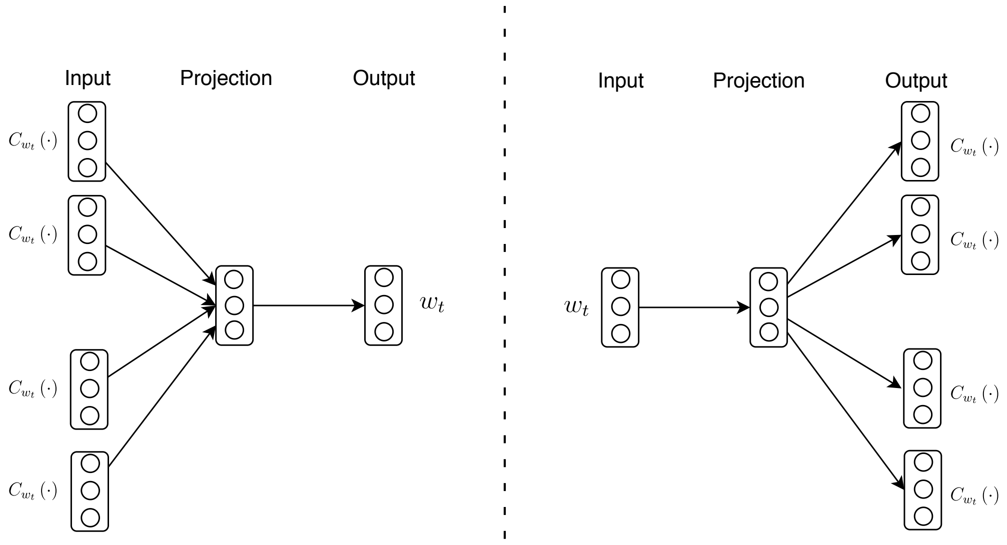
Non-contextual embeddings map source code tokens into a low-dimensional semantic space, where each code token is assigned with a unique real-valued vector. Non-contextual embeddings act as a static look-up table $\mathbf{E} \in |V| \times \mathbb{R}^d$ to map a token in the vocabulary, V to a d-dimensional vector. The embeddings are usually learned from a large corpus and can be applied to downstream tasks to either initialize the weights of the embedding layer (i.e., the input layer) of deep learning models or be used as feature vectors for traditional machine learning models. In this section, we introduce several state-of-the-art distributed code representation approaches in detail.

Table 2.1: Summary of different embedding techniques.

Technique	Category	Task type	Corpus type	Corpus level	Description
Word2vec	Non-Contextual	Unsupervised	Code as plain text	Token	Word2vec considers source code as plain text and adopts simple neural networks to learn the embeddings based on the tokens within a local context window, and thus, it may fail to take the advantage of 1) some global information, 2) the character level information, and 3) the structural information (e.g., AST).
GloVe					GloVe also considers source code as plain text and learns the embeddings based on the global word-word co-occurrence statistics. However, it only considers the token level information and ignores the character level information as well as the structural information (e.g., AST).
fastText				Character	fastText extends the skip-gram model (cf. Section 2.1.2.1) and takes into account subword information. However, it still ignores the global and structural information.
code2vec		Supervised	AST	Token	code2vec is a supervised approach which means to guarantee good results, it requires human-labeled data for training. Besides, code2vec is trained on the task of method name prediction and thus, the code embeddings produced by code2vec is task-specific and may not generalize well to other downstream tasks.
CodeBERT	Contextual	Unsupervised	Code as plain text & code documentation	Character	Both CodeBERT and CuBERT are transformer-based models and learn the context-sensitive representations of tokens. However, there may still exist limitations for these two models. The first one is that they both treat the source code as plain text and do not consider the structural information explicitly. Another limitation is the high computation cost for training such huge models (both have millions of parameters), making it almost impossible for us to modify and re-train the models by ourselves.
CuBERT					

Word2vec Word2vec has become popular in software engineering tasks [204] due to its high efficiency. In particular, Word2vec [123, 124] has two model architectures: Continuous Bag-of-Words (CBOW) and skip-gram, both consider the source code as plain text and take the textual context as input for training the embeddings.

An illustration of the CBOW and skip-gram models are shown in Figure 2.4, where w_t is one target token from the vocabulary, and C_{w_t} are the context tokens of the target word w_t.



(a) CBOW (b) skip-gram

Figure 2.4: An illustration of CBOW and skip-gram models architecture.

Continuous Bag-of-Words model. The CBOW model tries to predict the target token by considering its context within the local window. Formally, given a sequence of tokens $\mathcal{D}$, and w_t is the t^{th} token (i.e., target token) in the corpus, the objective of the model is to maximize the following objective function:

$$\mathcal{L} = \sum_{w_t \in \mathcal{D}} \log p\left(w_t | \mathcal{C}_{w_t}\right) \tag{2.1}$$

where $\mathcal{C}_{w_t}$ are the local context tokens of the target token w_t, and p is the conditional probability of generating the central target token w_t from given context tokens $\mathcal{C}_{w_t}$.

Skip-gram model. As Figure 2.4b shows, the skip-gram model shares a similar architecture with the CBOW model. Rather than predicting the target token based on the local context, it tries to predict the context tokens based on the target token. Thus, the objective of this model becomes to maximize the function:

$$\mathcal{L} = \sum_{w_t \in \mathcal{D}} \sum_{w_c \in \mathcal{C}_{w_t}} \log p\left(w_c | w_t\right) \tag{2.2}$$

The conditional probability, $p\left(w_c | w_t\right)$ is defined using the following softmax function:

$$p\left(w_c | w_t\right) = \frac{\exp(\mathbf{V}_{w_t}{}^{\intercal}\mathbf{U}_{w_c})}{\sum_{w \in \mathcal{D}} \exp(\mathbf{V}_{w_t}{}^{\intercal}\mathbf{U}_{w})} \tag{2.3}$$

where $\mathbf{V}_w$ and $\mathbf{U}_w$ denote the "input" and "output" vectors of token w in the vocabulary $\mathcal{D}$, respectively.

Limitations. As discussed above, Word2vec is one of the shallow window-based methods, which adopts simple neural networks to learn the embeddings based on the tokens within a local context window, and thus, it may fail to take advantage of some global information. For example, considering the preprocessed source code in Figure 2.2, assuming the window size is 10, and the target token is the method name "`accept`", then the first returned value "`false`" can be captured as one of the context tokens by the window. However, the last returned value "`true`" is missed as its distance to the target token is beyond the window size, which is not in accord with the programming rules: both should be the context tokens of "`accept`", as they are the returned values, which should be directly connected to the method.

GloVe To better capture the global statistic information of the training corpus, Pennington et al. [142] propose GloVe (Global Vectors for Word Representation). GloVe is also an unsupervised embedding learning algorithm and uses token-token co-occurrence statistics to obtain vector representations for source code tokens. Similar to Word2vec, in this model, the source code is treated as plain text, and the textual context is considered as the training context. The goal of GloVe is to minimize the following weighted least squares errors:

$$J = \sum_{i,j=1}^{|\mathcal{D}|} f\left(X_{i,j}\right) \left(w_i^T \tilde{w}_j - \log X_{i,j}\right)^2 \tag{2.4}$$

where X denotes the matrix of token-token co-occurrence counts, $f\left(X_{i,j}\right)$ is a weighting function, w_i and $\tilde{w}_j$ are word and separate context word vectors, respectively.

Limitations. Models like Word2vec or GloVe learn the embeddings at the token level. Although they can effectively capture the semantic properties of different tokens, they ignore the character level information. Considering the naming conventions for variables/methods in programming languages, for example, the Java camel cases or lowercase with tokens separated by underscores in Python, the insufficient use of the character level information is a non-negligible limitation for the embedding techniques that work at the token level. Besides, these techniques also suffer from the out-of-vocabulary (OOV) problem for the tokens that do not appear in the existing training corpus, especially for programming languages, as developers usually combine different tokens together as one.

fastText fastText[2] [19] is a recently proposed technique that exploits the internal structure of words (i.e., character level information), and tries to tackle the OOV problem by using character level units. In particular, fastText also considers the source code as plain text, where each token is represented by a set of n-grams appearing in this token. It then learns representations for the character n-grams and represents words as the sum of the character n-gram vectors. fastText extends the skip-gram model (cf. Section 2.1.2.1) by using a new softmax function which takes into account the subword representation. Similar to skip-gram, the goal of fastTest is also to maximize the function:

$$\mathcal{L} = \sum_{w_t \in \mathcal{D}} \sum_{w_c \in \mathcal{C}_{w_t}} \log p\left(w_c | w_t\right) \tag{2.5}$$

where $\mathcal{D}$ is the given sequence of tokens, $\mathcal{C}_{w_t}$ is the local context tokens of the target token w_t. The conditional probability, $p\left(w_c | w_t\right)$ is defined using the following softmax function:

$$p\left(w_c \mid w_t\right) = \frac{e^{s(w_t, w_c)}}{\sum_{w \in \mathcal{D}} e^{s(w_t, w)}} \tag{2.6}$$

$$s(w_t, w_c) = \sum_{g \in \mathcal{G}_{w_t}} \mathbf{z}_g^{\top} \mathbf{v}_{w_c} \tag{2.7}$$

where $s\left(w_t, w_c\right)$ is a score function, $\mathcal{G}_{w_t}$ is the set of n-grams appearing in w_t, $\mathbf{z}_g$ and $\mathbf{v}_{w_c}$ are vectors of the n-gram g and w_c, respectively.

Limitations. Although fastText has the ability to capture the character level information, the three embedding techniques discussed above still consider source code as plain text and take the textual context for embedding training. However, source code is by nature different from plain text, as it also contains structural information, which may be helpful for generating distributed representations of code tokens.

[2]https://github.com/facebookresearch/fastText

Code2vec Code2vec is a recently released code representation model by Alon et al. [11] that takes into account the structural context (i.e., the abstract syntax tree representation of the source code). Given the AST representation of a code snippet, code2vec collects the paths between every two AST leaf nodes, and thus it represents the code snippet as a bag of paths. Code2vec then employs a path-attention network with fully connected layers to learn vector representations of the tokens and method names. The embeddings are trained in a supervised process with the objective to minimize the cross-entropy loss of predicting method names,

$$\mathcal{L}(p\|q) = -\sum_{y \in Y} p(y) \log q(y) \tag{2.8}$$

where, $p(y)$ is the distribution of the ground truth, if y is the true label of the case, then $p(y) = 1$, and 0 otherwise (i.e., binary indicator of whether y is the actual label.), $q(y)$ is the predicted probability.

Limitations. As code2vec is a supervised approach which means to guarantee good results, it requires human-labeled data for training. Besides, it is trained on the task of method name prediction, and thus, the code embeddings produced by code2vec are task-specific and may not generalize well to other downstream tasks [72].

Limitations of non-contextual embeddings. Non-contextual embeddings (e.g., Word2vec and GloVe) have been playing an important role for improving the results of downstream tasks. Despite the powerful ability to represent source code tokens into vectors, these non-contextual embeddings also have their limitations: they are context-independent and assign each token with a single static representation. Therefore, they cannot effectively capture the different nuances of the same token in different contexts. For example, given the following code snippet, non-contextual embeddings assign the keyword "`public`" with

```
public class Accept{
    ...
    public int a;
    ...
    public boolean accept(final File file) {
        ...
    }
}
```

Figure 2.5: An example code snippet with public deceleration for class, method, and variable.

only one unique vector, despite that they appear three times with different functionalities.

However, considering the fact that they are modifiers for different levels of source code (i.e., class, attribute, and method), they should have different representations to better capture the properties.

2.1.2.2 Contextual embeddings

To address the limitations of non-contextual embeddings, researchers recently proposed several methods to learn the context-dependent code embeddings (also known as PLM, short for pre-trained language models), such as CodeBERT and CuBERT. Unlike static code embeddings, contextual embeddings are dynamic representations of tokens, that is the same token can have different representations based on the different surrounding context. For example, the three "`public`" keywords in Figure 2.5 would be assigned different vectors with respect to their different contexts. Recall that non-contextual embeddings act as a look-up table where each row of the real numbers is the vector representation of the corresponding token. However, for contextual embeddings, they are actually complicated transformer-based models, which take a sequence of code tokens as input and can return a set of fine-tuned embeddings for each token respectively. Formally, given a target token, w_t, together with the whole code snippet where it appears, $w_1, \cdots, w_t, \cdots, w_n$, the adjusted vector for w_t is

$$[\mathbf{v}_{w_1}, \cdots, \mathbf{v}_{w_t}, \cdots \mathbf{v}_{w_n}] = f(w_1, \cdots, w_t, \cdots, w_n) \tag{2.9}$$

where f is the pre-trained model with millions of parameters.

The pre-trained models (contextual embeddings) are usually trained on a general and large corpus and can be specialized to different downstream tasks by adding a task-specific layer and fine-tuning the parameters based on the training dataset of the specified task. In our work, to make the contextual embeddings suitable for our downstream tasks, we extract the embedding layer of the model, save the weights into the Word2vec format, and then use them as described in Section 2.1.2.1.

In this section, we first introduce the BERT architecture as a background and then describe two recent applications of BERT on learning contextual embeddings for source code (i.e., CodeBERT and CuBERT).

BERT The contextual embeddings are popularized by BERT (Bidirectional Encoder Representation from Transformer) [35]. As shown in Figure 2.6, BERT uses the bidirectional Transformer encoder which can effectively exploit both the left and right contexts of a target token.

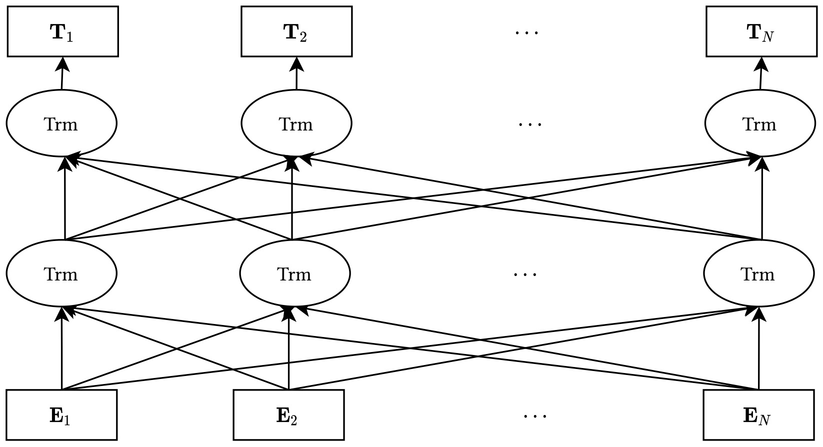


Figure 2.6: The overall architecture of BERT. **E** and **T** are the input and output vectors respectively, and Trm is the encoder of Transformer.

In the work of Devlin et al. [35], the authors present two main models: $\mathbf{BERT_{BASE}}$ which has a total number of 110M parameters and $\mathbf{BERT_{LARGE}}$ with 340M parameters. To effectively learn the model parameters, two objectives are designed for BERT: masked language model (MLM) and next sentence prediction (NSP).

Unlike the most common language model, which uses the previous sequence of tokens to predict the next token (the loss function is shown in Eq. 2.10):

$$\mathcal{L} = -\sum_{w_t \in \mathcal{D}} \log p\left(w_t \mid w_1, \cdots, w_{t-1}\right) \tag{2.10}$$

in masked language model, some of the tokens in a sequence are randomly masked and the goal is to predict these masked tokens based on their surrounding unmasked context tokens:

$$\mathcal{L} = -\sum_{w_t \in \mathcal{M}} \log p\left(w_t \mid w_1, \cdots, w_{t-1}, w_{t+1}, \cdots, w_N\right) \tag{2.11}$$

where $\mathcal{M}$ represents the masked tokens and $w_1, \cdots, w_{t-1}, w_{t+1}, \cdots, w_N$ represent the rest of tokens in the sequence.

BERT also utilizes the next sentence prediction as the second objective to capture relationships between sentences for some sentence-based downstream tasks (e.g., question answering (QA)). In this task, the goal is to predict whether the sentence is the next sentence of the current:

$$\mathcal{L} = -\log p(y \mid \mathbf{s_t}, \mathbf{s_{t+1}}) \tag{2.12}$$

where $y = 1$ if $\mathbf{s_{t+1}}$ is the next sentence of $\mathbf{s_t}$ and $y = 0$ otherwise.

To apply BERT to downstream tasks, Devlin et al. [35] also propose to use the two-step training strategy (1) pre-training on unlabeled data and (2) fine-tuning using labeled data from the downstream tasks.

Since the release of BERT, researchers have made a few changes to the original model and achieved continuous improvements. For example, Liu et al. [106] find that training the BERT model without the NSP loss can slightly improve downstream task performance. Thus, they propose RoBERTa (short for A Robustly Optimized BERT Pretraining Approach), which improves the performance of $\mathbf{BERT_{BASE}}$ on downstream tasks by re-training the model with larger batches and more data, but without NSP loss.

CodeBERT and CuBERT Given the revolutionized success of BERT for many NLP tasks, it has been widely applied to many other domains. For example, Feng et al. [44] propose CodeBERT, which shares exactly the same model architecture as $\mathbf{RoBERTa_{BASE}}$. Kanade et al. [71] propose CuBERT to learn contextual code embeddings using the $\mathbf{BERT_{LARGE}}$ model.

Both CodeBERT and CuBERT use the BERT model architecture and treat the source code as plain text for training. The differences between CodeBERT and CuBERT mainly come from the way of constructing the training corpus. CodeBERT considers the natural language texts (i.e., the description documentation of source code) and source code as two different types of data and constructs the training corpus (i.e., bimodal data and unimodal data) based on these two types of data. However, CuBERT does not separate the natural language texts and source code and mixes natural language tokens with source code tokens during training.

Limitations. Based on our understanding, we find that there may still exist limitations for these two models. The first one is that they both treat the source code as plain text and do not consider the structural information explicitly. Another limitation is the high computation cost for training such huge models (both have millions of parameters.), making it almost impossible for us to modify and re-train the models by ourselves. For example, CodeBERT spends more than ten days to finish the training using 16 interconnected NVIDIA Tesla V100 GPUs.

2.2 Related Work

Source code embeddings is an essential part of many SE tasks [6, 11, 21, 28, 29, 57, 144, 178, 185]. Due to the advancement of neural networks, researchers propose various approaches for learning code embeddings to assist in SE tasks. In this section, we report related works for each category of code embedding presented in Section 2.1.1.

Textual context-based code embeddings. Prior work extracts the local textual information from the source code and then applies embedding techniques to the extracted textual information. For example, Harer et al. [57] propose a source-based model for automated software vulnerability detection. In their model, they first tokenize the collected open-source C/C++ programs into sequences of tokens and then apply Word2vec to convert code tokens into vector representations. The learned Word2vec representation of code tokens is finally fed into a TextCNN model [74] for classification. Chen and Monperrus [29] train Doc2vec [80] on a corpus of Java files. Source code components from each Java file are extracted and tokenized. The tokenized source code components are used to train a Doc2vec model for automated program repair. Similarly, White et al. [185] train source code embeddings using Word2vec from the normalized file-level corpora. The trained embeddings are then used for the initialization of the embedding layer of the recursive autoencoder which is a type of neural network that recursively learns the representations of the code snippet. In addition, Efstathiou and Spinellis [43] adopt fastText [19], which utilizes the subword information, to train code embeddings for different programming languages, including Java, Python, PHP, C, C++, and C#. However, they only propose potential applications of the models without evaluation. Theeten et al. [166] propose to use the skip-gram model of Word2vec [123, 124] to generate embeddings for library packages of different programming languages, which are later used for retrieving the similar libraries of a given library.

Intuitively, using local textual context is reasonable as developers always code the related statements together. However, during the embedding training, neither using a too-large local window nor a too-small window is desired. A too-large local window size may include redundant or unrelated tokens (i.e., noise tokens) in, while a too-small local window size may lose the important context tokens. In addition, considering the code snippet as plain text results in the omitting of the structural information in the source code that may be important for some downstream tasks.

Structural context-based code embeddings. To leverage the rich structural information of source code, some researchers propose AST-based representation approaches. An AST represents the source code with a tree structure, which has been proven to be useful in a wide range of software engineering fields. For example, Zhang et al. [204] propose to learn source code representations based on abstract syntax trees. They train the program embeddings for two downstream tasks, i.e., code clone detection and source code classification. Similarly, Büch and Andrzejak [21] implement an AST-based Recursive Neural Network (RNN) for code clone detection. Alon et al. [11] propose code2vec, which also relies on the AST representation of the source code. The ASTs are converted into a set of triples which are later fed into an attention-based neural network for the task of method name prediction. Bertolotti and Cazzola [16] try to learn the statement-level code representations based on the sub-trees of the ASTs. The code embeddings are then evaluated on three downstream SE tasks: code summarization, statement separation, and code search.

Except for the direct use of ASTs, recently, researchers have proposed to adopt more sophisticated structural information from source code. For example, Tufano et al. [169] try to combine different code representations for detecting code clones. In their work, they consider four different training contexts i.e., identifiers, AST, bytecode, and control flow graph (CFG) extracted from the source code fragments. Zeng et al. [201] propose deGraphCS to transfer source code into variable-based flow graphs and then apply a gated graph neural network to model them. Similarly, Liu et al. [105] convert the program to a graph with syntactic edges (AST Edge, NextToken SubToken) and data-flow edges (ComputedFrom, LastUse and LastWrite) to represent the source code. However, they only focus on the task of code search. Yu et al. [196] adopt a graph-based code semantics learning method to encode CFG (Control Flow Graph) or PDG (Program Dependency Graph) for semantic code clone detection.

Allamanis et al. [8] extract the data flows from the source code and then use a Gated Graph Neural Network to learn the program representation. However, the data flows they extracted are based on the AST representation of the source code. In other words, they explicitly expose part of the AST of a program (i.e., the subtree that contains syntax tokens corresponding to declarations and updates of variables) as the structured input to the embedding learning model. This might be useful for downstream tasks that are sensitive to the data operations of a program, for example, the task used in their work, variable misuse detection. Although by doing this, they can focus on the utilization of the data flows, there is still a chance of missing some important information about the program, at least, one cannot build a program only based on the data-flow graph.

To further explore the structural information of the source code for producing generalizable code embeddings, Sui et al. [162] utilize control flows and data flows of a program. They extend the structural information by 1) mining long-range data flows across different methods, and (2) precisely extracting the data-flow information based on the pointer alias information. The long-range and precise data flows make the generated code embeddings able to capture the method or data dependence of the program, which is useful for tasks that involve the interaction between different methods, for example, in the task of code summarization: if the target method calls another method in the method body, it would be useful to utilize the long-range data flows to analyze what happens between these two methods. However, this kind of global information may not bring many benefits for other tasks, such as the task of code authorship identification, as the programs (i.e., the input of the task) are isolated from each other and written by different authors, where the method-level approach (e.g., code2vec) may perform better.

Chapter 3

Revisiting the Use of Code Embeddings for SE Tasks

Word representation plays a key role in natural language processing (NLP). Various representation methods have been developed, among which pre-trained word embeddings (i.e., dense vectors that represent words) have shown to be highly effective in many neural network-based NLP applications. However, the use of pre-trained code embeddings for software engineering (SE) tasks has not been extensively explored. A recent study by Kang et al. [72] finds that code embeddings may not be readily leveraged for the downstream tasks that the embeddings are not trained for. However, Kang et al. [72] only evaluate two code embedding approaches on three downstream tasks, and both approaches may have not taken full advantage of the context information in the code when training code embeddings. Considering the limitations of the evaluated embedding techniques and downstream tasks in Kang et al. [72], we would like to revisit the prior study by examining whether the lack of generalizability of pre-trained code embeddings can be addressed by considering both the textual and structural information of the code and using unsupervised learning.

In this chapter, we propose a framework, StrucTexVec, which uses a two-step unsupervised training strategy to incorporate the textual and structural information of the code. Then, we extend prior work [72] by evaluating seven code embedding techniques and comparing them with models that do not utilize pre-trained embeddings in six downstream tasks. Our results first confirm the findings from prior work, i.e., pre-trained embeddings may not always have a significant effect on the performance of downstream SE tasks. Nevertheless, we also observe that: (1) different embedding techniques can result in diverse performance for some SE tasks; (2) using well-pre-trained embeddings usually improves the performance of SE tasks (e.g., all six downstream tasks in our study); (3) the structural context has a non-negligible impact on improving the quality of code embeddings (e.g., embedding approaches that leverage the structural context achieve the best performance in five out of six downstream tasks among all the evaluated non-contextual embeddings), and thus future work can consider incorporating such information into the large pre-trained models. Our findings imply the importance and effectiveness of combining both textual and structural contexts in creating code embeddings. Moreover, one should be very careful with the selection of code embedding techniques for different downstream tasks, as it may be difficult to prescribe a single best-performing solution for all SE tasks.

3.1 Introduction

In recent times, distributed representations of words, also called word embeddings, have shown to be highly effective in many neural network models-based natural language processing (NLP) tasks, such as named entity recognition (NER), part-of-speech (POS) tagging [84], and sentence classification [78]. In NLP, various word embedding techniques have been developed to encode words with different meanings into a low-dimensional vector space. Meanwhile, distributed code/program representations (i.e., code embeddings) have also proven to be useful in assisting in software engineering tasks, such as automatic program repair [29, 178, 185], software vulnerability prediction [57, 144], method name prediction [6, 11], and code clone detection [21].

Researchers have worked on a number of approaches for representing source code into vectors in SE tasks. Among these, some directly apply word embedding techniques to source code to produce representations for code tokens, for example, Theeten et al. [166] use Word2vec [123, 124] to generate embeddings for software libraries, while other researchers have proposed task-specific approaches for SE tasks. For example, Alon et al. [11] propose a path-attention network to learn source code embeddings for the task of method name prediction. These generated source code embeddings that are trained on large source code datasets can later be used for other SE tasks, and thus are also called pre-trained code embeddings.

Although different code embedding learning techniques have been proposed, the use of pre-trained code embeddings for different SE downstream tasks has not been extensively explored. A recent study by Kang et al. [72] evaluates two code embedding approaches (i.e., GloVe [142] and code2vec [11]) on three downstream SE tasks, namely code comment generation, code authorship identification, and code clone detection. They find that the pre-trained code embeddings may not be readily leveraged for the downstream tasks that the embeddings are not trained for.

Intuitively, pre-trained code embeddings can bring more knowledge about the semantic and syntactic meanings of code tokens as they are trained on large external datasets. Thus, models using pre-trained code embeddings are expected to perform better than models without that information. On the other hand, both studied embedding techniques only utilize partial information during the embedding training and do not take full advantage of the information from the source code. In particular, GloVe treats the source code as plain text and only considers the unstructured local textual information, and code2vec parses each method in the source code to an abstract syntax tree (AST) and focuses on the utilization of the structural information extracted from such ASTs. Hence, we would like to find out whether the poor performance of the pre-trained code embeddings can be addressed by combining both the textual and structural information in the source code, as well as how the different types of information affect the performance of downstream tasks. In addition, we would like to understand how the advancement of embedding techniques in recent years impacts the findings from the prior research.

Therefore, in this chapter, we revisit and extend the assessment of using pre-trained code embeddings across a wider variety of SE tasks, aiming to provide more insights to guide further research that leverages code embeddings. Note that the main goal of this chapter is not meant to propose entirely new methods. Instead, the goal and main contribution of the chapter is to revisit the findings from prior research in order to understand whether they still hold with the fast progress of related research in recent years and with the consideration of extra information (e.g., method invocation).

Our work extends prior work [72] from two aspects. First, in addition to GloVe and code2vec, we evaluated more code embedding techniques. In particular, we propose a two-stage embedding learning approach called StrucTexVec, designed specifically for source code data with the special consideration of learning from both the textual and structural information. In particular, in the first stage, to capture the structural information, we pre-train the embeddings by customizing the dependency-based word embedding approach [84]. Unlike code2vec, which only considers the AST information within every single method, StrucTexVec also utilizes method call and variable reference information. In the second stage, to incorporate the textual context information, we re-train the embeddings on the tokenized source code. The code embedding learning is framed as an unsupervised learning procedure, as we aim to generalize the learned embeddings to different downstream tasks. Thus, StrucTexVec does not require any manual labeling of the training data. In addition, we also consider Word2vec [123, 124], fastText [19] and the recently released contextual embedding techniques, such as CodeBERT [44] and CuBERT [71] for training code embeddings. In total, we evaluate seven code embedding techniques with different configurations.

Second, we also extend prior work [72] by considering more downstream tasks. To assess the effectiveness of using pre-trained code embeddings in SE tasks and understand the impact of structural and textual information on creating generalizable code embeddings, we conduct a comprehensive quantitative evaluation in six downstream SE tasks: code comment generation, code authorship identification, code clone detection, source code classification, logging statement prediction, and software defect prediction. We apply and compare the seven learned code embeddings in these benchmark tasks. The source code and benchmark tasks are publicly available. The contributions of this chapter are as follows[1]:

- We revisit and extend previous work by evaluating more embedding techniques across a wider variety of downstream SE tasks.
- Our findings confirm the challenge of using pre-trained code embeddings in downstream SE tasks [72], as using pre-trained code embeddings may not always achieve boosting in performance.
- We observe that using pre-trained embeddings performs better than not using them in all the downstream tasks. However, different embedding techniques can result in diverse performance, and there does not exist an embedding technique that outperforms others in all nor even the majority of the tasks.
- We also observe that both the structural and textual information have a non-negligible impact on the quality of pre-trained code embeddings and find that the structural information has a larger impact on the quality of the code embeddings than the textual information. Researchers may consider incorporating the structural information into CodeBERT or CuBERT for further improvement.

Our findings suggest that future research and practice should take careful consideration on the selection of code embedding techniques before training their models for different tasks, as it may be impossible to prescribe a single best-performing solution for all SE tasks.

Chapter organization. We describe our proposed approach, StrucTexVec in Section 3.2. Section 3.3 presents the experimental setup. Section 3.4 presents our experimental results and answers to research questions. Section 3.5 discusses our lessons learned. Section 3.6 presents threats to the validity of our study. Finally, Section 3.7 concludes this chapter.

[1]We share the trained embeddings together with the downstream tasks at Google Drive.

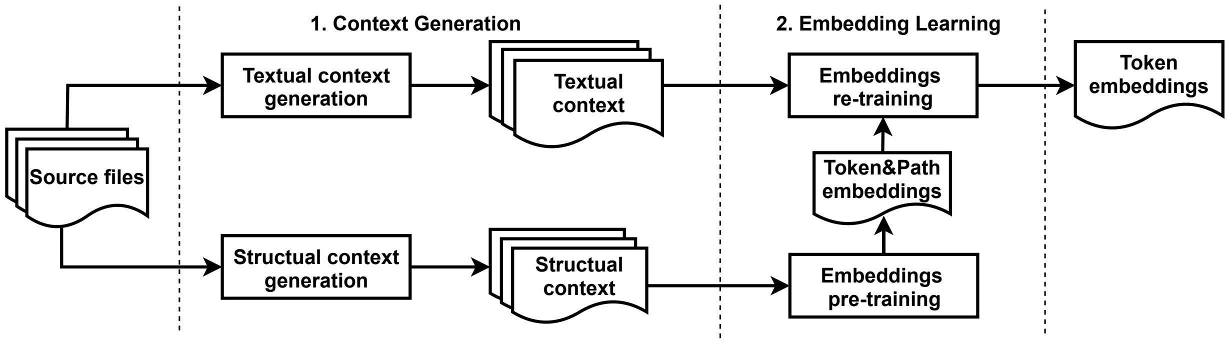


Figure 3.1: The overall framework of StrucTexVec.

3.2 StrucTexVec: Embedding with Structural and Textual Information

To understand the impact of the structural and textual information on the performance of downstream tasks, we propose StrucTexVec, which consists of a context generation phase followed by a two-stage embedding learning phase. Figure 3.1 outlines the overall framework of StrucTexVec. StrucTexVec preprocesses a collection of source code files to generate the textual and structural context. In the embedding learning phase, the customized dependency-based skip-gram technique [84] is used to train the token embeddings based on the structural context. Then, to incorporate the textual information, StrucTexVec re-trains the token embeddings (our focus is the token embeddings, and thus the path embeddings are ignored during the re-training stage) on the tokenized textual context. Below, we elaborate on each of the phases of StrucTexVec.

3.2.1 Context Generation

In this section, we describe the procedures for generating the textual and structural context from source code files.

3.2.1.1 Textual context generation

Word2vec, GloVe, and fastText all use a window with a fixed length to construct the target word's context [19, 123, 124, 142] from the training corpus. In software engineering tasks, many existing approaches consider the source code as plain text (i.e., sequences of tokens) and achieve promising results [5, 7, 61, 179]. Similarly, in this work, we also utilize the textual context and treat the source code files as plain text.

As described in Section 2.1.1, to convert the source code into the trainable textual context, the source code is first tokenized into a sequence of tokens, where all the non-identifiers (e.g., quotation marks) are removed. Meanwhile, following previous studies [11, 28, 72], the tokenized source code is lowercased.

3.2.1.2 Structural context generation

Apart from using the textual context for source code embedding training, some researchers [10, 18, 148] also utilize the structural context for software engineering tasks. Therefore, in our work, we also adopt the structural context. In particular, to enrich the context, our structural context contains three components: (1) AST paths, (2) method calls, and (3) variable references.

In StrucTexVec, we use srcML[2] [31] to represent source code as abstract syntax trees. As described in Section 2.1.1, the leaf nodes are tokens in the source code that are connected by a set of srcML tags that provide the syntax structure of the code.

Based on the XML tree representation provided by srcML, we then extract the structural context into a sequence of path triples. Our work shares an analogous way with that of Alon et al. [11] to extract within-method triples. However, rather than only considering the information within a single method as in Alon et al. [11], we enrich the structural context by mining the following three types of context: (1) AST paths, (2) method calls, and (3) variable references as described below.

AST path context. Given the ASTs of the source code, we perform a structural traversal to extract a collection of path triples. In each triple, $\langle w_1, p, w_2 \rangle$, w_1 and w_2 are two different leaf nodes in one method's AST, p is the shortest path between these two nodes. The leaf nodes are source code tokens and the shortest path describes the syntactic relationship between any two of them. Algorithm 1 presents the details to construct such path triples. For example, as shown in Figure 2.3, given two source code tokens, e.g., "`public`", and "`accept`", the AST node sequences in the shortest path is $\langle specifier, type, function, name \rangle$. Considering the traversal directions, the final representation of the path becomes $specifier^{\uparrow}$-$type^{\uparrow}$-$function$-$name^{\downarrow}$, where the $\uparrow$ and $\downarrow$ are traversing directions and no direction means that it is an inflection node of a path. Thus, we get the path triple, $\langle$ "`public`", $specifier^{\uparrow}$-$type^{\uparrow}$-$function$-$name^{\downarrow}$, "`accept`"$\rangle$. Similarly,

[2]https://www.srcml.org/.

Algorithm 1: TripleGeneration

```
   Input: AST of a method, $M$.
   Output: A collection of path triples, $T$.
1  begin
2      $T \longleftarrow \varnothing$
       // Extract all the leaf nodes of $M$
3      $leafNodes \longleftarrow$ FindLeafNodes($M$)
4      foreach $w_1 \in leafNodes$ do
           // Find the path from root to the leaf node
5          $path2w_1 \longleftarrow$ GetPathFromRootTo($w_1$)
6          foreach $w_2 \in leafNodes$ do
7              if $w_1 \neq w_2$ then
8                  $path2w_2 \longleftarrow$ GetPathFromRootTo($w_2$)
                   /* Return the longest common path prefix of $path2w_1$ and $path2w_2$.
                      */
9                  $commPrefix \longleftarrow$ GetCommonPrefix($path2w_1$, $path2w_2$)
10                 remove the subpath $commPrefix$ from $path2w_1$ and $path2w_2$
                   /* Concatenate $path2w_1$ and $path2w_2$ with the last element in
                      $commPrefix$.  */
11                 $p \longleftarrow path2w_1 + commPrefix[-1] + path2w_2$
12                 $T \longleftarrow T + < w_1, p, w_2 >$
13     return $T$
```

we can change the target node and the source node to collect more path triples for embedding learning.

Method call context. We aim to extract the method calls within one project. We first identify all project-defined methods and the methods that are called by the identified project-defined methods. Then, we start to collect the call chain information between these methods. To accelerate the process of constructing call graphs, we use a heuristic that involves only faster shallow exact method name matching. More specifically, srcML provides a *function* tag that helps us to identify all the project-defined methods and a *call* tag to label the methods that are called by another method. For example, the previous code snippet (see Figure 2.1) defines a method "`accept`", and assume it is called in another method that is defined in this project, "`connect`", and therefore, we have the triple, $\langle$"`connect`", *call*, "`accept`"$\rangle$, where the "`accept`" is the project-defined method name and called by the method "`connect`".

Variable reference context. We also extract the variable references. We first identify all class variables and instance variables in one Java file using srcML, and then we collect methods that contain references to the previously identified variables by using the heuristic of exact variable name matching. For example, in the previous code snippet (see Figure 2.1), the variable "`fileFilters`" is declared and initialized as an instance variable and referenced in method "`accept`", and therefore, we have the triple, ⟨"`accept`", $reference$, "`fileFilters`"⟩, where "`accept`" is the method which contains references to the instance variable "`fileFilters`".

The output of our context generation phase (i.e., the structural and textual context) are used as the input for the embedding learning phase.

3.2.2 Embedding Learning

In order to combine both the textual and structural knowledge into code embeddings, StrucTexVec adopts a two-step training strategy: (1) pre-training token and path embeddings using the customized dependency-based model [84] and (2) re-training the token embeddings using Word2vec [123, 124].

3.2.2.1 Path-based model for embedding pre-training

As explained in Section 2.1, the original Word2vec models use a local fixed-size window to construct a word's context, and then the context words are used for embedding training [123, 124]. Different from the original models, Li et al. [84] improve Word2vec by integrating the syntactic dependency information between words into the embeddings. Since it is a recently released model and achieves competitive results on different tasks in natural language processing [84], in this work, following previous work [19] which extends the skip-gram model of Word2vec, we also customize the improved skip-gram as a path-based skip-gram for training code embeddings, aiming to incorporate the structural context of source code.

Path-based skip-gram. Figure 3.2 is an overview of the customized path-based skip-gram (PSG). In the original work of Li et al. [84], a word is modeled by its context of the syntactic dependency information. In our model, words are replaced with the tokens in the source code, and the dependency information is changed to the paths between these tokens in the ASTs of the source code. By doing this, we can apply the model to our extracted structural context. Following Li et al. [84], we use negative sampling to improve the computation efficiency. As Figure 3.2 shows, in the modified model, the token (i.e., $\mathbf{V}_{\text{``}\mathtt{public}\text{''}}$) is the target token, the path (i.e., $\mathbf{V}_{specifier\uparrow\text{-}type\uparrow\text{-}function\text{-}name\downarrow}$) is the token's connecting path, the negative token (i.e., $\mathbf{V}_{\neg\text{``}\mathtt{public}\text{''}}$) is selected from the vocabulary and

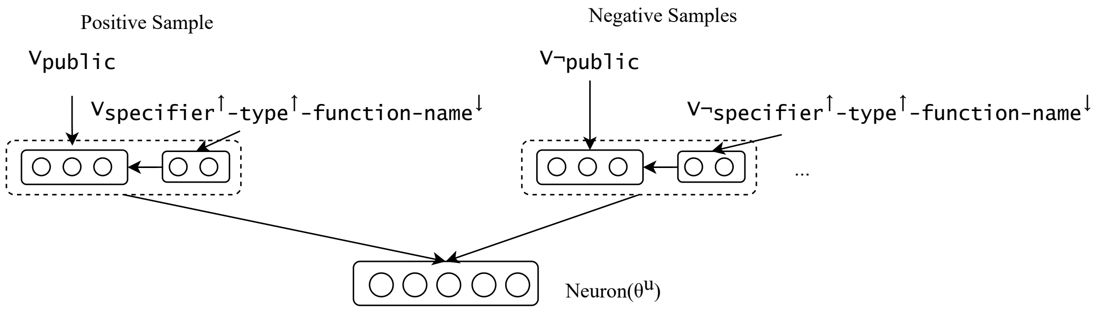

Positive Sample
Negative Samples
V public
V specifier↑-type↑-function-name↓
V ¬public
V ¬specifier↑-type↑-function-name↓
...
Neuron(θu)

Figure 3.2: The overview of the path-based skip-gram model with negative sampling (take “`public`” as an example).

the negative path (i.e., $\mathbf{V}_{\neg specifier^{\uparrow}\text{-}type^{\uparrow}\text{-}function\text{-}name^{\downarrow}}$) is selected from the path sets. All the negative samples are randomly selected based on the frequency of occurrence in the training corpus. We build two vocabularies, the tokens in the training corpus and those in the extracted paths. The vector representations of both vocabularies are updated during the training process.

After that, we concatenate the two negative parts together and form a negative sample for later embedding learning. We take the following as the objective function of the model:

$$\mathcal{L} = \sum_{w_t \in \mathcal{D}} \prod_{\widetilde{w_t} \in \mathcal{C}_{\mathcal{D}}(w_t)} \prod_{u \in \{w\} \cup NEG^{\widetilde{w_t}}(w_t))} \mathcal{L}(w_t, \widetilde{w}_t, u)), \tag{3.1}$$

where $NEG^{\widetilde{w_t}}(w_t))$ is defined as the negative sampling set for target token w_t, $\mathcal{C}_{\mathcal{D}}(w_t)$ is the context set for w_t, and $\mathcal{L}(w_t, \widetilde{w}_t, u) = L^{w_t}(u) \cdot \log[\sigma(x_{\widetilde{w_t}}^{\intercal} \theta^u)] + [1 - L^{w_t}(u)] \cdot \log[1 - \sigma(x_{\widetilde{w_t}}^{\intercal} \theta^u)]$, where $\sigma(\cdot)$ is the *sigmoid* activation function, θ^u is the parameter vector of NS neuron, and $L^{w_t}(u)$ is an indicator function of which value depends that u is a positive example or negative example.

We would like to note that the path-based model is different from code2vec in the following aspects (1) our path-based model is an unsupervised approach, which does not require any manual labeling of the training data, aiming to produce more generalizable embeddings; 2) we attempt to include more types of information, such as the method call context and the variable reference context, which are not included in code2vec and Word2vec.

The path-based model produces a vector for each token and each path that captures the structural context of the source code. (i.e., the Token&Path embeddings in Figure 3.1).

3.2.2.2 Word2vec for embedding re-training

To further incorporate the textual context of code tokens, we adopt the original skip-gram to re-train the code embeddings produced by the path-based skip-gram. Here, we use the token vectors produced by the pre-training stage to initialize the embeddings instead of using the original random initialization. As introduced in Section 2.1.2.1, skip-gram takes a sequence of tokens as input, in this work, words are replaced with tokens in the textual context.

The output of our two-stage embedding learning process (i.e., the token embeddings) are used as the input for our downstream tasks.

3.3 Experimental Setup

In this section, we present details of our dataset used for training the code embeddings. We also introduce six common downstream tasks for quantitative evaluation of the pre-trained embeddings, three of which, i.e., (1) code comment generation, (2) code authorship identification and (3) code clone detection, are used for evaluating pre-trained embeddings in prior research [72]; while (4) source code classification, (5) logging statement prediction and (6) software defect prediction, are newly added in this chapter.

3.3.1 Dataset for Learning Pre-trained Embeddings

In this work, we use the *Java-Small* dataset[3] to build the pre-trained embeddings for all the non-contextual embedding techniques. This dataset is collected from Java projects hosted on GitHub. Note that for CodeBERT and CuBERT, due to the limitations of the computation resources, we use their released models, instead of training CodeBERT and CuBERT from scratch. We save their embedding layers into Word2vec/GloVe format to integrate into our evaluation pipelines.

After fetching the files of the training projects, we first perform filtering to remove the irrelevant files and only keep the source code files with the *.java* extension. As illustrated in Figure 3.1, we composite two types of contexts from the filtered source code files: (1) structural context, which refers to extracted path triples in the source code and (2) textual context, which is the plain text of Java files.

[3]https://s3.amazonaws.com/code2vec/data/java-small_data.tar.gz

3.3.2 Settings for Embedding Learning

At the pre-training stage, the token vectors are randomly initialized and trained using the path-based model. At the re-training stage, the token vectors are initialized by the embeddings produced by the path-based model and further trained using the skip-gram implemented in Gensim[4] [149].

Moreover, to investigate the impact of pre-trained embeddings on software engineering tasks, we also evaluate the other six existing embedding techniques as described in Section 2.1 and compare their performance with the models without pre-trained embeddings. To ensure a fair comparison across all the embedding techniques, we either follow the parameter settings in previous work (e.g., 128 dimensions) or use the default parameter values when the parameters are not specified in previous work (e.g., training epochs). To avoid bias, we do not try to fine-tune these settings only for our method. The detailed parameter settings are shown in Table 3.1.

Table 3.1: Parameter settings for different embedding techniques.

	Non-contextual embeddings					Contextual embeddings	
	Word2vec	GloVe	fastText	code2vec	StrucTexVec	CodeBERT	CuBERT
Vocabulary	109,743	192,363	109,743	507,271	192,362	50,265	50,297
Epoch	5	5	5	20	10 & 5	-	2
Window	5	5	5	5	5	-	-
Negtive	5	-	5	-	4 & 5	-	-
Dimension	128	128	128	128	128	768	1024

Note (1) Dimension: following the settings in prior work [11, 84, 204], we set the dimension of the trained token vectors to 128. For CodeBERT and CuBERT, we take the results from their released models. 2) Epoch: StrucTexVec contains a two-step training, in our experiments, 10 epochs for pre-training and 5 epochs for re-training, both are the default values of the released source code.

The embeddings of StrucTexVec, GloVe, fastText, and Word2vec are all trained in CPUs and code2vec is trained in an NVIDIA GTX 1080 Ti GPU, and it takes less than 30 minutes to finish the training process of each of the embedding techniques, which is acceptable. For CodeBERT and CuBERT, we do not train them from scratch and choose to use the already released models, as it requires not only more GPUs but also a long period (several days) to finish the training. For example, CodeBERT spends more than ten days to finish the training using 16 interconnected NVIDIA Tesla V100 GPUs.

[4]https://radimrehurek.com/gensim/

3.3.3 Evaluation Tasks

In this section, we briefly describe the six downstream tasks that are used to evaluate our pre-trained embeddings, as well as the corresponding datasets and evaluation metrics. Five of the six tasks use neural network-based methods and one task (i.e., software defect prediction) uses a traditional machine learning method (i.e., logistic regression). Our focus is the impact of the embeddings on different downstream tasks, i.e., whether the pre-trained code embeddings can improve the model performance or not.

Besides, for comparison between different embedding techniques, all the embeddings are used in the same manner for downstream tasks, that is for each task, only the embeddings are changed, and other parameters are kept the same. For example, for deep learning-based tasks, the code embeddings are used to initialize the embedding layer, and OOV tokens are randomly initialized, which can be later updated based on the training data from different tasks.

Code comment generation is a task to automatically generate code comments for a code snippet [63, 122, 127, 160], which is helpful in program understanding and maintenance. Code comment generation is considered a downstream task in previous work [72] to evaluate the effectiveness of code embeddings.

In our work, we follow the work of Kang et al. [72] and evaluate the embeddings based on the approach proposed by Hu et al. [63]. Hu et al. [63] treat the code comment generation task as a neural machine translation task, where the input is the source code snippet and the output is the code comment. Thus, they adopt an encoder-decoder model. In particular, they use two Long Short-Term Memory (LSTM) layers for both the encoder and decoder and 500 hidden units for each layer. During model training, both the learning rate and dropout rate are set to 0.5. The model is trained for 50 epochs. The training dataset is provided by Hu et al. [63], which was initially collected from GitHub. The dataset contains over 330,000 <method, comment> pairs for training, 5000 pairs for validation, and 5000 for testing.

We follow the work of [63, 72] and use the machine translation evaluation metric BLEU [141] to measure the quality of the generated comments[5].

Code authorship identification is a task of identifying the author of a given code [2, 64]. This task has attracted increasing attention in the field of privacy and security, where it can be used to identify the authors of malware and other malicious programs. We follow the work of Kang et al. [72] and select this task as a downstream task.

[5]The detailed explanation of BLEU score can be found in Section 6.4.

In our work, we evaluate the embeddings based on the approach proposed by Kang et al. [72]. Kang et al. [72] treat the code authorship identification task as a multi-class classification problem, where the input is the code snippet and output is the author. They build a neural network-based model, which contains two LSTM layers and a fully connected layer. During model training, we set the batch size to 64 to guarantee a relatively smaller training loss. The model is trained for 50 epochs. We use the same dataset as that of Kang et al. [72], which was collected from Google Code Jam. The dataset contains 2250 programs written by 250 authors, among which there are 2000 programs for training and 250 programs for testing, and within each part the classes are balanced.

We follow the previous work [72] and use the test accuracy to measure the performance of the models with different code embeddings.

Code clone detection is a task of identifying all pairs (or groups) that are clones (of a given type). It is a useful task for program maintenance [12, 42, 70, 121, 153, 167, 181, 184]. For example, if a bug is identified in one code fragment, all the other duplicate code fragments also need to be checked for the same bug. This task is also identified as a downstream task to evaluate code embeddings in prior work [72].

In general, there are four different types of code clones based on the type of similarity two code fragments have [79]:

- Type-1 clone refers to identical code fragments except for changes in comments, whitespace, and layout.
- Type-2 clone refers to identical code fragments except for differences in identifier names or literal values, comments, types, and layouts.
- Type-3 clone refers to code fragments that are syntaxally similar but have statements added, modified, or removed.
- Type-4 clone refers to code fragments that are syntaxally different but with the same functionality.

In our work, we use the approach proposed by Zhang et al. [204], as it is recently proposed and gives competitive results. Zhang et al. [204] consider the code clone detection task as a binary classification task, where the input is two code snippets and the output is 1 if they are duplicates and 0 otherwise. They use a bidirectional Recurrent Neural Network based model, and the model is trained for 15 epochs with a batch size of 128. This task contains two datasets, which are constructed from standard BigCloneBench (BCB) [163] and Online Judge system (namely, OJClone). For BCB dataset, The similarity of two fragments of Type-1 and Type-2 is 1. Type-3 contains two subcategories: Strongly Type-3 and Moderately Type-3, of which the similarities are in the range [0.7, 1) and [0.5, 0.7), respectively. The similarity of Type-4 is in the range [0, 0.5) [204].

We follow the work of [204] and use F1-measure (F1) as the evaluation metric in this task.

Source code classification is a typical classification task that classifies code fragments into corresponding categories. This task is identified as a downstream task as it is widely studied in the literature [50, 73, 131, 171, 204] and has various applications [50, 73, 131, 171, 204].

In our work, the source code classification task is considered as a multi-class classification problem. We apply a convolutional neural network proposed by Kim [74] to the source code. We choose to use this model as it is widely used for classification tasks and achieves competitive results, and we want to cover more different types of neural network models. The input to the model is the source code and the output is its class label (e.g., functionality). During training, we use the default parameters, that is we set the learning rate to 0.01, batch size to 64, and dropout rate to 0.5. We train the model with 50 epochs. The dataset is collected from the Online Judge system[6] and provided by Mou et al. [131].

Similar to the task of code authorship identification, we follow the work of [204] and use the test accuracy as the evaluation metric.

Logging statement prediction is a task of predicting whether there is a need to insert logging statements for a given code snippet. Logging statements play important roles in the daily tasks of developers [37, 88], and this task can provide logging suggestions that are helpful for software developers [88].

Li et al. [88] consider the logging statement prediction task as a binary classification problem, where the input is the code snippet without the logging statement and output is the decision whether to insert a logging statement or not. In this task, we also use the approach proposed by Kim [74] as in the task of source code classification. The evaluation dataset is provided by Li et al. [88], which contains five subject systems.

In this work, same as prior work [88], the balanced accuracy (BA) metric is used to evaluate the performance of the model with different embeddings.

Software defect prediction is a task of predicting whether the code snippet contains defects or not. Defect prediction can help avoid future bugs in software releases and improve the quality of software [179]. This task is selected as a downstream task because it is widely studied in the literature [179].

[6]`https://sites.google.com/site/treebasedcnn/`

For this task, following prior work by [179], we leverage the Logistic Regression (LR) classifier where the input features are the average of the embeddings of the code tokens in a file and the output is the 1 if there is a defeat detected or 0 otherwise. The dataset is provided by Wang et al. [179], and they use F1 score as the evaluation metric. Thus, we follow their work and use the same metric for this task.

3.4 Experimental Results

In this section, we show the quantitative results on the previously identified downstream tasks, and based on the results, we aim to answer the following research questions:

RQ1: How effective are pre-trained embeddings in improving the performance of downstream SE tasks?

To evaluate the effectiveness of using pre-trained code embeddings, we compare models using pre-trained embeddings, including the non-contextual embeddings and contextual embeddings, to models that do not use pre-trained code embeddings (i.e., None).

Models using pre-trained embeddings can perform better than models without pre-trained embeddings. Table 3.2 shows the evaluation results of utilizing different code embeddings on six downstream tasks. The best results are highlighted in bold. As shown in the table, models using pre-trained embeddings achieve the best results in all six tasks. For example, by using embeddings produced by code2vec, we obtain a 2.6% absolute increase in accuracy on the source code classification task compared to the model without pre-trained embeddings. In addition, we observe that there is a slight improvement in the tasks of code comment generation and logging statement prediction when using the embeddings generated by StrucTexVec than other prior non-contextual pre-trained embeddings that only consider structural or textual information, which implies the effectiveness of combining both the textual and the structural context in creating generalizable code embeddings. We further analyze the datasets of these two tasks and find that when there is a relatively larger training dataset, StrucTexVec performs better. For example, for the task of code comment generation, there are more than 330,000 training samples, and there are more than 20,000 training samples for the task of logging statement prediction. Moreover, we find that in all the evaluated tasks, code embeddings trained in an unsupervised manner do not always outperform embeddings trained on a specific task. The results demonstrate that existing neural networks can benefit from the pre-trained embeddings,

Table 3.2: Evaluation results on the test set of six downstream tasks. The second last row shows the percentage of the best result produced by each approach on 22 datasets and the last row is the weighted averaged percentage of best results on six downstream tasks (i.e., each task's contribution to the percentage is weighted by its number of datasets).

				Non-contextual embeddings					Contextual embeddings	
Downstream Tasks	Evaluation Metrics	Dataset	None	Word2vec	GloVe	fastText	code2vec	StrucTexVec	CodeBERT	CuBERT
Code comment generation	BLEU	GitHub	14.9	15.4	15.9	14.6	15.3	16.0	**16.7**	16.1
Code authorship identification	Accuracy	Google Code Jam	87.5	80.2	87.5	77.1	85.4	86.5	87.0	**89.1**
Code clone detection	F1	BCB	92.7	**93.8**	**93.8**	**93.8**	93.5	93.6	93.5	93.6
		OJClone	85.1	86.8	81.4	78.0	**89.7**	88.1	85.9	81.6
Source code classification	Accuracy	OJ dataset	88.5	87.0	89.2	77.7	**91.2**	89.1	79.8	75.8
Logging statement prediction	Balance Accuracy	Airavata	**95.6**	94.3	94.2	93.1	94.8	94.5	**93.8**	93.4
		Camel	76.6	77.8	77.5	76.4	77.4	**79.2**	77.1	75.0
		CloudStack	85.9	86.0	85.5	84.7	86.9	**87.3**	86.0	86.7
		Directory-Server	82.9	84.1	85.6	84.7	84.0	86.6	**88.0**	81.9
		Hadoop	76.7	73.6	71.5	71.7	72.3	71.0	75.4	**77.6**
Software defect prediction	F1	Ant 1.5 ->1.6	28.0	35.5	36.0	32.9	47.6	34.2	36.4	**54.8**
		Ant 1.6 ->1.7	33.1	44.9	45.1	39.6	48.4	43.4	51.9	**52.9**
		Camel 1.2 ->1.4	23.3	43.3	45.5	43.8	43.2	**46.8**	45.6	44.2
		Camel 1.4 ->1.6	26.3	47.0	49.8	46.0	50.0	50.2	**51.2**	50.3
		jEdit 3.2 ->4.0	32.7	52.0	56.2	55.9	56.6	59.5	**61.5**	59.4
		jEdit 4.0 ->4.1	40.6	60.5	60.1	59.7	57.9	**64.7**	62.4	59.9
		Log4j 1.0 ->1.1	45.5	65.7	67.6	62.7	66.7	62.7	70.4	**72.0**
		Lucene 2.0 ->2.2	58.1	62.6	60.6	**65.1**	63.3	63.9	63.8	62.6
		Lucene 2.2 ->2.4	60.8	**66.3**	64.7	65.5	60.6	65.2	64.4	63.8
		POI 1.5 ->2.5	68.1	64.8	80.1	67.4	81.7	77.8	83.4	**85.1**

	POI 2.5 ->3.0	67.5	72.4	72.9	72.6	**74.8**	71.4	72.3	72.4
	Xalan 2.4 ->2.5	49.9	41.7	42.9	44.3	**53.6**	47.5	41.5	50.4
Percentage of best result (%)		4.5	9.1	4.5	9.1	18.2	18.2	18.2	**27.3**
Weighted averaged percentage of best result (%)		3.3	9.7	9.7	9.7	**27.8**	9.4	20.0	20.0

Note:
(1) The weighted averaged percentage of best results on six downstream tasks for each technique is calculated as: $\frac{\sum_{t \in tasks} \frac{\#\ of\ best\ performing\ datasets}{\#\ of\ total\ datasets\ in\ t}}{6} * 100\%$;
(2) For CodeBERT and CuBERT, we save their embedding layers into Word2vec/GloVe format to integrate to our evaluation pipeline. Although they all have achieved comparable performance in the downstream tasks, CodeBERT and CuBERT are contextual embeddings, which means the way we use them may not unleash their full power.

which challenges the findings of Kang et al. [72]. The findings highlight the potential of applying well-trained code embeddings to downstream SE tasks.

On the other hand, we find that including more types of training information cannot always guarantee better results, and other factors (e.g., training model, different preprocessing strategies) can also have an impact on the quality of the generated embeddings. For example, both the CodeBERT and CuBERT are trained only based on the textual information, but both of them achieve results comparable with the approaches considering both the textual and structural information (e.g., StrucTexVec). Besides, although StrucTexVec attempts to include more structural information (i.e., AST paths, method call, and variable reference), it cannot always outperform code2vec in the six downstream tasks. To understand why the performance may differ even though we use more types of training information, we do a further analysis of these two techniques. As a result, we summarize that the performance difference can come from the following aspects: (1) The difference between the training objectives. For the training objectives, our method is a task-agnostic embedding approach, which does not need to be trained together with the downstream tasks; code2vec is task-specific and trained together with the downstream task (i.e., method name prediction.). Thus, for the tasks that share similar intrinsic properties with the downstream task that is used to train the embeddings, code2vec would perform better. For example, the task of source code classification, which is to classify code fragments into corresponding categories, is similar to the method name prediction, where a category (i.e., method name) is assigned to a code fragment based on its functionality. (2) The difference between the training models. As we described in Section 3.2.2, during the embedding learning, we simply concatenate the embeddings of the token and the AST path into one single vector and then use this vector for future training. However, in code2vec, the paths and tokens are treated differently at the beginning and then fused together by using an attention layer, which is better at transmitting the information between the paths and tokens than our method. As a result, even if we use more information in our method, due to the limited ability to embed such information into the code embeddings, our method performs worse on some datasets. (3) The difference between the vocabulary sizes. We find that there are 507,271 tokens for code2vec and 192,362 tokens in our generated embeddings, which means more tokens are filtered out for our method during the preprocessing step, and this poses a non-negligible effect on the quality of the generated embeddings.

Besides, for the performance of non-contextual embeddings on the downstream tasks, we find that the vocabulary size of the embeddings has a more significant impact on the traditional machine learning models than on deep learning-based models. For example, Word2vec and fastText have a relatively smaller vocabulary size and perform worse on the task of software defect prediction that uses a traditional machine learning model. On the contrary, code2vec has the largest vocabulary size and performs better on the same task. This can be explained by the fact that a smaller vocabulary size causes more OOV tokens, and thus weakens the representation ability of the code embeddings, especially considering the fact that the code embeddings are directly used as features for the traditional machine learning models. However, for deep learning-based models, the embeddings are only used to initialize the weights of the first embedding layer, and the weights are later adjusted to better fit the training data. As a result, the impact of OOV tokens may be diminished or even erased during the model training and weight updating.

For a specific downstream task, different embedding methods can result in diverse performance, and there does not exist an embedding technique that consistently outperforms others across all or even the majority of the tasks. Table 3.2 compares the results for applying different embeddings to six downstream tasks. The tasks of code comment generation and code authorship identification illustrate a diverse performance that may be caused by different embeddings. For example, on the code comment generation task, using embeddings trained on fastText can only have a 14.6 of BLEU, compared to 15.9 when using embeddings trained on GloVe. In addition, different evaluation tasks result in different orderings of embedding techniques, raising the question that there may not exist a single optimal vector representation for all SE tasks. For instance, code embeddings suitable for source code classification (e.g., code2vec) even perform no better than random embeddings on code authorship identification. This may come from the fact that different SE tasks might highly differ in their nature and thus require different external information to boost performance. The findings suggest that one should be very careful with the selection of code embedding techniques before starting the model training in terms of different tasks. In particular, the non-contextual embeddings that leverage structural information of the code, including StrucTexVec and code2vec, perform better than other non-contextual embedding techniques and have the best results in five out of six downstream tasks.

Using pre-trained embeddings may not always improve the performance of downstream tasks significantly. We find that by using pre-trained embeddings, although we can obtain an increase in performance on different tasks, the improvement may be limited. For example, in the task of logging statement prediction, we only obtain a maximum increase of 0.5% by utilizing the pre-trained embeddings (the model without pre-trained embeddings compared to that using pre-trained embeddings produced by CodeBERT). In addition, using pre-trained embeddings causes a decrease in accuracy for the task of code authorship identification. This observation is similar to that of prior studies [72, 84, 170]. One possible reason is that the neural network-based models themselves are powerful enough and already have good results, thus it is difficult to have a large improvement (e.g., the great performance in code clone detection task shown in Table 3.2).

The result indicates the limited effect of pre-trained embeddings, as they only work on initializing the embedding layer for neural network-based models. This confirms the findings of Kang et al. [72]. Namely, code embeddings may not be a key role in boosting the performance of deep learning models. Software engineering researchers and practitioners should not only rely on using embeddings to improve their automated techniques.

Summary

In general, using pre-trained code embeddings can improve the performance of downstream SE tasks. However, different embedding techniques can result in diverse performance. Practitioners and researchers should be careful when selecting the embedding techniques for their specific tasks, as there is no single best solution.

RQ2: How do the structural and the local textual information affect the performance of the pre-trained embeddings?

To verify the effectiveness of incorporating different information extracted from source code (i.e., structural and local textual information), we design ablation experiments on these six downstream tasks.

First, to analyze the effect of the structural information, we treat the source code as plain text and only learn from the local textual information. More specifically, we remove the pre-training stage from StrucTexVec, which results in the original Word2vec model. Then, to analyze the performance gain achieved due to the utilization of the local textual context, we train the embeddings only based on the structural information extracted from the source code. In other words, we remove the re-training stage from StrucTexVec, resulting in the path-based skip-gram with negative sampling (i.e., StrucTexVec $^{-text}$ in Table 3.3). The results of our ablation experiments are shown in Table 3.3.

The structural information extracted from the source code can improve the performance of the code embeddings. By comparing the results of Word2vec with that of StrucTexVec, in total, we find that StrucTexVec can outperform Word2vec in all downstream tasks. The comparison results demonstrate that incorporating the structural context can help improve the performance of the embeddings. For example, on the code authorship identification task, by pre-training the embeddings using the structural context, StrucTexVec has an accuracy of 86.5% compared to 80.2% without the pre-training phase. This is consistent with the observation of Zhang et al. [204]. We consider that the poor performance of GloVe in the work by Kang et al. [72] may be related to the exclusion of structural information in the embeddings.

Table 3.3: Evaluation results of embeddings trained with and without the structural and local textual contexts. Word2vec is equivalent to the variant of StrucTexVec which removes the structural information from the training process; StrucTexVec $^{-text}$ only utilizes the structural information for embedding learning.

Downstream Tasks	Code comment generation	Code authorship identification	Code clone detection		Source code classification	Logging statement prediction				
Datasets	GitHub	Google Code Jam	BCB	OJClone	OJ dataset	Airavata	Camel	CloudStack	Directory-Server	Hadoop
StrucTexVec	**16.0**	**86.5**	93.6	**88.1**	89.1	94.5	**79.2**	**87.3**	86.6	71.0
Word2vec	15.4	80.2	**93.8**	86.8	87.0	94.3	77.8	86.0	84.1	**73.6**
StrucTexVec $^{-text}$	15.7	79.7	93.6	86.9	**89.7**	**95.3**	76.0	85.2	**90.0**	71.2

Downstream Tasks	Software defect prediction											
Datasets	Ant 1.5 ->1.6	Ant 1.6 ->1.7	Camel 1.2 ->1.4	Camel 1.4 ->1.6	jEdit 3.2 ->4.0	jEdit 4.0 ->4.1	Log4j 1.0 ->1.1	Lucene 2.0 ->2.2	Lucene 2.2 ->2.4	POI 1.5 ->2.5	POI 2.5 ->3.0	Xalan 2.4 ->2.5
StrucTexVec	34.2	43.4	**46.8**	**50.2**	**59.5**	**64.7**	62.7	63.9	65.2	77.8	71.4	**47.5**
Word2vec	35.5	44.9	43.3	47.0	52.0	60.5	**65.7**	62.6	**66.3**	64.8	**72.4**	41.7
StrucTexVec $^{-text}$	**38.1**	**50.2**	46.4	45.2	57.4	58.9	62.9	**66.9**	63.1	**81.2**	71.8	43.6

Code embeddings can benefit from the local textual context. According to the results of StrucTexVec $^{-text}$ and StrucTexVec from Table 3.3, we find that StrucTexVec can achieve better performance than $StrucTexVec^{-text}$ in five out of six downstream tasks. The comparison results indicate that by re-training the embeddings on the textual context, we can achieve an overall better performance. For example, on the code authorship identification task, the embeddings trained only with the structural information have a lower accuracy compared to those using both structural and textual information (i.e., 79.7% vs. 86.5%). This is consistent with the intuition that developers put similar code statements together and thus using a local context window is able to capture some semantic information from the source code.

However, the benefit from the local textual context is limited for some downstream tasks. We also find that the improvement is not always significant. For example, for the tasks of code comment generation, logging statement prediction, and software defect prediction, there is only a 0.2% to 0.3% absolute increase. Moreover, for some tasks and datasets, re-training the code embeddings using local textual context even causes a degradation of the performance. For example, the accuracy of the source code classification task decreases from 89.7% to 89.1% when utilizing the local textual information. One possible reason is that the structural information extracted from the source code is rich enough for the downstream tasks and incorporating the local textual context cannot provide much more benefit.

The structural information has a stronger impact on the quality of the code embeddings than that of local textual information. By comparing StrucTexVec $^{-text}$ with Word2vec, we can see that StrucTexVec $^{-text}$ outperforms Word2vec in four out six downstream tasks. For example, on software defect prediction task, removing the local textual information from StrucTexVec (i.e., StrucTexVec $^{-text}$) causes a degradation of 0.2%, while Word2vec only has an accuracy of 54.7%, which is a 2.6% absolute decrease. Our results suggest the promising research direction of utilizing the structural information for SE tasks.

Summary

On one hand, we find that including structural or local textual information in embeddings can improve the performance of downstream SE tasks. On the other hand, the impact of structural information is stronger for some downstream tasks. This finding highlights the importance and effectiveness of incorporating AST-based structural information for SE tasks.

3.5 Discussion

In this section, we discuss our lessons learned and compare them with the findings of Kang et al. [72]. The summary of the comparison is presented in Table 3.4.

Models using pre-trained embeddings can perform better than models without pre-trained embeddings. The choice of embedding techniques has a non-negligible impact on the model performance. We evaluate the effect of utilizing pre-trained code embeddings on six downstream tasks. We observe that utilizing the pre-trained code embeddings can improve the performance of existing models that do not use the pre-trained embeddings in all downstream tasks. For example, in the task of code comment generation, using pre-trained code embeddings produced by StrucTexVec results in a BLEU score of 16.0, which is 7.4% relatively higher than the model without the pre-trained embeddings. Our findings are different from that of Kang et al. [72]. The different results may be caused by (1) the selection of training corpus: we use the *Java-Small* dataset while Kang et al. [72] chose the *Java-Large* dataset which may contain noise data that can affect the quality of the generated code embeddings; (2) we evaluated more code embeddings produced by different techniques, among which some embedding techniques cannot benefit the models without pre-trained embeddings. For example, using code embeddings produced by fastText negatively impacts the performance of existing models.

Simpler baselines may not always outperform complex techniques. The model parameters have a non-negligible impact on the performance of the deep learning-based models. In our evaluation, we see that on the task of code authorship identification, all of our reported results outperform the simpler approach that uses simple

TF-IDF features reported by Kang et al. [72] (i.e., an accuracy of 77%). The only difference is that we set the batch size to 64 instead of the default 128. This finding shows that by tuning the parameters of the complex models, we can outperform the simpler baselines. Also, to verify our findings, we implement another two simpler baselines that use traditional features (i.e., PROMISE and TF-IDF) for the software defect prediction task, and we also observe that the use of pre-trained code embeddings can outperform the simpler baselines. Our results concur with the findings of Kang et al. [72] that other considerations (e.g., pre-processing) may have an impact on the performance of deep learning models.

Using pre-trained embeddings may not always improve the performance of downstream tasks significantly. Our findings are similar to that of Kang et al. [72], although we can obtain an increase in the performance of different tasks, the improvement may be limited. For example, in the task of logging statement prediction, the best-performing code embeddings only have a 0.5% absolute improvement compared to the model without the pre-trained code embeddings. One possible reason is that the neural network-based models themselves are powerful enough and already have good results and the code embeddings are only used for initializing the embedding layer of these models, thus it is difficult to have a large improvement for some downstream tasks. The results suggest researchers should be careful when deciding whether to use pre-trained embeddings for their specific downstream tasks.

Further investigation of the composition of code embeddings is needed. Pre-trained code embeddings are numerical vectors for individual tokens in the source code and a key challenge is how to represent the whole code snippet with a sequence of code tokens. In practice, multiple methods for code embedding composition (i.e., combining the embeddings of each token in the source code to an embedding of the entire code snippet) are adopted, such as simply adding or averaging the embeddings of all the tokens. Similar to that of Kang et al. [72], we find that the way of composition of code embeddings can impact the performance of the models. We implement four experiments for the task of software defect prediction with different composition strategies, i.e., summation, averaging, TF-IDF weighted averaging, and TF-IDF weighted summation. For example, when the four composition strategies are performed on code2vec for the task of software defect prediction, we get the F1 scores, 52.1, 58.7, 53.3, and 53.7, respectively. The results show that different composition strategies can result in diverse results and simple averaging still is the best way to represent the source code.

Code embeddings can benefit from considering both the structural and textual information. And the embeddings produced by StrucTexVec and code2vec which leverage structural information of the source code perform better than embeddings that do not. In our experiment, to check the effect of different information

Table 3.4: Comparison with the findings from previous work [72].

Our findings	Previous findings	Discussion
Models using pre-trained embeddings can perform better than models without pre-trained embeddings. The choice of embedding techniques has a non-negligible impact on the model performance.	Code embeddings cannot be used readily to improve the performance of simpler models.	In our evaluation, we find that in all six downstream tasks, the use of code embeddings can improve the performance of the models. The difference in the findings may be caused by (1) the training corpus, in this work, we select the Java-Small dataset; (2) we evaluate more embedding techniques on more downstream tasks.
Simpler baselines may not always outperform complex techniques. The model parameters have a non-negligible impact on the performance of the deep learning-based models.	Simpler baselines run faster and may outperform complex techniques.	In our evaluation, we see that on the task of code authorship identification, the use of code embeddings outperforms the simpler approach that uses simple TF-IDF features reported by Kang et al. [72]. The only difference is that we set the batch size to 64 instead of the default 128. Besides, to verify our findings, we implement another two simpler baselines which use traditional features for the software defeats prediction. As the results show, some of the embeddings still perform better. Meanwhile, our results concur with the findings of Kang et al. [72] that other considerations (e.g., pre-processing) may have an impact on the performance of deep learning models.
Using pre-trained embeddings may not always improve the performance of downstream tasks significantly.	Code embeddings may not be able to boost the performance of neural network-based models.	Our findings are similar to that of Kang et al. [72]. Although we can obtain an increase in the performance of different tasks, the improvement may be limited. One possible reason is that the neural network-based models themselves are powerful enough and already have good results, thus it is difficult to have a large improvement for some downstream tasks.

Composing a meaningful representation from a set of code tokens using pre-trained code embeddings is a challenging task and further investigation of the composition of code embeddings is needed.	The composition of source code token embeddings requires further investigation.	To further investigate the composition of code embeddings, we do another four experiments for the task of software defect prediction, using different composition strategies. i.e., summation, averaging, TF-IDF weighted averaging, and TF-IDF weighted summation. We find that simple averaging still is the best way to represent the source code.
Code embeddings can benefit from considering both the structural and textual information. And the embeddings produced by StrucTexVec and code2vec that leverage structural information of the source code perform better than other non-contextual embeddings that do not.	The poor performance of utilization of code embeddings may indicate that token embeddings learned over source code may not encode enough information usable for different downstream tasks.	In our experiment, to check the effect of different information on generating the code embeddings, we use different training contexts and design ablation experiments. The results indicate the embeddings can encode different information for downstream tasks.

on generating the code embeddings, we use different training contexts and design ablation experiments. The results indicate that except for the training techniques, the training context also impacts the quality of the code embeddings. Especially, the embeddings (i.e., StrucTexVec and code2vec) that incorporate the structural context yield better results than other non-contextual models. The results show that the embeddings can encode different information for downstream tasks.

3.6 Threats to Validity

This section discusses the threats to the validity of our work. We consider three types of threats.

External validity. One major threat of using pre-trained code embeddings in downstream tasks is the computational costs of training the embeddings. In our work, the embeddings of StrucTexVec, GloVe, fastText, and Word2vec are all trained in CPUs and code2vec is trained in an NVIDIA GTX 1080 Ti GPU, and it takes less than 30 minutes to finish the training process of each of the embedding techniques, which is acceptable compared to the computational cost of the running the downstream tasks (e.g., it takes more than 10 hours to finish the task of code comment generation). However, the training of CodeBERT and CuBERT, not only requires more GPUs but also several days to finish the training. For example, CodeBERT spends more than ten days to finish the training using 16 interconnected NVIDIA Tesla V100 GPUs, which might be challenging for us to train the embeddings in our own machine. The findings in this work are concluded based on the evaluation results of seven code embedding techniques on six downstream tasks, and the code embeddings are trained on Java projects. Thus, we cannot confirm that our findings may generalize to all the SE tasks, programming languages, and code embedding techniques. Besides, in this work, we adopt external SE tasks to reflect the impact of different code embedding techniques, and each task has its specific model to train, and thus the parameters involved during the embedding evaluation may have an impact on the conclusions. To minimize the influence of these parameters, in our work, we intentionally do not do any other parameter tuning on the model structures (e.g., layers, hidden dimensions, etc.) and try to use the same experimental settings that are reported in the literature, hence only examining the impact of different embedding techniques on the downstream tasks. Another factor that can influence the results and conclusions is the evaluation metrics used in the downstream tasks, as code embedding techniques that perform better under one evaluation metric may perform worse when evaluated using other metrics. In this work, to reduce the selection bias, we try to follow previous works and select the evaluation metrics that are used in the existing papers related to the downstream tasks. In addition, we provide our data and source code for future work to replicate and further improve the evaluation. For the only change of batch size in the task of code authorship identification, we want to explain that we initially set the batch size to 128 and observed that the precision remains at around 50% even on the training dataset, which was far behind the expected performance of deep learning models on the simple classification tasks. Thus, we reduce the batch size, and after that, the training precision improves to more than 90%. Also, this improvement confirms the finding from previous work that using small batch sizes achieves better training stability [117]. Moreover, the datasets used for different downstream tasks are unbalanced which would have an impact on the conclusions if we compare the results on the dataset level. As described in Section 3.3, all the datasets are provided by previous work and are commonly used benchmark datasets. To avoid such influence, we only focus on the task level comparison. Besides, there is a lack of ways for direct evaluation of the quality of code embeddings. Future studies can develop some datasets or tasks, such as token similarity, that can be directly used for code embedding evaluation.

Internal validity. One of the threats to the internal validity is related to the conclusion of results in response to RQ2. In order to answer RQ2, we conducted ablation experiments to analyze the impact of structural and textual context. However, the analysis in RQ2 may not indicate the actual impact of structural and textual information, as they may have overlappings. For example, in the AST representation of the source code, the code tokens are also considered during the embedding training. Besides, in our experiments, we only evaluate the embeddings that are trained based on the AST or plain text of the source code. However, there exists other information from the source code that may be more valuable for representing the properties of the source code. For example, [8] exploit the use of data flows in a program and [162] further extend the use of structured information extracted from the source code and they adopt the interprocedural program dependence for representing the source code into vectors. Another threat comes from the fact that there is a lack of interpretability of the code embeddings. The embeddings are real-valued numbers and are hard to analyze directly, thus, our findings cannot explain when and why the embeddings are effective.

Construct validity. The embedding training dataset selection may be biased, as we only select the top-ranked Java projects based on the number of stars they have, and we may still miss some Java projects that are from unpopular fields. Future studies can collect projects across different fields to complement the findings of our study. Another threat is the parameters of the code embedding techniques during embedding learning, such as the embedding dimensions and negative samples. Although, by fine-tuning the parameters, we can make the model better conform to the downstream tasks. However, in our work, we try our best to follow the literature and intentionally do not fine-tune the parameters to avoid bias from the unfairness among the tasks. We leave it as future work to analyze the impact of different combinations of parameter settings on the quality of the code embeddings.

3.7 Conclusion

In this chapter, we revisit and extend a recent study by Kang et al. [72] on the assessment of pre-trained code embeddings in SE tasks. Complementing the two evaluated pre-trained embedding techniques in prior work, we propose an unsupervised framework, StrucTexVec, for enhancing the learned code embeddings by incorporating both the textual and structural knowledge into the embedding training process. In total, we evaluate the effectiveness of seven techniques for pre-trained code embeddings on six downstream SE tasks. On one hand, we find that, in general, models using pre-trained embeddings can perform better than the models without pre-trained embeddings, and both the structural information and the textual information have a non-negligible impact on the performance of the downstream SE tasks. On the other hand, our work concurs with prior research that pre-trained embeddings may not always improve the performance of downstream SE tasks significantly, and different embedding techniques can lead to diverse results. Our results suggest the need for researchers and practitioners to carefully consider the choices of embedding techniques when conducting SE tasks. Our findings also shed light on future research for improving embedding techniques to assist in SE tasks.

Chapter 4

Towards Learning Generalizable Code Embeddings

Code embeddings have seen increasing applications in software engineering (SE) research and practice recently. Despite the advances in embedding techniques applied in SE research, one of the main challenges is their generalizability. A recent study finds that code embeddings may not be readily leveraged for the downstream tasks that the embeddings are not particularly trained for.

In this chapter, we propose GraphCodeVec, which represents the source code as graphs and leverages the Graph Convolutional Networks to learn more generalizable code embeddings in a task-agnostic manner. The edges in the graph representation are automatically constructed from the paths in the abstract syntax trees, and the nodes from the tokens in the source code. To evaluate the effectiveness of GraphCodeVec, we consider six downstream benchmark tasks. For each downstream task, we apply the embeddings learned by GraphCodeVec and the embeddings learned from four baseline approaches and compare their respective performance. We find that GraphCodeVec outperforms all the baselines in five out of the six downstream tasks and its performance is relatively stable across different tasks and datasets. In addition, we perform ablation experiments to understand the impacts of the training context (i.e., the graph context extracted from the abstract syntax trees) and the training model (i.e., the Graph Convolutional Networks) on the effectiveness of the generated embeddings. The results show that both the graph context and the Graph Convolutional Networks can benefit GraphCodeVec in producing high-quality embeddings for the downstream tasks, while the improvement by Graph Convolutional Networks is more robust across different downstream tasks and datasets. Our findings suggest that future research and practice may consider using graph-based deep learning methods to capture the structural information of the source code for SE tasks.

4.1 Introduction

Despite recent advances in code embeddings, one of the main challenges of applying such embeddings in research and practice is their generalizability to downstream tasks that the embeddings were not particularly trained for. Recently, Kang et al. [72] evaluate two pre-trained code embeddings generated by GloVe [142] and code2vec [11], by applying these two pre-trained embeddings to three downstream SE tasks, including code comment generation, code authorship identification, and code clone detection. However, the results show that code embeddings may not be readily leveraged in the models of the downstream tasks for which they have not been trained. In other words, pre-trained code embeddings may not generalize to different downstream tasks.

On the other hand, both studied embedding techniques in the prior work [72] have their limitations. In particular, GloVe [142] treats the source code as plain text and only considers the unstructured local textual information which may miss the useful syntax information from the source code. Code2vec [11] parses each method in the source code to an abstract syntax tree (AST) and focuses on the utilization of the structural information extracted from such ASTs. However, the token vectors are learned using a supervised approach, where the training objective is method name prediction instead of a task-agnostic purpose. Therefore, in this work, **we aim to find out whether the lack of generalizability of these code embeddings can be alleviated by learning task-agnostic embeddings from both the syntax and semantic information of the source code in a task-agnostic manner**.

Meanwhile, the recently proposed graph-based deep learning methods [93] have been successfully employed in several SE tasks such as variable name prediction [8] and variable misuse prediction [8]. However, such graph-based methods have not been used for learning source code embeddings. Therefore, in this chapter, we adopt the Graph Convolutional Networks (GCN) [33, 77] to learn code embeddings due to its ability to handle structural information in graphs. We first construct graph representations from the abstract syntax trees (ASTs) of the source code, then leverage the GCN model to train the code embeddings from the context information provided by the graph representations. Unlike previous work [8, 11, 204] which learns code representations for specific tasks, this work learns task-agnostic code embeddings, aiming to effectively apply the learned embeddings to different downstream SE tasks.

To quantitatively assess the quality of our learned code embeddings in SE tasks, we use and extend the existing benchmark tasks published by Kang et al. [72]. Specifically, we add three new downstream tasks to the existing ones, resulting in a total of six downstream tasks: code comment generation, code authorship identification, code clone detection, source code classification, logging statement prediction, and software defect prediction. We apply our learned code embeddings in these benchmark tasks and compare them with four baseline approaches. Specifically, we organize the discussion of our results along with the following three research questions (RQs).

RQ1 How effective is GraphCodeVec compared with other baseline embedding techniques in representing the source code? We compare GraphCodeVec with the other four state-of-the-art baseline embedding techniques in the six downstream tasks. We observe that GraphCodeVec outperforms the baseline approaches in five out of the six downstream tasks.

RQ2 How does the structural context information of the source code impact the effectiveness of the embeddings generated by GraphCodeVec? We perform an ablation experiment to understand the impact of the context information extracted from the structures of the source code (i.e., the ASTs) on GraphCodeVec. We find that although overall, such structural context information can benefit GraphCodeVec in producing code embeddings for the downstream tasks, there may be cases where the structural information may not provide additional benefit.

RQ3 How does the GCN model impact the effectiveness of the embeddings generated by GraphCodeVec? We perform another ablation experiment to understand the impact of the used model (GCN) for training the code embeddings. We find that using the GCN model performs better than using a shallow neural network as used in Word2vec.

The main contributions of this work include:

- We propose a source code embedding approach, GraphCodeVec, which represents the source code as graphs and utilizes the Graph Convolutional Networks (GCN) to learn task-agnostic code token representations.
- We conduct comprehensive experiments on the benchmark downstream tasks, which demonstrates that GraphCodeVec performs comparable or better than the existing approaches on all the studied downstream tasks.
- We perform ablation experiments to understand the impact of the important modeling decisions (i.e., training context and training model) on our approach and demonstrate that both the structural context information and the GCN model benefit our approach in producing more generalizable code embeddings.

- We share our trained embeddings and downstream tasks with the research community[1].

Chapter organization. In Section 4.2, we describe our proposed approach. Section 4.3 presents our experimental setup. Section 4.4 discusses the experimental results of evaluating GraphCodeVec along three research questions. In Section 4.5, we further discuss the impact of different parameter settings and different data sampling strategies on the performance of code embeddings. Section 4.6 discusses the threats to the validity of our study. Finally, Section 4.7 concludes this chapter.

4.2 Approach

Prior work [72] finds that pre-trained code embeddings may not be readily leveraged for the downstream tasks that the embeddings are not trained for. However, considering the limitations of the existing code embedding techniques, we propose GraphCodeVec,

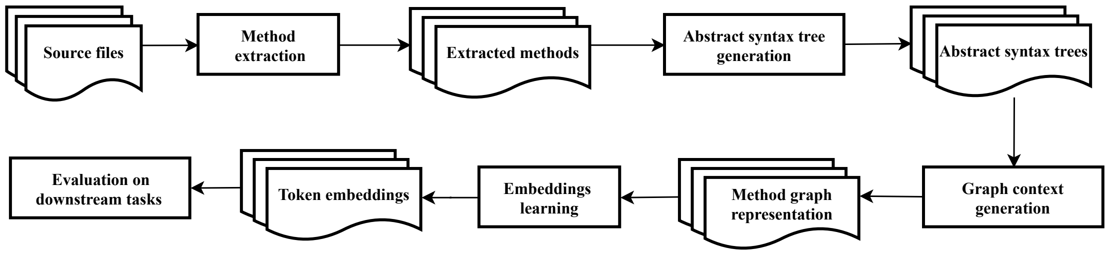


Figure 4.1: The overall framework of GraphCodeVec. Note: we apply the same token embeddings trained from a general dataset on all downstream tasks.

which consists of a training context preparation phase followed by an embedding learning phase. Figure 4.1 outlines the overall framework of GraphCodeVec. GraphCodeVec first extracts methods from a collection of source code files (i.e., Java classes), which are later transformed into AST representations. Based on these AST representations of methods, a context graph is then constructed for each extracted method. In the embedding learning phase, the GCN embedding approach [170] is used to train the token embeddings based on the graph context. Below, we describe the training context preparation and embedding learning phases in detail.

[1]The embeddings and downstream tasks are available at Google Drive.

4.2.1 Training Context Preparation

In this section, we describe the procedures of how to represent the source code using a graph. Formally, given a code snippet $\mathcal{D} = (w_1, w_2, \ldots, w_n)$, where w_n is the nth token in the code, the goal of this step is to generate its graph representation, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of nodes (i.e., tokens in the source code), $\mathcal{E} = \{e_{u,v} | u, v \in \mathcal{V}\}$ refers to the edges in the graph ($e_{u,v}$ represents the edge connecting nodes u and v).

4.2.1.1 AST generation

Apart from using the local window to construct the context, many NLP tasks adopt Syntactic Dependency Parse (SDP) to composite the context [78, 82, 84, 147, 194]. Meanwhile, previous studies [10, 18, 148] demonstrate that software engineering tasks can greatly benefit from leveraging the syntax information of programming languages. Hence, in this section, we follow a similar approach with that of Alon et al. [10] to extract the AST representations of source code.

In GraphCodeVec, source code is first transformed into ASTs using JavaParser[2], which provides the functionality of converting source code into tree representations. The structural syntax information of each method is preserved in an AST tree. For example, given the following code snippet, JavaParser produces the tree representation shown in Figure 4.3.

```
public void printName (String someone){
    name = someone;
    System.out.println(name);
}
```

Figure 4.2: An example code snippet.

As Figure 4.3 shows, the leaf nodes are tokens in the source code that are connected by a set of JavaParser AST node types that provide the syntax structure of the code.

Based on the AST, we then extract the nodes and edges from the AST and represent the source code using a graph. Our work shares a similar way with Alon et al. [10, 11].

[2]https://javaparser.org/.

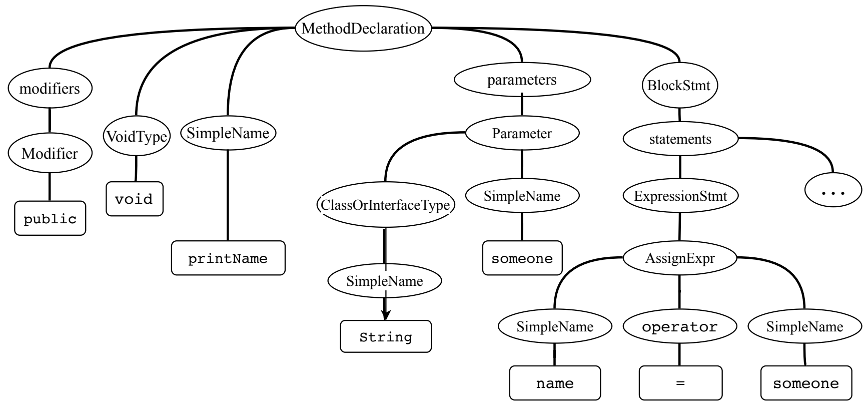


Figure 4.3: Tree representation of the code snippet generated by JavaParser. For simplicity, only part of the tree is displayed.

4.2.1.2 Graph context construction

Once we have the AST representation of each method of the source code, we start to construct the graph context. We first traverse the extracted ASTs (see Section 4.2.1.1) to collect all the leaf nodes for each method (i.e., code tokens in the source code). The collected leaf nodes are the nodes in the constructed graph. We adopt the depth-first search algorithm implemented in "`TreeVisitor`"[3] for the traversal. To construct a graph representation of the method, we also need to identify the AST node types connecting these leaf nodes. The identified AST node types are the edges in the constructed graph. Given any two different leaf nodes, w_1 and w_2, the edge, $e_{1,2}$ is the shortest path between these two nodes in the method's AST. We also keep the path traversing direction to preserve as much information as possible. As a result, we can collect two different type paths for each pair of leaf nodes. The reason why we preserve the path direction is that different paths represent different syntactic relationships between these nodes. For example, in our above example, "`name = someone;`", for the token "`name`", "`someone`" is the source expression (i.e., assigner), and for the token "`someone`", "`name`" is the target variable (i.e., assignee). In other words, the dependency relationship from "`name`" to "`someone`" is different from the dependency relationship from "`someone`" to "`name`". Moreover, the direction of the dependency relationship is not only considered in SE tasks (e.g., [11]) but also in NLP tasks (e.g., [84]). By doing such a directed structural traversal, we construct the graph representation of the source code, where nodes represent the code tokens in the source code while the edges represent the AST node types connecting two nodes. In the constructed graph, there are

[3]https://www.javadoc.io/doc/com.github.javaparser/javaparser-core/3.6.0/com/github/javaparser/ast/visitor/TreeVisitor.html.

N nodes and $N*(N-1)$ directed edges[4] describing the syntactic relationship between any two nodes, and N is the number of leaf nodes in the AST (i.e., code tokens in the source code).

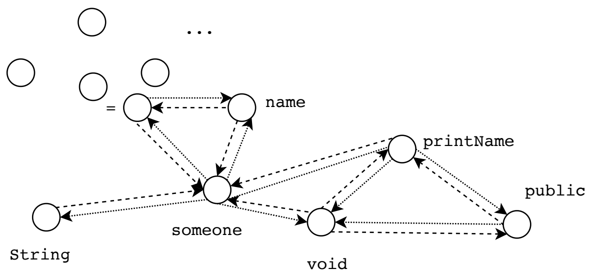


Figure 4.4: Graph representation of the code snippet based on the AST. For simplicity, only part of the graph is displayed.

Figure 4.4 illustrates a simple example of how to construct a graph from an AST. Basically, we look for the subtree rooted on the closest common ancestor of the source and the target leaf nodes. The detailed procedure is as follows:

1. Given an abstract syntax tree of a method, e.g., "`printName`", we first collect all the leaf nodes.

2. We then choose two of the leaf nodes as the target and source nodes (e.g., the parameter type "`String`" and the parameter name "`someone`"), respectively.

3. Next, we extract the paths from the root node, `MethodDeclaration`, to the target and source nodes respectively (i.e., ⟨`MethodDeclaration, parameters, Parameter,ClassOrInterfaceType, SimpleName`⟩ and ⟨`MethodDeclaration, parameters, Parameter, SimpleName`⟩). The longest common prefix of these two paths is ⟨`MethodDeclaration, parameters, Parameter`⟩.

4. We then remove the longest common prefix from the two paths, resulting in two sub-paths, ⟨`ClassOrInterfaceType, SimpleName`⟩ and ⟨`SimpleName`⟩. We keep the last element of the common prefix (i.e., `Parameter`).

5. We preserve the path direction from the target node to the source node and connect the path elements with $-$. Specifically, we reverse the sub-path connecting the target node and assign the up direction (represented as $\uparrow$). For the sub-path

[4]We further filter the edges by a length threshold, explained later in this section.

connecting the source node, we remain in the same order and assign the down direction (represented as ↓). For example, after this step, the two paths become `SimpleName`$^{\uparrow}$-`ClassOrInterfaceType`$^{\uparrow}$ and `SimpleName`$^{\downarrow}$.

6. We then concatenate the two sub-paths with the preserved last element of the common prefix (i.e., `Parameter`), `SimpleName`$^{\uparrow}$-`ClassOrInterfaceType`$^{\uparrow}$-`Parameter`-`SimpleName`$^{\downarrow}$. Finally, we have two nodes, "`String`" and "`someone`" and the edge connecting them, i.e., `SimpleName`$^{\uparrow}$-`ClassOrInterfaceType`$^{\uparrow}$-`Parameter`-`SimpleName`$^{\downarrow}$, where the ↑ and ↓ are the traversing directions and no direction means an inflection node of a traversing path.

7. We repeat steps (2) through (6) for each pair of source and target nodes, until we collect all the nodes and edges in the AST.

However, the number of edges is approximately the square of the number of tokens (i.e., leaf nodes). To reduce the size of the training data, we follow previous work [11] and limit the number of edges by a maximum length: if the length of an edge (i.e., the number of AST node types in the shorted path) exceeds the threshold, the edge will be ignored. In our work, we follow the work of code2vec [11], and set the threshold to eight as we find that two tokens connected by a longer edge usually do not have a direct structural relationship. Note that a relatively longer edge can preserve a more complete relationship between the leaf nodes, in other words, with a larger threshold, in the constructed graph, the target node can have edges to more other nodes, and thus, generating more training context. Meanwhile, if the threshold is too large, more indirect relationships with the target node would be included, which may introduce more noise to the training corpus, leading to poor quality of the generated code embeddings. And if the threshold is too small, although the target token would have a more direct relationship with other nodes, the number of connected nodes would be small and lead to insufficient training data. Thus, the threshold should be tuned for specific tasks or training contexts.

The output of our training context preparation phase (i.e., the graph context of code tokens) is used as the input for our embedding learning.

4.2.2 Embedding Learning

This section provides a detailed description of our approach to learning distributed token representations in a task-agnostic manner. More specially, in this work, we adopt the Graph Convolutional Networks (GCN) [170] to train the token embeddings based on the graph context generated in Section 4.2.1. The reason why we choose GCN is that it can not only preserve both the semantic information (i.e., leaf nodes in ASTs), but also the structural information (i.e., the connecting paths in ASTs) of the source code [8].

Figure 4.5 illustrates our embedding learning phase. Assuming the target token is "`someone`", the relevant context tokens (e.g., "`name`", "`String`", "`printName`", "`void`") are fed into the GCN model for predicting the target token, "`someone`". **Formally, given a graph representing the source code snippet, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the goal is to learn a d-dimensional embedding for each token in $\mathcal{V}$.**

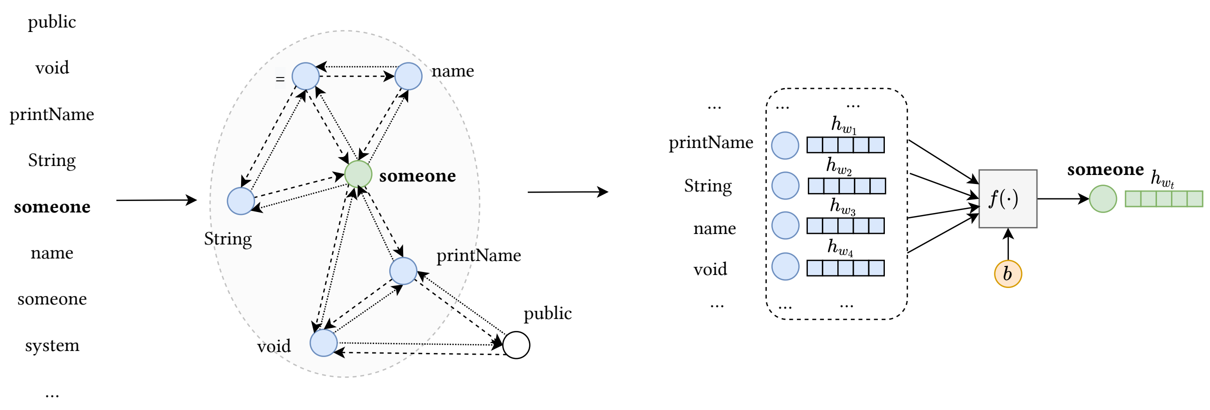


Figure 4.5: An overview of our embedding learning phase: assume the target code token is "`someone`", the nodes in blue are the relevant context tokens which are fed into a one-layer Graph ConvolutionalNetwork (GCN) for learning the distributed representations of the target token. h_{w_i}, h_{w_t} are the hidden representations of the context token and target token, respectively.

Similar to the Continuous Bag-Of-Words (CBOW) model [123, 124], which tries to predict the target token using its surrounding tokens within a local window, our approach utilizes the directly connected nodes (i.e., its neighbors), C_{w_t} to predict the given target node w_t.

Hidden representation for each node. The hidden representation (hidden state) of each node is the output of a convolutional layer in GCN. As Figure 4.5 shows, the hidden representation of the target token $h_{w_t} \in \mathbb{R}^d$ is updated based on its neighbors in the graph context. More specially, the representation for the target node w_t at the $(l+1)$th layer in GCN is computed by:

$$h_{w_t}^{l+1} = f\left(\sum_{w_c \in C_{w_t}} \left(W_{e_{w_c,w_t}}^{l} h_{w_c}^{l} + b_{e_{w_c,w_t}}^{l}\right)\right) \tag{4.1}$$

where $W^l_{e_{w_c,w_t}}$ and $b^l_{e_{w_c,w_t}}$ are a trainable weight matrix and a bias, and $h^l_{w_c}$ is the hidden representation for context node w_c at the lth layer.

Edge-wise gating mechanism. As described in Section 4.2.1, to reduce the number of edges in the graph, we do filtering using a threshold of edge length. In addition, there may exist different relationships among the leaf nodes: some are weak and meaningless, while others may be more meaningful. For example, we see in Figure 4.5 that even though the target token "`someone`" is directly connected with the token "`void`", their relationship is not meaningful. In comparison, the relationship between "`name`" and "`someone`" is stronger. Therefore, we should assign different weights to different context nodes when calculating the hidden representation for the target node.

To address this issue, we adopt the edge-wise gating mechanism [113]. For each target node w_t, the weight score with its context token w_c is calculated as follows:

$$g^l_{e_{w_c,w_t}} = \sigma\left(W'^k_{e_{w_c,w_t}} h^k_{w_c} + b'^k_{e_{w_c,w_t}}\right) \tag{4.2}$$

where $W'^k_{e_{w_c,w_t}}$ and $b'^k_{e_{w_c,w_t}}$ are trainable parameters and $\sigma(\cdot)$ is the sigmoid function. Thus, the hidden representation of the target nodes is formulated as:

$$h^{l+1}_{w_t} = f\left(\sum_{w_c \in C_{w_t}} g^l_{e_{w_c,w_t}} \times \left(W^l_{e_{w_c,w_t}} h^l_{w_c} + b^l_{e_{w_c,w_t}}\right)\right) \tag{4.3}$$

Training objective. Given a graph representation of the source code, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and the target node, w_t (the t^{th} node), the objective of the model is to maximize the following objective function:

$$\mathcal{L} = \sum_{w_t \in \mathcal{V}} \log P\left(w_t | \mathcal{C}_{w_t}\right) \tag{4.4}$$

where $\mathcal{C}_{w_t}$ is the context nodes (i.e., neighbors in the graph) of the target nodes w_t, $P\left(w_t | \mathcal{C}_{w_t}\right)$ is the conditional probability of observing the target node w_t given the context nodes, $\mathcal{C}_{w_t}$. $P\left(w_t | \mathcal{C}_{w_t}\right)$ is defined using the following *softmax* function:

$$P\left(w_t | \mathcal{C}_{w_t}\right) = \frac{\exp(\mathbf{v}_{w_t}{}^\intercal h_{w_t})}{\sum_{w \in \mathcal{V}} \exp(\mathbf{v}_w{}^\intercal h_{w_t})} \tag{4.5}$$

where $\mathbf{v}_w$ and h_w denote the target embedding and hidden representation of the node w, respectively.

Optimization. One issue in GraphCodeVec is the high cost of computation in the *softmax* function (i.e., Equation 4.5) because it involves the iteration through every node over $\mathcal{V}$. To address this issue, different optimization strategies can be applied, such as hierarchical softmax and negative sampling [124]. Hierarchical softmax [126, 128] uses a binary tree to represent the tokens in the vocabulary, $\mathcal{V}$, where each leaf node of the tree is a token. The probability of traversing from the root to the leaf node (i.e., target token) along the unique path is used to estimate the conditional probability. By doing such an approximation, the complexity of calculating the probability of each word goes down from $O(|\mathcal{V}|)$ to around $\log_2(|\mathcal{V}|)$ [107, 150]. While negative sampling is more straightforward [107, 150]. The idea of negative sampling is to update a small sample of the token vectors rather than all of them, such that the computing cost of the *softmax* function can be reduced. In this work, following previous work [51, 142, 170], we adopt the negative sampling, as it tends to give better results than hierarchical softmax [52, 142].

The output of our embedding learning phase (i.e., the token embeddings) are used as the input for our downstream tasks for evaluation.

4.3 Experimental Setup

In this section, we present details of our embedding training settings and describe the six downstream tasks used in our quantitative evaluation. Three of the SE tasks, i.e, (1) code comment generation, (2) code authorship identification, and (3) code clone detection, are used for the evaluation of code embeddings in prior research [72]; while the other three, i.e., (4) source code classification, (5) logging statement prediction and (6) code defects prediction, are newly added in our extended benchmark. We select these tasks either due to the fact that they are chosen for evaluating code embeddings in previous work [72], or they are of great importance for SE community and commonly studied in the literature.

4.3.1 Dataset Preparation

In our experiments, the dataset used for embedding learning comes from the *Java-small* dataset[5], which is provided by Alon et al. [11] and originally based on the dataset of Allamanis et al. [7]. This dataset is collected from publicly available open-source GitHub repositories.

Following the previous approach for pre-processing the source code [11, 28, 72], we convert the tokens into lower cases and remove all the non-identifiers (e.g., quotation marks). Meanwhile, we follow the common practice [123, 124, 142, 170] and ignore all

[5]https://s3.amazonaws.com/code2vec/data/java-small_data.tar.gz

tokens with a total frequency of less than five as there is not enough data to do any meaningful training on those rare tokens [19, 149, 182]. While constructing the graph representation, due to the limitation of the memory, we only keep the top-100 most frequent edge types (i.e., edges with the identical path representation) and others are replaced with a unique identifier (i.e., -1). As during embedding learning, we need to batch the training context with different edge types into the GCN model, and in the GCN model, we create an adjacency matrix for each edge type, that means if there are a large number of edge types, the model requires more memory to keep these matrices and would run out of memory and cannot be moved to GPU for embedding training. Besides, as recommended by Vashishth et al. [170], we also limit the size of each graph to a maximum of 100 unique nodes and 800 edges; that is, if the size of the graph exceeds the threshold, the graph will be removed from the training set. After preprocessing, we collect 637,108 training methods (there are 665,115 methods before the preprocessing), each of which is represented by a graph for subsequent embedding learning. In this work, considering the fact that code2vec can only be trained on method level corpus (cf., Section 2.1.2.1), to have a fair comparison with these baselines, we only construct the method level graph context. However, as ASTs can represent the source code with different levels (e.g., method level, statement level, class level, etc.), our method can also be applied to other types of training data.

The datasets used in the downstream tasks may have different vocabulary from the training dataset, *a.k.a*, the out-of-vocabulary (OOV) problem. To handle the OOV tokens, we choose to randomly initialize the vector representation of tokens that only appear in downstream tasks to minimize the impact of these unseen tokens (i.e., to make tasks with OOV vocabulary predictable). By doing this, we can make sure that all tokens in downstream tasks have vector representations, therefore it is always predictable (but may lead to poor performance as the vectors representing these OOV tokens are not learned from their context).

4.3.2 Training Details

While training the model, we follow the settings in prior work [11, 72, 204] and set the dimension of token vectors to 128. To prevent overfitting and avoid performance degradation, we set the number of GCN layers to 1, as GCN tends to suffer performance degradation with increased depth (i.e., number of layers) [26, 85, 90, 151, 209]. The training batch size is set to 64 by default. Considering (1) the small number of weights of our model (i.e., one layer and 128 input dimensions), (2) the relatively large size of the training data (i.e., more than half a million graphs), and (3) the remarkable learning ability of GCNs from the graph data, we train our embeddings for one epoch and the training loss is small enough. This is consistent with the finding of Mikolov et al. [123], that is for word embeddings, training a model on a relatively large dataset using one epoch gives comparable or better results than more epochs on the same dataset. As indicated in a prior work by Mikolov et al. [124], the number of negative samples in the range of two to five is useful for large training datasets and five to 20 for small training data. Hence, in this work, to balance efficiency and accuracy, we set the number to five. The training of our embeddings is conducted in a machine with an NVIDIA GTX 1080Ti GPU and 32GB memory. We summarize the thresholds and hyperparameters used in our experiment in Table 4.1.

We evaluate the quality of the trained embeddings on six downstream tasks. For the downstream tasks that use neural network-based models, the embeddings are used to initialize the embedding layer of neural networks, as changing the embeddings of the embedding layer would affect the way the model is learned and thus the models with different code embeddings would have different performance. For the downstream tasks that use traditional machine learning models, the embeddings are used as feature vectors (i.e., each dimension of the embeddings is treated as a feature). For example, we have a code snippet "`String name = someone`", and each token ("`string`", "`name`", and "`someone`"; "`=`" is removed) within the vocabulary has its corresponding vector representation, such as [0.1, 0.2, 0.3, ...], [0.1, 0.1, 0.1, ...] and [0.2, 0.2, 0.3, ...], these vectors can be summed up (or other operations) as a feature vector (each feature is one dimension of the embedding), which later can be used for traditional machine learning models.

4.3.3 Baselines

To evaluate the effectiveness of our trained embeddings, we compare GraphCodeVec with the following existing embedding models (i.e., the baselines):

- **Word2vec**[6] is a popular unsupervised word embedding method proposed by Mikolov et al. [123]. We use the implementation in Gensim[7] [149].
- **GloVe** is an unsupervised algorithm using token-token co-occurrence statistics, proposed by Pennington et al. [142].
- **fastText** is proposed by Facebook's AI Research lab [19]. It is an unsupervised algorithm, which utilizes the subword information to enrich the word vectors. In

[6]There are two variants in the implementation of Word2vec (i.e., Skip-gram and CBOW) and two different optimization strategies (i.e., negative sampling and hierarchical softmax). Following previous work [170], we here select the CBOW with negative sampling as a representation for comparison.

[7]https://radimrehurek.com/gensim/

Table 4.1: Hyperparamters and thresholds used during the two stages of GraphCodeVec for generating the code embeddings.

Stage	Name	Default value	Description
Training context generation (cf., Sec. 4.2.1 and RQ2)	Edge length	8	Edge length is the number of AST nodes connecting two leaf nodes (i.e., code tokens). It would influence the quality and quantity of the training context. A smaller value would result in a tighter connection but fewer connected nodes to the target token, leading to not enough training context. On the contrary, a larger value may include more unrelated token pairs and introduce noise to the training context. In our work, we follow the work of code2vec [11] and set it to eight to make a fair comparison.
	Unique node	100	Unique node refers to the number of unique tokens within each method. This parameter would influence the size of each constructed graph for training. The values should be tuned based on the GPU memory size, as we need to batch the graphs into the GPU for training, if the graphs are too large, the model requires more memory to keep these data and would throw an "out of memory" error. Vashishth et al. [170] suggest the number of unique nodes should be set to no larger than 100. And in our settings, only about 4% of the graphs are filtered which not only has a small effect on the quantity of the training context but also can avoid the memory error.
	Edge	800	Edge refers to the total number of extracted AST node types within each method. It has a similar effect with the parameter Unique node on the size of the contracted graph. Also, the values should be tuned based on the GPU memory size, in our work, we set this parameter to 800, as we find that only about a small portion (i.e., 4%) of the graphs are filtered out.
	Window (cf., RQ2)	5	Window (cf., RQ2) is the maximum distance between the current and its neighboring word within a method. It is similar to edge length, which also has an impact on the constructed training context. A larger window size would be able to capture broader context, but with the possibility of introducing noise, as the context tokens might not be tightly related to the target token. On the contrary, a smaller window size may contain more focused information about the target word but may not be able to capture sufficient context. Setting the context window size to five is commonly done in the literature [19, 82, 123, 124, 149].
Embedding learning (cf., Sec. 4.2.2)	Layer	1	Layer refers to number of layers in GCN. It controls the depth of a GCN and directly influences the quality of the model. Previous work [26, 85, 90, 151, 209] shows that GCN tends to suffer performance degradation with increasing depth (i.e., number of layers). In our work, we follow the work of [170] and use the default value.
	Dim.	128	Dim. is the dimensionality (i.e., vector size) of each token. It has a non-negligible impact on the quality of the embeddings. A small vector size cannot preserve the properties of the tokens of high dimensional spaces, leading to the degradation of quality of learned embeddings. However, a too large size requires more computing resources and training time, and may suffer the sparsity problem if the training data is not enough. In our work, we follow the settings in prior work [11, 72, 204].
	Neg.	5	Neg. refers to the number of negative samples used when updating the wights of the model. A larger value means more samples to calculate and thus more training time needed. This parameter should be adjusted based on the size of training context. As suggested by Mikolov et al. [123, 124], 5-20 samples works well for smaller datasets, and 2-5 words for large datasets.
	Batch size	64	Batch size defines the number of training samples presented in a single batch. A larger size can speed up the training process but requires more GPU memory [15] while using small batch sizes achieves better training stability [117]. In this work, we use the default 64.
	Dropout rate	0	Dropout rate is the probability of dropping a unit out. Dropout is a regularization technique for avoiding the model overfitting. As larger models (more layers or more units) tend to more easily overfit the training data [151, 161] and considering the small size of our model, we don't use this strategy, instead, we reduce the training epochs to avoid overfitting.
	Epoch	1	Epoch is the number of iterations through the entire training dataset. This factor affects the performance of the embeddings directly. Increasing the number of epochs may overfit the model and a small number of epochs may lead to a not fully trained model. In our work, considering the size of the training dataset and the number of weights of our model (i.e., one layer, 128 input dimensions) [123, 124], we train our model for one epoch, as the training loss is small enough.

their approach, each word is represented as a bag of character n-grams, and the word is represented as the sum of these character n-grams representations. We use the implementation in Gensim[8] [149].

- **code2vec**[9] is a recently proposed supervised model for source code representation. Prior work [72] evaluates code2vec on three downstream SE tasks. This model is proposed by Alon et al. [11] and utilizes the AST information to learn code embeddings.

We train these embeddings on the same dataset that is used for training our GraphCodeVec embeddings (i.e., the *Java-small* dataset). To make a fair comparison, we do the same preprocessing as in Section 4.3.1, that is converting the tokens into lower cases and removing all the non-identifiers, as well as ignoring all tokens with a total frequency lower than five.

4.3.4 Downstream Tasks for Evaluation

To control the quality of the embedding evaluation experiments, we enrich the work of Kang et al. [72] by adding three new tasks and adopting different modeling methods for the six tasks, including deep learning approaches and traditional machine learning methods. Specifically, for the first five tasks, including (1) code comment generation, (2) code authorship identification, (3) code clone detection, (4) source code classification, and (5) logging statement prediction, we use neural network-based approaches; while for the task of (6) software defect prediction, we follow the approaches used in their original work and adopt traditional machine learning methods (i.e., logistic regression, LR in short). More details can be found in Section 3.3.3.

We intentionally select both the deep learning and the traditional machine learning approaches to ensure the code embeddings are adequately evaluated across different tasks (i.e., six downstream tasks) and modeling approaches (i.e., traditional machine learning and deep learning). However, we specifically select LR for the only task of software defect prediction due to the fact that most of the downstream tasks that rely on code embeddings use deep learning models, thus we only select one task and put more focus on the impact on deep learning models. We run the experiments with 10-fold cross-validation to mitigate the effects of the random separation of the training and test sets, and report the average scores of the results of the 10-fold cross-validation. For the models' selection for downstream SE tasks, we follow the rules that (1) used in previous work [72, 139, 179], and (2) commonly used and have the state-of-the-art or competitive results [75, 204]. To further ensure a fair

[8]https://radimrehurek.com/gensim/
[9]https://github.com/tech-srl/code2vec

comparison with baselines, we either follow the parameter settings in previous work or use the default parameters and avoid fine-tuning these settings only for our method.

In the evaluation, our focus is the effectiveness of different embeddings instead of the approaches for the specific tasks themselves. Thus, we do not aim to reach the SOTA for a specific task. Moreover, for each downstream task, we try to use the same experimental settings that are reported in the literature, hence only examining the impact of different embedding techniques on the downstream tasks.

4.4 Experimental Results

In this section, we discuss our experimental results of the evaluation of our proposed approach, GraphCodeVec, organized along three research questions (RQs). For each RQ, we explain the motivation and the approach before discussing the corresponding results.

RQ1: How effective is GraphCodeVec compared with other baseline embedding techniques in representing the source code?

Motivation

Prior research [11, 21, 29, 43, 57, 63, 169, 204] proposes different distributed code representations (i.e., code embeddings) approaches to assist in software engineering tasks (e.g., method name prediction and software vulnerability prediction). However, a recent study by Kang et al. [72] finds that code embeddings may not be readily leveraged to enhance existing models for the downstream tasks which they have not been trained for. Therefore, in this research question, we would like to explore whether our task-agnostic GCN-based approach (i.e., GraphCodeVec) can produce more generalizable token embeddings for a variety of SE tasks compared with other baselines.

Approach

To answer our first research question, we need to train the token embeddings produced by different embedding techniques (i.e., GraphCodeVec and baselines, cf., Section 4.3.3). As shown in Figure 4.6, during code embeddings training, the same preprocessed training dataset (i.e., the *Java-small* dataset, cf., Section 4.3.1) is used by different embedding techniques.

Once we finish the code embedding training, we then need to evaluate these pre-trained embeddings. However, as there is no direct evaluation methodology for evaluating the

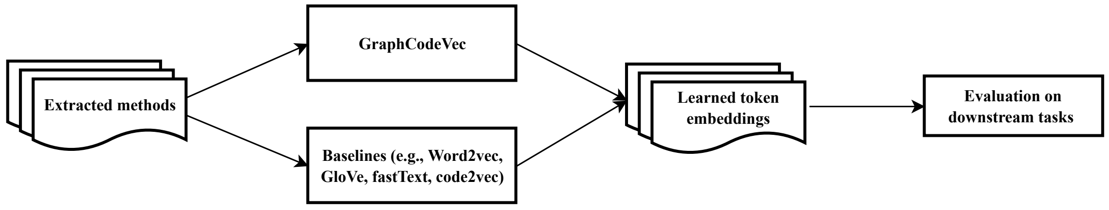

Extracted methods
GraphCodeVec
Baselines (e.g., Word2vec, GloVe, fastText, code2vec)
Learned token embeddings
Evaluation on downstream tasks

Figure 4.6: The overall design of the approach for RQ1. In this experiment, the same preprocessed dataset is used by GraphCodeVec and baselines.

quality of code embeddings, we thus follow previous work [72] and use six downstream SE tasks to evaluate the quality of code embeddings. Each of the tasks has its respective dataset for model training and evaluation, and the only varying factor in each evaluation task is the code embeddings (produced by GraphCodeVec and baselines) used for code token representation (i.e., for each task, only the embeddings are changed, and other parameters are kept the same), and thus we can conclude the performance changes are caused by the code embeddings. Note that the change of code embeddings would also impact the weights learned for each model, which is discussed in Section 4.6. The detailed description of the downstream SE tasks and the corresponding evaluation metrics are presented in Section 4.3.4.

Results

Overall, GraphCodeVec performs comparable or better than all baseline approaches on all downstream tasks. The experimental results are provided in Table 4.2 with the best results for each task and dataset highlighted in bold. In particular, GraphCodeVec achieves the best results in five out of the six tasks[10], including the tasks of code authorship identification, code clone detection, source code classification, logging statement prediction, and software defect prediction. To better illustrate the results, we specifically compare with GloVe, as did in Kang et al. [72], since it was one of the most important work aiming for generating task-agnostic embeddings at the time of our research. Then, we conduct a statistical analysis using a Wilcoxon signed-rank test to compare the performance of GraphCodeVec and the performance of GloVe. We use a p-value that is below 0.05 to indicate that the performance difference is statistically significant. For the differences that are statistically significant, we further compute the Cliff's delta effect size. The reason why we use the Wilcoxon signed-rank test and Cliff's delta is that they both do not assume a normal distribution of the compared data. As shown in Table 4.2, GraphCodeVec performs better than Glove in 16 out of 23 cases, and 68.8% of the improvements are statistically significant with a magnitude of large. We obtain a 5.0% relative increase

[10]We compute the average score across all datasets for each task, and GraphCodeVec achieves the highest average score for five out of six downstream tasks.

Table 4.2: Evaluation results of using GraphCodeVec and baselines on the test sets in the six downstream tasks.

				Baselines			
Downstream Tasks	Evaluation Metrics	Dataset	GraphCodeVec	Word2vec	GloVe	fastText	code2vec
Code comment generation	BLEU	GitHub	20.7(-4.7%)^{*L}	21.1	**21.7**	19.9	21.0
	ROUGE		36.1(-2.4%)^{*L}	36.9	**37.0**	36.0	36.3
Code authorship identification	Accuracy	Google Code Jam	**80.2(+1.1%)**	78.9	79.3	76.6	79.4
Code clone detection	F1	BCB	93.4(+0%)	93.4	93.4	93.4	93.4
		OJClone	**93.8(+8.7%)**^{*L}	88.4	86.3	84.6	93.4
Source code classification	Accuracy	OJ dataset	**93.7(+5.0%)**^{*L}	85.5	89.2	76.7	91.4
Logging statement prediction	BA	Airavata	**95.7(+0.7%)**	95.3	95.1	95.1	95.0
		Camel	**81.4(+0.4%)**	80.9	81.1	79.8	80.5
		CloudStack	86.3(-0.8%)	86.5	**87.0**	86.7	86.1
		Directory-Server	**89.1(+2.5%)**	87.9	86.9	88.6	87.6
		Hadoop	75.6(+0.7%)	**75.7**	75.0	74.4	73.9
Software defect prediction	F1	Ant 1.5 -> 1.6	42.7(+23.5%)^{*L}	35.9	34.6	36.0	**47.5**
		Ant 1.6 -> 1.7	**50.5(+13.0%)**^{*L}	43.9	44.7	44.2	46.8
		Camel 1.2 -> 1.4	**44.6(+5.3%)**^{*L}	41.9	42.3	41.8	42.7
		Camel 1.4 -> 1.6	46.7(-4.6%)^{*L}	45.3	49.0	45.8	**49.6**
		jEdit 3.2 -> 4.0	57.0(-2.3%)	53.4	58.3	53.6	**57.9**
		jEdit 4.0 -> 4.1	58.0(-3.3%)^{*L}	**61.0**	60.0	60.7	58.5
		Log4j 1.0 -> 1.1	**72.5(+9.1%)**^{*L}	64.0	66.5	63.1	68.5
		Lucene 2.0 -> 2.2	**67.0(+9.3%)**^{*L}	63.1	61.3	63.2	63.2
		Lucene 2.2 -> 2.4	65.2(+2.4%)^{*L}	**65.4**	63.7	65.3	62.4
		POI 1.5 -> 2.5	**84.6(+4.0%)**^{*L}	65.7	81.4	65.1	82.1
		POI 2.5 -> 3.0	**74.9(+2.6%)**^{*L}	72.5	73.0	72.2	74.0
		Xalan 2.4 -> 2.5	**52.5(+24.1%)**^{*L}	42.5	42.3	42.4	51.2

Note: The best results for each task and dataset are highlighted in bold. The numbers in the brackets indicate the relative change of GraphCodeVec to GloVe. The * means that the difference is statistically significant. The superscript L represents large effect size.

in accuracy on the source code classification task compared to the representative baseline (i.e., GloVe). Moreover, for the evaluation on the task of software defect prediction, which uses a traditional machine learning approach (i.e., Logistic Regression), our embeddings reach the best results on more than half of the datasets. For the Log4j dataset, we obtain around 10.1% absolute increase (24.1% relative increase) in the F1 score compared to that of GloVe. The results demonstrate that the learned embeddings from GraphCodeVec can better represent the source code and generalize to various downstream tasks. Besides, we find that on the task of code authorship identification, by using the code embeddings generated by GloVe, we achieve an accuracy of 79.3% and outperform the simpler approach in the work of Kang et al. [72] which uses the TF-IDF features. This finding is different from that of Kang et al. [72]. The difference may be caused by the different preprocessing steps on the training corpus and parameters for GloVe training. This finding suggests that we should be careful with the parameter selection and corpus reprocessing. To further investigate the influence of these factors, we have conducted more than 20 new experiments with different experimental settings, the results are discussed in Section 4.5 and Section 4.6.

However, we observe that for some downstream tasks (e.g., source code classification and code authorship identification), different embedding techniques can result in diverse performance. In particular, for the source code classification task, using the embeddings trained by fastText can only have a 76.7% of test accuracy, compared to an 89.2% test accuracy when using the embeddings trained by GloVe. This finding suggests practitioners should be careful with the selection of code embedding techniques for different downstream tasks, as they may produce diverse results. On the other hand, we also observe that leveraging different embeddings may not always impact the performance of downstream tasks significantly. This observation is similar to that of prior studies [72, 84, 170]. We find that by using different embeddings, although we can obtain different performances on different tasks, the difference is limited in some cases. For example, the different embedding techniques result in the same F1 score of 93.4% on the BCB dataset for code clone detection. One possible explanation is that the approaches used in the SE tasks are already powerful enough and there is enough dataset for learning a good model. Thus, the impact of using different embedding techniques may be negligible.

Compared to other embedding techniques, GraphCodeVec produces more stable results across all the downstream tasks and datasets. Figure 4.7 shows the comparison of performance results produced by GraphCodeVec and baselines. In this figure, to show the difference to the best performance of each task, the results are scaled to the range of 0-100%, which is the ratio of the current method's performance to the best performance of one task, and in each boxplot, we consider all the measures for all the datasets (i.e., there are 23 data points in each boxplot). We did not rank all the results and check the overall ranking of each technique, because different downstream tasks use different measures with different ranges. Thus, for each downstream task, we normalize the performance of each technique against the best performance across all techniques (i.e., we use scaled performance). The scaled performance has consistent ranges across different downstream tasks thus allowing better comparison and visualization of the performance of different techniques. We also calculate the coefficient of variation (CV) for all the embedding techniques to quantify the variances. The results show that GraphCodeVec has a relatively lower variance among all the tasks. For example, the biggest relative difference appears in the task of software defect prediction on the Ant dataset, which is 42.7% compared to the best result, 47.5%. Meanwhile, code2vec also has a stable performance on SE tasks, but its median is lower than that of GraphCodeVec. On the contrary, some embedding techniques lead to unstable results. For example, the fastText embedding technique achieves the best results on the BCB dataset but the worst result (i.e., 84.6% compared to the best result, 93.4%) on the OJClone dataset for the code clone detection task. Future work that depends on embedding techniques should consider a stable technique such as GraphCodeVec, otherwise the performance may be compromised.

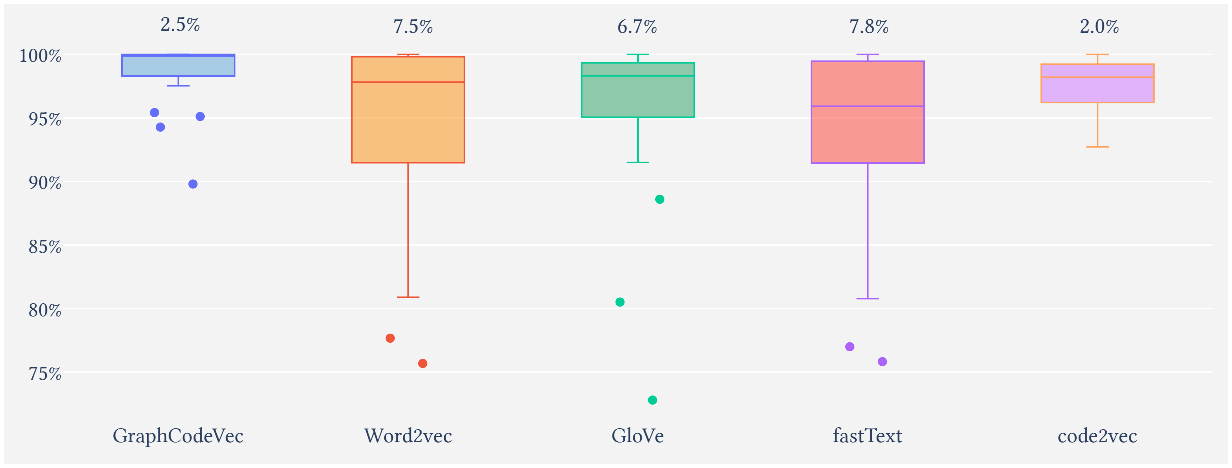


Figure 4.7: Comparison of the results of GraphCodeVec and baselines. The horizontal axis represents all the evaluated methods; the vertical axis is the scaled performance of different methods, which is calculated as the ratio of the current method's performance to the best performance of one task. The numbers on top of each box are the corresponding coefficient of variance.

Discussion

In the above paragraphs, we have quantitatively demonstrated the superiority of GraphCodeVec on the six downstream tasks, thus in this part, we would like to discuss the limitations of GraphCodeVec, as well as provide a qualitative analysis of the learned embeddings to complement our quantitative evaluation on downstream tasks.

Strengths and limitations

As shown in Table 4.2, although, overall, GraphCodeVec performs the best compared to all baseline approaches on five out of six downstream tasks, there is still a non-negligible gap between GraphCodeVec and GloVe on the task of code comment generation. By comparing the natural property of these tasks, we find that our GraphCodeVec works better on the classification tasks, such as code authorship identification, code clone detection, and source code classification, etc., but not on the text generation task (i.e., code comment generation). For the classification tasks, the output is pre-defined labels and the embeddings only work

Table 4.3: The agreement of the results between GraphCodeVec and baselines on the task of code clone detection.

	Word2vec	GloVe	fastText	code2vec
OJClone	0.82	0.77	0.75	0.89
BCB-Type-1	1.00	1.00	1.00	1.00
BCB-Type-2	1.00	1.00	1.00	1.00
BCB-Type-3 (Strongly)	0.99	0.99	0.99	1.00
BCB-Type-3 (Moderately)	0.99	0.99	0.99	0.99
BCB-Type-4	1.00	1.00	0.99	0.99

Note: We use Cohen's kappa to measure the agreement between the results generated by our method and that of the other four baselines.

in the first embedding layer which converts the source code tokens to real number vectors. However, for the task of code comment generation, we use an encoder-decoder architecture where in the encoder part, similar to classification tasks, the embeddings are utilized to transform source code tokens into vectors, while in the decoder part, the same code embeddings (instead of word embeddings trained on comments or texts) are also used to convert the comment tokens (i.e., extracted from code comments) into vectors. As the code tokens and comment tokens are naturally different and thus, using only one code embedding for both source code and code comments would confound the model, in other words, one good code embedding may not perform well on texts. Thus, we conclude that the poor performance may be caused by the fact that GraphCodeVec is able to capture the properties of the source code, but the learned knowledge is too specific for the source code and thus cannot be transferred to natural language tokens. In future work, to improve the performance of GraphCodeVec on such text generation tasks, we can enhance the model by jointly learning the code and word embeddings based on the code and text information (e.g., documents and comments).

Moreover, we also observe that for some tasks or datasets, GraphCodeVec does not bring significant benefits. For example, GraphCodeVec has the same results on the BCB dataset with the other embedding techniques for code clone detection[11] but best performance (i.e., 8.7% improvement) on the OJClone dataset. Besides, similar results are also observed on the Camel dataset for logging statement prediction, where GraphCodeVec has a small improvement (i.e. 0.4%) compared to other embedding techniques but a relatively larger improvement (i.e., 2.5%) on the Directory-Server dataset. One explanation for this phenomenon is that larger training datasets may produce more powerful models and mitigate the differences between different embedding techniques. To obtain such fully trained models, one possible way is to collect enough training datasets. Thus, we check the sizes of the datasets, and we find that the size of the BCB dataset is almost twice larger than that

[11] We further check the results, and as Table 4.3 shows, all the code embeddings almost produce identical (clones) results on the BCB dataset.

of the OJClone dataset and the size of the Camel dataset is more than five times larger than that of Directory-Server. The findings highlight that GraphCodeVec can work better for downstream tasks that have small training datasets. In other words, if the model cannot learn enough knowledge from the training dataset, we can use the embeddings generated by GraphCodeVec, as it can bring more external knowledge to the trained model, which is another ultimate goal of the pre-trained embeddings (i.e., learning useful knowledge from external datasets to improve the performance of downstream tasks).

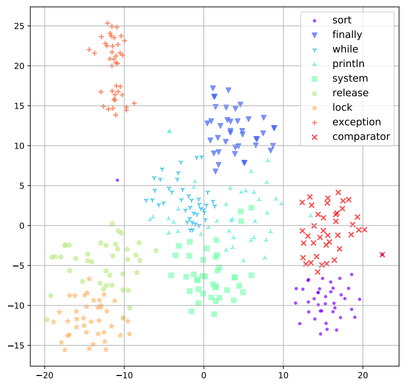
Qualitative analysis of the learned embeddings

To further understand the trained embeddings, following prior work [11, 166], we discuss the characteristics of the trained embeddings from a qualitative perspective. We manually inspect code embeddings on one qualitative task, i.e., token similarity, as it is usually considered the most straightforward feature to evaluate token representations [11, 123, 124, 166].

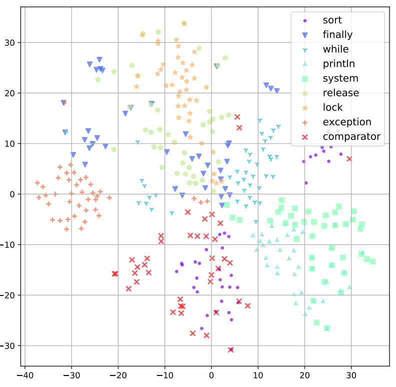
We select the target tokens and query their most similar tokens and then explore them intuitively. However, we should be aware that there is no explicit guideline for selecting the representative tokens, thus qualitative analysis might be subjective. In this work, we try our best to avoid the bias and select the subject tokens based on the following three criteria: (1) tokens should be well-known in the vocabulary - to ensure that evaluators are familiar with the characteristics of the tokens, (2) some of the tokens should provide different functionalities - to ensure that their embeddings have a low similarity and thus are located far from each other in the semantic space, and (3) some of the tokens should share similar functionalities - to ensure that their embeddings have high semantic similarity.

Following prior work [11, 84, 166], we manually chose nine tokens from the vocabulary with different frequencies. All the selected tokens are either Java reserved words (e.g. "`println`" and "`finally`") or frequently used methods (e.g. "`sort`") and some of them share similar functionalities (e.g., "`sort`" vs. "`comparator`") and others provide different functionalities (e.g., "`while`" vs. "`sort`"). For each chosen token, we retrieve its 40 most similar tokens (using cosine similarity) according to different embeddings.

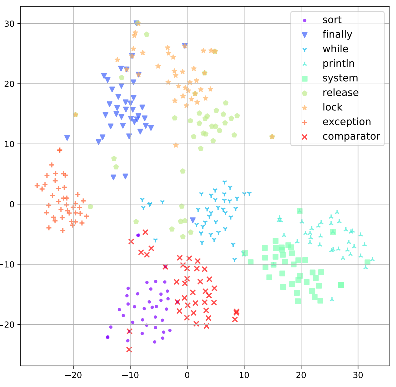
Visualization. In order to visualize high-dimension (i.e., 128 dimensions) embeddings, we compress them down to a low-dimensional space (i.e., two dimensions) using t-SNE [112]. The idea of t-SNE is to reduce dimensions while trying to preserve the information of the original data points, namely, keeping similar tokens close on the plane while maximizing the distance between dissimilar tokens. We plot the target tokens and their most similar tokens. A good code embedding should project similar (e.g., similar functionalities) code tokens into the space with a shorter distance and project the unrelated code tokens far from each other.



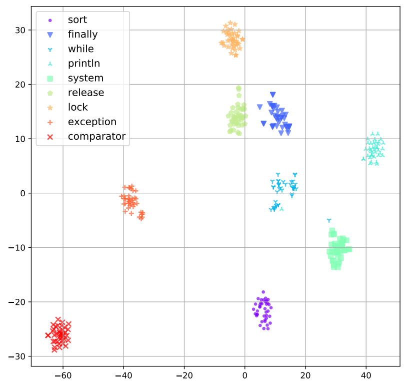
(a) GraphCodeVec



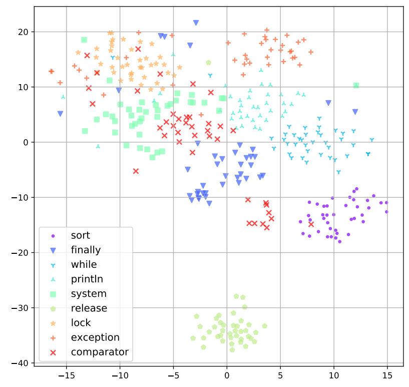
(b) Word2vec



(c) Glove



(d) fastText



(e) code2vec

Figure 4.8: Visualization of the target tokens and their 40 most similar tokens. The horizontal and vertical axes show the two dimensions that are reduced from the original 128 dimensions using the t-SNE.

As shown in Figure 4.8, for the visualization of our embeddings (i.e., GraphCodeVec), we see that several clusters are plotted closely, such as the clusters of "`sort`" and "`comparator`", which is consistent with the fact that they are frequently used together when performing sorting actions. Meanwhile, we do co-occurrence statistics (i.e., count the number of times that every two tokens are used together) of the listed keywords on the training corpus and find that for the token "`comparator`", "`sort`" is the one that occurs together more times than any other listed keywords, which conforms to our interpretation. Besides, for fastText, each target token's cluster is clearly separated from that of other target tokens. However, fastText cannot project similar tokens with a relatively shorter distance. For example, the cluster of "`sort`" is plotted closer to that of "`system`" or "`while`", instead of "`comparator`". For the comparison between GraphCodeVec and Glove, if we focus on the inter-relationships between these clusters, they both project "`sort`" and "`comparator`", "`release`" and "`lock`" as well as "`system`" and "`println`" closer in the space; however, if we focus on the intra-relationships within each cluster, GloVe projects the token "`release`" scattered across different clusters, while on the contrary each cluster of GraphCodeVec is more compact.

This finding confirms that GraphCodeVec can project syntactically similar tokens to the vector space with a relatively short distance. Although the visualization cannot provide us with a direct measurement of the quality of the embeddings, it still helps us gain insights into the characteristics of the resulting embeddings.

Meanwhile, we also try to manually inspect the top-10 nearest neighbors of the given token using cosine similarity. For example, given the target token, "`while`", we retrieve its top-10 most similar tokens and examine whether the token, "`for`" appears in the list or not. The results show that the "`for`" token only appears in the top-10 nearest neighbors of "`while`" when retrieved using the embeddings generated by Word2vec. This observation shows that the trained embeddings may return some results that are different from the prior knowledge of developers or researchers and are hard to interpret [72]. This finding also suggests the necessity of exploring the characteristics of the learned embeddings from different perspectives and shows that Word2vec may perform better than other embeddings when used for retrieving similar tokens.

However, the above findings may not violate the first visualization part of the qualitative analysis. For the visualization using the t-SNE, we retrieve the 40 most similar tokens for the given target token and plot the clustering figures for each token, which is different from retrieving top-k nearest neighbors and checking whether the expected tokens are in the list or not.

Summary

Our evaluation results show that GraphCodeVec achieves better results than all the baselines in five out of six downstream tasks. Besides, GraphCodeVec has the most stable results on all downstream tasks. Future research and practice that rely on code embeddings should be careful with the selection of code embedding techniques for specific downstream tasks, as they may produce diverse results.

RQ2: How does the structural context information of the source code impact the effectiveness of the embeddings generated by GraphCodeVec?

Motivation

In RQ1, our results show that our GCN-based approach GraphCodeVec has the most stable performance and outperforms the baseline approaches. On one hand, prior studies [11, 21, 63, 169, 204] show that incorporating the structural information (e.g., AST structure of source code) of a particular source code of interest may provide promising results in some software engineering (SE) tasks that rely on neural network-based techniques and code is structured by its nature (e.g, class, method, and block) and thus the code embeddings may benefit from the structural representation. On the other hand, there are some studies that treat the source code as plain text and achieve satisfactory results [29, 43, 57]. Therefore, in this research question, we aim to understand how the structural information (i.e., the graph context extracted from the ASTs) affects the performance of GraphCodeVec.

Approach

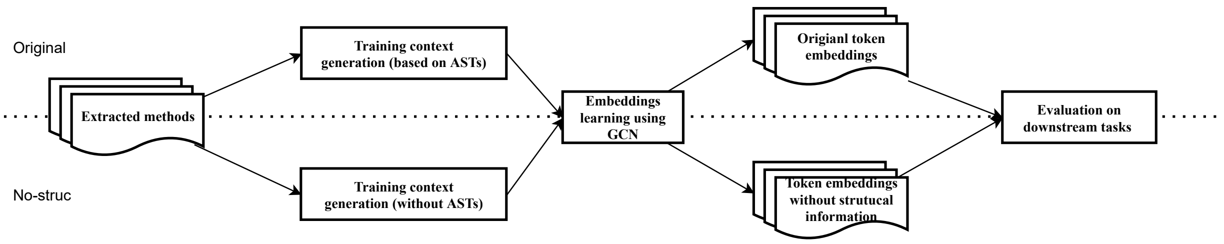

Original
No-struc
Extracted methods
Training context generation (based on ASTs)
Training context generation (without ASTs)
Embeddings learning using GCN
Origianl token embeddings
Token embeddings without strutucal information
Evaluation on downstream tasks

Figure 4.9: The overall design of the approach for RQ2. In this figure, original refers to our GraphCodeVec. No-struc refers to the method that does not utilize the ASTs while keeping other settings the same as GraphCodeVec.

In RQ1, the training context for generating code embeddings by GraphCodeVec is constructed based on the graph representation of source code, which preserves the structural information of source code. Thus, in this section, to analyze the impact of our graph context on generating the embeddings, we design an ablation experiment on these six downstream tasks. In this experiment, as shown in Figure 4.9, the embedding training technique is the same (i.e., GCN), with the only difference in the training context. We generate the training context from the extracted methods without the structural information (unlike our GraphCodeVec, which generates the training context based on ASTs) and then feed it into the GCN model to obtain the code embeddings. We then compare the performance of the embeddings trained with and without the AST structure using the same training technique (i.e., GCN). As the training context is the only changing factor, thus, we state that the performance changes are caused by the different training context. That is if the embeddings with the AST information perform better, then we can conclude that our embeddings can benefit from utilizing the ASTs. More specifically, we treat the source code as plain text and do not consider the AST relationship among the tokens. Below we discuss the details of how we extract the training context and incorporate it in GCN.

First, the source code is transformed into plain text, of which all the tokens are lowercased, and the non-identifiers (including punctuations such as ";" and operators such as "=") are removed. As the training model (i.e., GCN) requires graph-format data as input, to make source code suitable for training, we then adopt a local window to convert the plain text into graphs. Given a target token, all the surrounding tokens located in this window are connected to the target token in the graph by an edge. For example, given the code snippet in Figure 4.2, assuming the target token is "`public`", then "`void`", "`printname`", "`string`", "`someone`", and "`name`" are the neighboring nodes in the generated training context.

```
public void printname string someone name someone system out println name
```

We construct a graph context for the target token "`public`" in the format shown in Figure 4.10.

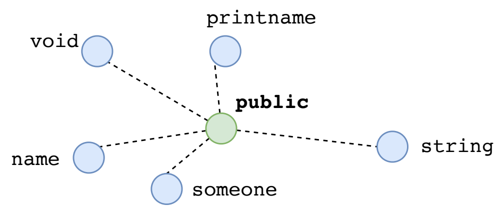


Figure 4.10: An overview of the constructed graph context based on the plain text.

More specifically, the target token "`public`" is the central node of this graph and connects to all the other five nodes, among which there is no edge between each other. We then feed the generated context to the embedding learning phase. Finally, the learned embeddings are evaluated on the six downstream tasks. In our experiment, we set the window size to five on each side surrounding the target token, which is by default used in previous work [19, 82, 123, 124, 149]. Note that a larger window size would be able to capture more broad context, but with the possibility of introducing noise as the context tokens might not be tightly related to the target token. On the contrary, a smaller window size may contain more focused information about the target token but may not be able to capture sufficient context.

Table 4.4: Evaluation results of GraphCodeVec with and without utilizing the graph context extracted from the ASTs.

Dowsnstream Tasks	Code comment generation		Code authorship identification	Code clone detection		Source code classification	Logging statement prediction				
Datasets	GitHub		Google Code Jam	BCB	OJClone	OJ dataset	Airavata	Camel	CloudStack	Directory-Server	Hadoop
Metrics	BLEU	ROUGE	Accuracy	F1		Accuracy	BA				
Original	**20.7**	**36.1**	**80.2**	**93.4**	**93.8**	**93.7**	**95.7**	**81.4**	**86.3**	**89.1**	**75.6**
No-struc	**20.7**	36.0	80.0	**93.4**	93.5	93.6	95.3	80.6	86.0	87.7	74.7

Dowsnstream Tasks	Software defect predicttion											
Datasets	Ant 1.5->1.6	Ant 1.6->1.7	Camel 1.2->1.4	Camel 1.4->1.6	jEdit 3.2->4.0	jEdit 4.0->4.1	Log4j 1.0->1.1	Lucene 2.0->2.2	Lucene 2.2->2.4	POI 1.5->2.5	POI 2.5->3.0	Xalan 2.4->2.5
Metrics	F1											
Original	42.7	**50.5**	**44.6**	46.7	57.0	58.0	**72.5**	**67.0**	65.2	**84.6**	**74.9**	**52.5**
No-struc	**43.9**	48.3	43.5	**48.3**	**57.6**	**58.9**	66.7	59.8	**65.9**	82.7	74.6	51.9

Results

Overall, the graph context extracted from the ASTs can improve the performance of the code embeddings generated by GraphCodeVec; however, GraphCodeVec may not always significantly benefit from the utilization of the graph context. Table 4.4 shows the results of comparing the performance of the original GraphCodeVec and the one that does not use the structural information. In the table, Original refers to our GraphCodeVec. No-struc is the variant of GraphCodeVec, which only utilizes the plain text of the source code instead of the graph context extracted from the ASTs while keeping other settings the same as GraphCodeVec. In total, as Table 4.4 shows, we find that our original GraphCodeVec outperforms No-struc (i.e., the variant of GraphCodeVec that does not consider the graph context extracted from the ASTs) in five out of six downstream tasks[12]. The comparison results demonstrate that even though we train the embeddings using the same model, utilizing the graph context extracted from the ASTs can help improve the performance of the embeddings. For example, on the logging statement prediction task, by training the embeddings using the graph context, GraphCodeVec has an overall balanced accuracy of 85.6% compared to 84.8% without the graph context.

On the other hand, for some tasks, the improvement is limited, and incorporating the graph context extracted from the ASTs may cause performance degradation on some

[12]We compute the average score across all datasets for each task, and GraphCodeVec achieves the highest average score for five out of six downstream tasks.

datasets. For example, on the task of code authorship identification, the overall improvement is only 0.2% and 0.1% for the task if source code classification. In addition, in almost half of the datasets of the software defect prediction task, utilizing the graph context degrades the performance of GraphCodeVec. The result indicates the limited effect of incorporating the graph context in some cases. One possible reason is that some tasks may not be sensitive to the structural information of the source code, thus using a structured code representation may not improve the performance significantly.

We further conducted two more experiments with different window sizes (i.e., two and eight) and the result (shown in Table 4.6) shows that paying more attention to closer neighbors (a smaller window size) would bring more benefits. As when we reduce the window size to two, we observe statistically significant improvement in five out of seven (i.e., seven cases have significant performance changes among which five cases have improvement) cases (71.4%), and when we increase the window size to eight, we observe statistically significant improvement in four out of eight cases (50%).

Summary

Although overall, the structural information extracted from the ASTs can benefit GraphCodeVec in producing code embeddings for the downstream SE tasks, there may be cases where the structural information may not provide additional benefit.

RQ3: How does the GCN model impact the effectiveness of the embeddings generated by GraphCodeVec?

Motivation

Prior work [84] proposes a novel word embedding approach for NLP tasks that adopts a shallow, two-layer neural network instead of Graph Convolutional Networks to incorporate the syntactic information between words and achieves promising results. Their results raise our concern about whether a simple two-layer neural network is powerful enough to model the syntactic information within the corpus. Therefore, in this research question, we want to study how the GCN model affects the performance of GraphCodeVec for generating the code embeddings for the downstream tasks.

Approach

In RQ2, we analyze the impact of structural context information on the effectiveness of the embeddings generated by GCN. We train two different code embeddings using different training contexts (i.e., with and without AST information) but the same training

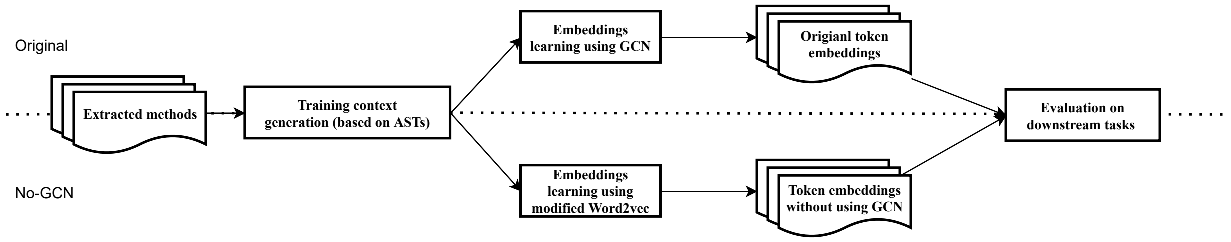

Original
No-GCN
Extracted methods
Training context generation (based on ASTs)
Embeddings learning using GCN
Embeddings learning using modified Word2vec
Origianl token embeddings
Token embeddings without using GCN
Evaluation on downstream tasks

Figure 4.11: The overall design of the approach for RQ3. In this figure, original refers to our GraphCodeVec. No-GCN refers to the method that does not utilize the GCN for embedding learning.

embedding technique (i.e., GCN). In this section, to analyze the impact of the GCN model on generating the embeddings, similar to RQ2, we design an ablation experiment on these six downstream tasks. In this experiment, as shown in Figure 4.11, we adopt two different training techniques with the same training context, which both consider the structural information for code embedding learning. Specifically, we implement another method, namely, No-GCN [84] for comparison. No-GCN uses a similar approach to extract the graph context from the ASTs but adopts a shallow, two-layer neural network to train embeddings. No-GCN was originally proposed by Li et al. [84] for learning word embeddings by incorporating the dependency information between words in a sentence. Li et al. [84] modify the original Word2vec model and integrate the syntactic dependency information between words into the embeddings. In this work, we customize No-GCN by replacing the syntactic dependency with the AST paths extracted from the source code.

No-GCN uses a similar way of extracting the training context from the source code. It first transforms the source code into ASTs, then traverses the trees to collect triples, where the first and last elements are the leaf nodes of an AST and the second element is the AST path connecting the other two elements. For example, given a target token, "`public`" in Figure 4.3, it starts from "`public`" and keeps traversing the tree until it reaches another leaf node (e.g., "`void`"), and the traversing path is recorded. By doing this, it can collect a set of triples that can be used for training the code embeddings. Similar to our work, the number of triples is also limited by the length of an AST path.

Different from the GCN used in this chapter, No-GCN modifies the original Word2vec model to include the AST paths instead of only considering the tokens (more details can be found in the work [84]).

Results

The comparison results with No-GCN show the advantage of using GCN for modeling the graph context. Our experimental results for comparing GraphCodeVec with No-GCN on the six SE tasks are presented in Table 4.5. As Table 4.5 shows, we find

Table 4.5: Evaluation results of utilizing different models to train the code embeddings from the graph context.

Dowsnstream Tasks	Code comment generation		Code authorship identification	Code clone detection		Source code classification	Logging statement prediction				
Datasets	GitHub		Google Code Jam	BCB	OJClone	OJ dataset	Airavata	Camel	CloudStack	Directory-Server	Hadoop
Metrics	BLEU	ROUGE	Accuracy	F1		Accuracy	BA				
Original	20.7	36.1	**80.2**	**93.4**	**93.8**	**93.7**	95.7	**81.4**	**86.3**	**89.1**	**75.6**
No-GCN	**21.4**	**36.7**	79.8	**93.4**	91.0	90.1	**95.9**	80.4	**86.3**	87.4	74.7

Dowsnstream Tasks	Software defect predicttion											
Datasets	Ant 1.5->1.6	Ant 1.6->1.7	Camel 1.2->1.4	Camel 1.4->1.6	jEdit 3.2->4.0	jEdit 4.0->4.1	Log4j 1.0->1.1	Lucene 2.0->2.2	Lucene 2.2->2.4	POI 1.5->2.5	POI 2.5->3.0	Xalan 2.4->2.5
Metrics	F1											
Original	42.7	**50.5**	**44.6**	46.7	57.0	**58.0**	**72.5**	**67.0**	**65.2**	**84.6**	**74.9**	**52.5**
No-GCN	**43.3**	49.8	44.1	**49.6**	**58.6**	57.3	69.1	63.0	62.0	77.2	71.8	48.4

that overall our GraphCodeVec has the best results in five out of six downstream tasks. For example, on the source code classification task, No-GCN achieves a test accuracy of 90.1%, while GraphCodeVec reaches 93.7%. The comparison results show that GCN is more suitable for representing the source code as graphs and capturing the syntactic structure of the source code when generating code embeddings. However, similar to the results in RQ1, GraphCodeVec also does not reach the best results on the task of code comment generation. This may be due to the fact that GraphCodeVec is good at capturing the properties of source code, while the task of code comment is for generating the natural language texts, and thus our approach cannot perform well. Besides, we find that the improvement for the task of code authorship identification is limited, with only 0.4% absolute increase. This observation further confirms our findings in RQ2 that some tasks may be not sensitive to the structural information of the source code.

Compared to the impact of the graph context in RQ2, we find that the GCN model has a more stable influence on the performance of GraphCodeVec. On the one hand, in RQ2, replacing the graph context with plain text causes a relatively smaller performance decrease on the downstream SE tasks compared to changing the training model in RQ3. For example, there is a 0.1% degradation of the test accuracy on the source code classification task after changing the training context in RQ2, compared to 3.6% degradation after changing the training model in RQ3. On the other hand, in RQ2, we do not observe improvement by using the graph context on almost half of the datasets of the software defect prediction task; while in RQ3, we observe improvement by using the GCN model on nine datasets. Our results suggest the promising research direction of using graph-based deep learning methods for SE tasks.

Summary

Instead of using a vanilla neural network, the use of Graph Convolutional Networks can robustly benefit the performance of GraphCodeVec for training code embeddings for the downstream SE tasks.

4.5 Discussion

In Section 4.4, we have conducted several experiments and shown that our task-agnostic GraphCodeVec can effectively be applied to different downstream tasks. In this section, we would like to have a discussion about the impact of different model parameters and the results of repeating our experiments with different data sampling using a 10-fold cross-validation.

Impact of modeling parameters. GraphCodeVec contains two stages (i.e., training context generation and embedding learning) for learning the code embeddings where some thresholds and model hyperparameters are involved for generating the training corpus as well as defining the GCN structure. In this work, we either simply follow previous work or use the default settings of the model and do not try to fine-tune the parameters for fitting into different tasks. In this part, to examine whether our generated code embeddings can be further improved and assess the impact of the hyperparameters on the quality of the generated code embeddings, we conduct more than 20 new experiments with different parameter settings and the corresponding results are listed in Table 4.6. We also conduct a Wilcoxon signed-rank statistical test to check whether there is a significant performance change between the performance of the model using the newly configured parameters and that of the model using the default parameters results are significant, for significant changes, we further conduct Cliff's delta statistic to check the effect sizes. The significant changes are marked in bold as shown in Table 4.6.

The performance of GraphCodeVec on some tasks can be further improved by fine-tuning the model parameters. For example, if we set the dimensionality of the embeddings to 300, we observe a performance increase of the F1 score (i.e., from 93.8 to 95.6) on the OJClone dataset for the task of code clone detection. Meanwhile, changing the parameters can also decrease the performance of GraphCodeVec. In our experiment, using a smaller dimensionality (i.e., 50) of the embeddings leads to performance degradation in almost all the tasks and datasets. The reason may be that a small dimensionality of the embeddings cannot preserve the properties of the tokens of high dimensional spaces, leading to the degradation of the quality of learned embeddings. On the contrary, using a relatively

Table 4.6: Evaluation results of code embeddings generated by GraphCodeVec with different thresholds and model hyperparameters.

Stage	Name	Value	Downstream Tasks																						
			Code comment generation		Code authorship identification	Code clone detection		Source code classification	Logging statement prediction					Software defect predicttion											
			GitHub		Google Code Jam	BCB	OJClone	OJ dataset	Airavata	Camel	CloudStack	Directory-Server	Hadoop	Ant 1.5->1.6	Ant 1.6->1.7	Camel 1.2->1.4	Camel 1.4->1.6	jEdit 3.2->4.0	jEdit 4.0->4.1	Log4j 1.0->1.1	Lucene 2.0->2.2	Lucene 2.2->2.4	POI 1.5->2.5	POI 2.5->3.0	Xalan 2.4->2.5
			BLEU	ROUGE	Accuracy	F1		Accuracy	BA					F1											
		Default	20.7	36.1	80.2	93.4	93.8	93.7	95.7	81.4	86.3	89.1	75.6	42.7	50.5	44.6	46.7	57	58	72.5	67	65.2	84.6	74.9	52.5
Training context generation	Edge length	6	20.8 (+0.5%)	36.2 (+0.3%)	80 (-0.2%)	93.4 (+0.0%)	94.1 (+0.3%)	93.3 (-0.4%)	95.8 (+0.1%)	81.8 (+0.5%)	87.3 (+1.2%)	88.1 (-1.1%)	74.4 (-1.6%)	**48.9L (+14.5%)**	**52.2L (+3.4%)**	**43.3L (-2.9%)**	47.6 (+1.9%)	**59.4L (+4.2%)**	**55.2L (-4.8%)**	**67.6L (-6.8%)**	**64.3L (-4.0%)**	65.7 (+0.8%)	**82.5L (-2.5%)**	**72.9L (-2.7%)**	**49.7L (-5.3%)**
		10	20.8 (+0.5%)	36.2 (+0.3%)	80 (-0.2%)	93.4 (+0.0%)	93.1 (-0.7%)	93.4 (-0.3%)	94 (-1.8%)	79.2 (-2.7%)	86.4 (+0.1%)	**87.7S (-1.6%)**	74.9 (-0.9%)	**48.2L (+12.9%)**	50.4 (-0.2%)	**41.0L (-8.1%)**	46.9 (+0.4%)	56.2 (-1.4%)	**55.4L (-4.5%)**	**66.5L (-8.3%)**	**60.8L (-9.3%)**	**62.4L (-4.3%)**	**81.9L (-3.2%)**	**72.2L (-3.6%)**	**50.6L (-3.6%)**
	Unique node	50	20.7 (+0.0%)	36.2 (+0.3%)	80.1 (-0.1%)	93.5 (+0.1%)	94.6 (+0.9%)	93.9 (+0.2%)	95.6 (-0.1%)	81.1 (-0.4%)	86.1 (-0.2%)	**86.7L (-2.7%)**	74.7 (-1.2%)	42.5 (-0.5%)	51.1 (+1.2%)	44.4 (-0.4%)	46.5 (-0.4%)	57 (+0.0%)	58.3 (+0.5%)	70.9 (-2.2%)	66.7 (-0.4%)	65.2 (+0.0%)	83.9 (-0.8%)	74.5 (-0.5%)	53.1 (+1.1%)
		80	20.7 (+0.0%)	36.1 (+0.0%)	80 (-0.2%)	93.4 (+0.0%)	94.5 (+0.7%)	94 (+0.3%)	96.4 (+0.7%)	80.3 (-1.4%)	86.8 (+0.6%)	87.4 (-1.9%)	**73.4M (-2.9%)**	43.6 (+2.1%)	50.7 (+0.4%)	44.3 (-0.7%)	46.5 (-0.4%)	56.9 (-0.2%)	58.1 (+0.2%)	72.3 (-0.3%)	66.7 (-0.4%)	64.1 (-1.7%)	84.5 (-0.1%)	75.1 (+0.3%)	52.9 (+0.8%)
	Edge	400	20.7 (+0.0%)	36.2 (+0.3%)	80 (-0.2%)	**93.3N (-0.1%)**	93.8 (+0.0%)	93.4 (-0.3%)	95.3 (-0.4%)	80.8 (-0.7%)	86.9 (+0.7%)	**86.7M (-2.7%)**	75 (-0.8%)	43.9 (+2.8%)	50.3 (-0.4%)	44 (-1.3%)	46.5 (-0.4%)	57.2 (+0.4%)	58.4 (+0.7%)	72.4 (-0.1%)	**66.5L (-0.7%)**	64.7 (-0.8%)	84.3 (-0.4%)	74.7 (-0.3%)	52.6 (+0.2%)
		600	**20.8S (+0.5%)**	36.2 (+0.3%)	80 (-0.2%)	93.4 (+0.0%)	93.7 (-0.1%)	93.6 (-0.1%)	95.3 (-0.4%)	79.5 (-2.3%)	87 (+0.8%)	87.8 (-1.5%)	75.4 (-0.3%)	43.4 (+1.6%)	50.8 (+0.6%)	44 (-1.3%)	**45.9L (-1.7%)**	57.1 (+0.2%)	58.3 (+0.5%)	72.1 (-0.6%)	66.5 (-0.7%)	65 (-0.3%)	84.2 (-0.5%)	75.2 (+0.4%)	53.1 (+1.1%)
	Window (cf., RQ2)	2	20.7 (+0.0%)	36.1 (+0.3%)	80 (+0.0%)	93.5 (+0.1%)	92.2 (-1.4%)	93 (-0.6%)	95.3 (+0.0%)	79.8 (-1.0%)	87 (+1.2%)	87.5 (-0.2%)	75.1 (+0.5%)	43.2 (-1.6%)	**54.6L (+13.0%)**	**42.2L (-3.0%)**	48.8 (+1.0%)	**61.1L (+6.1%)**	59 (+0.2%)	**62.5L (-6.3%)**	**61.3L (+2.5%)**	**63.0L (-4.4%)**	**83.9L (+1.5%)**	74.3 (-0.4%)	52.3 (+0.8%)
		5	20.7	36	80	93.4	93.5	93.6	95.3	80.6	86	87.7	74.7	43.9	48.3	43.5	48.3	57.6	58.9	66.7	59.8	65.9	82.7	74.6	51.9
		8	20.6 (-0.5%)	36 (+0.0%)	80.3 (+0.4%)	93.5 (+0.1%)	**92.1L (-1.5%)**	93.9 (+0.3%)	95.1 (-0.2%)	79.9 (-0.9%)	**87.5L (+1.7%)**	87 (-0.8%)	74.2 (-0.7%)	42.7 (-2.7%)	**55.1L (+14.1%)**	**42.3L (-2.8%)**	48.5 (+0.4%)	**61.2L (+6.3%)**	58.6 (-0.5%)	**62.4L (-6.4%)**	**61.2L (+2.3%)**	**63.4L (-3.8%)**	82.9 (+0.2%)	74.1 (-0.7%)	52.7 (+1.5%)
Embedding learning	Layer	3	**20.5M (-1.0%)**	**35.8L (-0.8%)**	80.2 (+0.0%)	93.4 (+0.0%)	93.5 (-0.3%)	94 (+0.3%)	94.9 (-0.8%)	82 (+0.7%)	86.7 (+0.5%)	88.2 (-1.0%)	75.6 (+0.0%)	**46.2L (+8.2%)**	**47.5L (-5.9%)**	**40.1L (-10.1%)**	**45.1L (-3.4%)**	57.1 (+0.2%)	**56.9L (-1.9%)**	**64.4L (-11.2%)**	**63.6L (-5.1%)**	**67.8L (+4.0%)**	**81.3L (-3.9%)**	**73.2L (-2.3%)**	**53.7L (+2.3%)**
		5	**20.0L (-3.4%)**	**35.4L (-1.9%)**	80.1 (-0.1%)	93.4 (+0.0%)	**90.4L (-3.6%)**	93.2 (-0.5%)	94.8 (-0.9%)	82.1 (+0.9%)	87.2 (+1.0%)	**86.7L (-2.7%)**	74.5 (-1.5%)	**41.1L (-3.7%)**	**41.5L (-17.8%)**	**34.5L (-22.6%)**	**44.5L (-4.7%)**	**47.7L (-16.3%)**	**47.2L (-18.6%)**	**45.8L (-36.8%)**	0	**57.3L (-12.1%)**	**76.8L (-9.2%)**	**72.1L (-3.7%)**	**55.3L (+5.3%)**
	Dim.	50	20.6 (-0.5%)	36 (-0.3%)	**75.0L (-6.5%)**	93.4 (+0.0%)	**91.0L (-3.0%)**	93.3 (-0.4%)	94.6 (-1.1%)	80 (-1.7%)	85.1 (-1.4%)	87.9 (-1.3%)	75.1 (-0.7%)	**48.7L (+14.1%)**	**48.1L (-4.8%)**	**39.6L (-11.2%)**	46.6 (-0.2%)	**59.2L (+3.9%)**	57.3 (-1.2%)	72 (-0.7%)	**61.8L (-7.8%)**	**61.4L (-5.8%)**	**82.7L (-2.2%)**	74 (-1.2%)	52.8 (+0.6%)
		300	20.7 (+0.0%)	36.1 (+0.0%)	81.7 (+1.9%)	93.4 (+0.0%)	**95.6L (+1.9%)**	92.5 (-1.3%)	95.6 (-0.1%)	80.3 (-1.4%)	87.4 (+1.3%)	**87.5M (-1.8%)**	75.3 (-0.4%)	42.3 (-0.9%)	**52.4L (+3.8%)**	44.1 (-1.1%)	**48.2L (+3.2%)**	**60.8L (+6.7%)**	57.8 (-0.3%)	**68.4L (-5.7%)**	**62.9L (-6.1%)**	64.9 (-0.5%)	**83.8L (-0.9%)**	**72.1L (-3.7%)**	**50.6L (-3.6%)**
	Neg.	2	20.7 (+0.0%)	36.1 (+0.0%)	79.6 (-0.7%)	93.4 (+0.0%)	93.4 (-0.4%)	93.5 (-0.2%)	95.9 (+0.2%)	80.7 (-0.9%)	86.3 (+0.0%)	88.4 (-0.8%)	74.6 (-1.3%)	**50.8L (+19.0%)**	**48.7L (-3.6%)**	**40.1L (-10.1%)**	**48.3L (+3.4%)**	**59.3L (+4.0%)**	58.2 (+0.3%)	**65.4L (-9.8%)**	**60.9L (-9.1%)**	**63.0L (-3.4%)**	**83.0L (-1.9%)**	**73.3L (-2.1%)**	**49.3L (-6.1%)**
		10	**20.8S (+0.5%)**	36.2 (+0.3%)	80.7 (+0.6%)	93.4 (+0.0%)	93.6 (-0.2%)	93.1 (-0.6%)	95.6 (-0.1%)	80.2 (-1.5%)	86.4 (+0.1%)	88.3 (-0.9%)	76 (+0.5%)	**47.0L (+10.1%)**	**49.3L (-2.4%)**	44.4 (-0.4%)	**47.8L (+2.4%)**	**60.2L (+5.6%)**	**59.4L (+2.4%)**	**67.9L (-6.3%)**	**63.2L (-5.7%)**	**62.2L (-4.6%)**	**81.3L (-3.9%)**	**72.8L (-2.8%)**	**49.4L (-5.9%)**
	Batch size	32	**20.9S (+1.0%)**	**36.3M (+0.6%)**	80.3 (+0.1%)	93.4 (+0.0%)	**94.8S (+1.1%)**	**91.8L (-2.0%)**	95.5 (-0.2%)	80.9 (-0.6%)	87 (+0.8%)	87.7 (-1.6%)	73.7 (-2.5%)	**44.4L (+4.0%)**	**51.4L (+1.8%)**	**42.3L (-5.2%)**	**49.4L (+5.8%)**	**60.0L (+5.3%)**	**56.6L (-2.4%)**	**66.0L (-9.0%)**	67.1 (+0.1%)	**66.9L (+2.6%)**	**83.3L (-1.5%)**	**73.6L (-1.7%)**	**49.9L (-5.0%)**
		128	**20.6S (-0.5%)**	36 (-0.3%)	80 (-0.2%)	93.3 (-0.1%)	93.8 (+0.0%)	93.8 (+0.1%)	95.7 (+0.0%)	80.3 (-1.4%)	87.2 (+1.0%)	87.1 (-2.2%)	74.9 (-0.9%)	**44.5L (+4.2%)**	**52.0L (+3.0%)**	**40.3L (-9.6%)**	**44.6L (-4.5%)**	57.1 (+0.2%)	58.4 (+0.7%)	71.2 (-1.8%)	**60.9L (-9.1%)**	**63.2L (-3.1%)**	**83.5L (-1.3%)**	74.8 (-0.1%)	**53.5L (+1.9%)**
	Dropout rate	0.2	20.8 (+0.5%)	36.1 (+0.0%)	80.2 (+0.0%)	93.4 (+0.0%)	93.8 (+0.0%)	92.7 (-1.1%)	96.6 (+0.9%)	81.2 (-0.2%)	87.2 (+1.0%)	87.3 (-2.0%)	74.2 (-1.9%)	41.8 (-2.1%)	**46.9L (-7.1%)**	44 (-1.3%)	47.6 (+1.9%)	**59.8L (+4.9%)**	59.1 (+1.9%)	**66.7L (-8.0%)**	**62.6L (-6.6%)**	65.2 (+0.0%)	**80.0L (-5.4%)**	**73.3L (-2.1%)**	**53.9L (+2.7%)**
		0.5	**20.8S (+0.5%)**	**36.2S (+0.3%)**	80.1 (-0.1%)	93.4 (+0.0%)	93.7 (-0.1%)	93.8 (+0.1%)	94.4 (-1.4%)	80 (-1.7%)	86.7 (+0.5%)	87.4 (-1.9%)	75.3 (-0.4%)	**45.6L (+6.8%)**	51.1 (+1.2%)	**42.6L (-4.5%)**	**45.1L (-3.4%)**	**58.8L (+3.2%)**	57.3 (-1.2%)	**66.6L (-8.1%)**	**61.6L (-8.1%)**	64.7 (-0.8%)	**81.2L (-4.0%)**	**72.6L (-3.1%)**	**51.4L (-2.1%)**
	Epoch	5	**21.1L (+1.9%)**	**36.4L (+0.8%)**	80.3 (+0.1%)	93.4 (+0.0%)	94.3 (+0.5%)	**90.8L (-3.1%)**	95.3 (-0.4%)	81.2 (-0.2%)	86.9 (+0.7%)	87.7 (-1.6%)	74.5 (-1.5%)	42 (-1.6%)	**52.4L (+3.8%)**	**46.1L (+3.4%)**	**48.0L (+2.8%)**	**58.8L (+3.2%)**	**62.3L (+7.4%)**	**68.9L (-5.0%)**	**62.9L (-6.1%)**	**62.7L (-3.8%)**	83.7 (-1.1%)	74.5 (-0.5%)	**50.6L (-3.6%)**
		10	**21.2L (+2.4%)**	**36.6L (+1.4%)**	80.4 (+0.2%)	**93.3M (-0.1%)**	94.2 (+0.4%)	**90.9L (-3.0%)**	95.2 (-0.5%)	79.8 (-2.0%)	86.8 (+0.6%)	88.4 (-0.8%)	75.8 (+0.3%)	42 (-1.6%)	**51.4M (+1.8%)**	**47.2L (+5.8%)**	47 (+0.6%)	57.9 (+1.6%)	**62.6L (+7.9%)**	**69.9L (-3.6%)**	**63.0L (-6.0%)**	**62.4L (-4.3%)**	**81.9L (-3.2%)**	**72.3L (-3.5%)**	**47.6L (-9.3%)**

Note: The results that are significantly different from that of the default settings are highlighted in bold. The numbers in the brackets indicate the relative change to the default settings of GraphCodeVec. The letters S, M, L, and N represent small, medium, large and negligible effect sizes, respectively.

larger dimensionality can preserve more information and improve the quality of the generated embeddings. Another possible reason may be underfitting of the models used in downstream tasks, as a smaller input dimension means a simpler model and fewer weights to be learned during model training, and thus the model cannot capture the relationship between the input and output variables accurately, generating a high error rate on the testing data. To examine the impact of the number of training epochs, we provide another two experiments with more training epochs (i.e., five and ten epochs). During the embedding learning phase, the training loss reduces from 1.79 at the beginning of the first training epoch to 0.73 at the end of the first epoch, which further drops to 0.47 and 0.48 at the end of the fifth and the tenth epochs, respectively. Further, we also evaluated the quality of the newly generated embeddings on the downstream tasks. Overall, as shown in Table 4.6, when the training epoch increases to five, we observe seven improvements in the evaluation of the downstream tasks. On the other hand, we also observe that there are degradations on some (i.e. five) of the datasets from source code classification and software defect prediction tasks. The results indicate that the training epochs have different effects (either positive or negative depending on the downstream tasks) on the quality of the generated embeddings, and developers can fine-tune these epochs, especially for their tasks. In our experiments, as we stated in Section 4.3.4, we avoid fine-tuning these settings only for our method aiming for better performance.

A model with more layers (i.e., a deeper model) may not guarantee better performance, especially for GCN models. In our supplement experiments, we increase the depth (i.e., layers) of GraphCodeVec from one to three and five, we find a continuous performance degradation for almost all the tasks, the model even returns an F1 score of zero for the task of software defect prediction on the Lucene project. This finding is consistent with previous works [26, 85, 90, 151, 209], that is with an increased depth (i.e., number of layers), GCN tends to easily overfit the training data and suffer a continuous performance degradation. Except for reducing the number of layers (we set the layer to one to avoid overfitting and performance degradation, cf., Section 4.3.2) of the model, researchers [161] also propose to use dropout to prevent overfitting. Dropout is a regularization technique that randomly drops out the units along with their connections of neural networks. To examine the impact of using dropout on the quality of our generated code embeddings, we have experimented with two different dropout rates (i.e., 0.2 and 0.5). Overall, as Table 4.6 shows, we do not have obvious performance improvements when using different dropout ratios. This can be explained by the fact that our model (e.g., one layer, 128 input dimension, and trained for only one epoch) does not suffer the overfitting problem, and thus using dropout cannot further improve the performance of our embeddings on downstream tasks. However, the results in our experiments do not mean that dropout is useless, instead, it indicates that our model structure may not suffer from the overfitting issue. Moreover, previous work [22, 49, 52] and experiments[13,14] also show that using dropout may not always improve the performance of neural networks, which further confirms our findings. For example, Garbin et al. [49] and Cai et al. [22] observe that adding dropout may reduce the performance of the model. Besides, in the original work of dropout [161], the authors also explored the effect of changing data set size when dropout is used, and the results show that when the size of data sets is very small (e.g., 100, 500 samples) or very large (e.g., 50K samples), dropout may not give any improvements. These results suggest that we should be careful when applying the dropout to the neural networks. Meanwhile, previous work [52, 161] provides suggestions on how and when to use dropout to avoid overfitting. For example, it is expected that dropping the neurons in the model would reduce the effective capacity of a model, thus Srivastava et al. [161] suggest increasing the size of the model when using dropout and they suggest setting the number of units to n/p, where p is the dropout rate and n is the number of optimal units for a model without dropout. Besides, Goodfellow et al. [52] also suggests that when there is a large amount of training data, the benefit of using dropout may be outweighed by the computational cost of using dropout and larger models. Thus, considering our simple model architecture and the large size of the training dataset (i.e., over 60K samples), it is reasonable that using dropout does not significantly improve the quality of our generated embeddings. To better illustrate the ability of dropout in preventing model overfitting, future work can try to add more layers with more training epochs.

Traditional machine learning model (e.g., logistic regression used in the task of software defect prediction) is more sensitive to the changes of code embeddings. As shown in Table 4.6, almost any changes in the parameters of GraphCodeVec could lead to significant changes in the performance (either improvement or deterioration) of the software defect prediction task. This may be explained by the fact that the code embeddings are directly used as features for the traditional machine learning models, thus any changes in embeddings could be immediately propagated to the final output of these models. However, for deep learning-based models, the embeddings are only used to initialize the first embedding layer of which the value would be later adjusted to better fit the training data, as a result, the impact of utilizing different embeddings may be diminished or even erased during the model training and weights updating.

The thresholds used during the training context generation stage have a relatively more minor impact on the code embeddings generated by GraphCodeVec than that of the parameters involved during the embedding learning stage. For example, as we lower the thresholds of unique nodes (i.e., 50 and 80), there is only one significant performance

[13]https://github.com/mvshashank08/article-dropout

[14]https://github.com/harrisonjansma/Research-Computer-Vision/blob/master/08-12-18%20Batch%20Norm%20vs%20Dropout/08-12-18%20Batch%20Norm%20vs%20Dropout.ipynb

Table 4.7: Evaluation results of fastText with different preprocessing strategies.

Dowsnstream Tasks	Code comment generation		Code authorship identification	Code clone detection		Source code classification	Logging statement prediction				
Datasets	GitHub		Google Code Jam	BCB	OJClone	OJ dataset	Airavata	Camel	CloudStack	Directory-Server	Hadoop
Metrics	BLEU	ROUGE	Accuracy	F1		Accuracy	BA				
Original	**19.9**	**36.0**	**76.6**	**93.4**	**84.6**	**76.7**	95.1	79.8	86.7	88.6	**74.4**
Lowercase	19.3	35.3	70.5	**93.4**	75.2	60.2	**96.2**	**80.2**	**86.8**	88.5	74.1

Dowsnstream Tasks	Software defect prediction											
Datasets	Ant 1.5->1.6	Ant 1.6->1.7	Camel 1.2->1.4	Camel 1.4->1.6	jEdit 3.2->4.0	jEdit 4.0->4.1	Log4j 1.0->1.1	Lucene 2.0->2.2	Lucene 2.2->2.4	POI 1.5->2.5	POI 2.5->3.0	Xalan 2.4->2.5
Metrics	F1											
Original	**36.0**	**44.2**	41.8	45.8	**53.6**	60.7	**63.1**	63.2	**65.3**	65.1	72.2	42.4
Lowercase	29.3	41.7	**44.5**	**46.0**	53.3	**61.5**	58.7	**65.0**	62.3	**69.9**	**72.5**	**47.1**

Note: Original and Lowercase are two different preprocessing strategies, where Original contains three steps (1) remove non-identifiers, 2) filter out the rare tokens, and 3) lowercase all tokens; while Lowercase means that we only perform a lowercase preprocessing on the tokens (i.e., without removal of non-identifiers and low-frequency tokens). We compute the average score across all datasets for task-level comparison.

change among all 23 tasks or datasets. This finding shows that GraphCodeVec is different from fastText, which is sensitive to the preprocessing of the corpus [69]. To complement the experiments, we have done another experiment for fastText where we only perform a lowercase preprocessing on the tokens (i.e., without the removal of non-identifiers and low-frequency tokens). The results are shown in Table 4.7. In our experiment, we find that among all the six tasks, the performances of four tasks (i.e., code comment generation, code authorship identification, code clone detection, source code classification) have relatively large changes. The results confirm that fastText is sensitive to the preprocessing of training context. For example, on the task of source code classification, there is a 16.5% absolute decrease in fastText with different preprocessing strategies. Besides, by checking the significant performance changes caused by different GraphCodeVec settings, we find that changing parameters involved during the embedding learning stage has a higher possibility of causing significant performance changes in the different tasks and datasets. More specifically, modifying the threshold of unique nodes and the edge only causes one or two significant performance changes, even on the software defect prediction task, which is more sensitive to the change of code embeddings.

Impact of different data sampling. Different separations between training and test sets have a non-negligible effect on the performance of the models. Figure 4.12 shows the results of different embedding techniques on all datasets using 10-fold cross-validation. We can observe the apparent variances in almost all the tasks or datasets. For example, on the task of code authorship identification, all the results of the evaluated code embeddings have large variances, and the differences between the lowest and highest scores even exceed

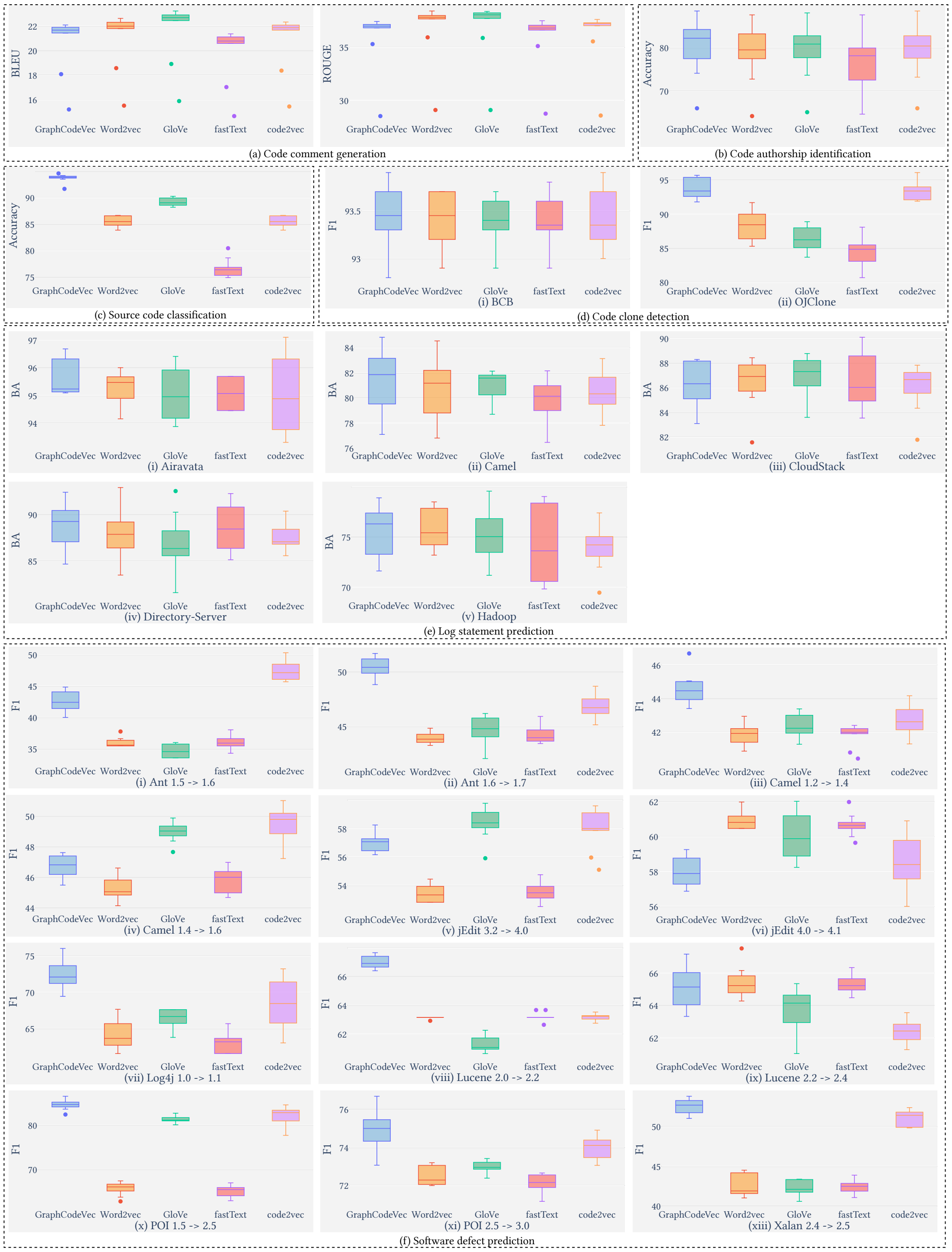


Figure 4.12: The results of different embedding techniques on all datasets using 10-fold cross-validation.

10%. Although the variances on the tasks of code comment generation and source code classification seem to be small, both have obvious outliers, and the range of the Y-axis is larger. This finding further indicates the necessity of running multiple times with different separations between the training and test datasets to mitigate the effects on the data separation. Otherwise, the reported conclusions may be misleading, as the rankings of the performance of different embedding techniques may differ.

Finally, we want to highlight that, on the one hand, fine-tuning parameters of GraphCodeVec for the different downstream tasks can usually result in improved performance. On the other hand, different parameters can have diverse impacts on the final performance, and if we do not know which settings to choose, starting from the suggestions from previous work is always not a bad choice.

4.6 Threats to Validity

This section discusses the threats to the validity of our work.

External validity. One major threat of using GCN for training embeddings is the computational costs. In our work, the embeddings are trained in an NVIDIA GTX 1080Ti GPU, and it takes around 18 minutes to finish the training process, which is acceptable. In fact, the major computational costs are caused by the downstream tasks. For example, it takes around 10 hours to finish the evaluation of the comment generation task. Considering the large amount of time and computing resources needed for executing the downstream tasks, our quantitative evaluation is conducted on six SE tasks. However, we train our embeddings in a task-agnostic manner using an independent dataset from the datasets used in the downstream tasks. Although our study only focuses on six tasks, the scale of our study is comparable to prior research on embedding evaluation [72]. Meanwhile, there exist other tasks that adopt the pre-trained embeddings, and we cannot confirm that our embeddings might be generalizable to all the tasks. For example, for the tasks that rely on both natural language texts and source code, such as traceability link recovery [136, 137] and user review classification [140], we think that our embeddings may not perform very well on these two tasks as our code embeddings are only trained on source code and cannot capture the properties of natural language texts, which is confirmed by the task of code comment generation. However, we believe it would be a very promising research direction to jointly learn the code and text embeddings, and in that way, the embeddings can be applied to such tasks which involve both texts and source code. Another threat is that some of our models used in downstream tasks may not give state-of-the-art results. For example, we use logistic regression in the task of software defect prediction, which is simple and a bit out of date, especially in the era of deep learning. However, our goal is to show the performance changes of different code embeddings. Although this model is simple, it is able to reflect the representation ability of different code embeddings. Nevertheless, we admit that our choices of the models in the studied downstream tasks pose a threat to the generalizability of our findings. Thus, using the downstream task of software defect prediction as an example, we experimented with other models, including Random Forest (RF), Naive Bayesian (NB), and Support Vector Machines (SVM). We observe that our general findings remain the same, and our proposed embedding approach achieves the best results for all the models except NB. We speculate that it may be because NB is not best suited for the task as it holds a strong assumption on the independence of the features which are difficult to satisfy in the resulting embeddings. In fact, the performance of NB is among the worst of all the considered models. On the other hand, we encourage future work to validate our findings on more downstream tasks and models. Moreover, there is a lack of qualitative tasks for quality evaluation, and all the downstream tasks are external tasks, which means we cannot do the evaluation directly. To minimize the threat and explore the internal characteristics of embeddings, we also provide a qualitative evaluation. While the qualitative evaluation may include subjective bias in terms of selection of example tokens and interpretation of their projection in the semantic space, that may be introduced by the different backgrounds of researchers. However, we have already provided the trained embeddings, and readers can explore the properties among the tokens of their own interests. Future studies can apply GraphCodeVec to other tasks, such as method name prediction, and develop some qualitative evaluation datasets, such as token similarity or token analogy test sets. For the comparison of the results, we report the final score of each evaluation metric. However, small variations (e.g., when one instance is classified in a different direction) may change the results. Our goal is to understand the performance changes between different code embeddings among different tasks. Although a small number of misclassifications may cause significant changes in the final scores, especially for small datasets, even in such cases, the improvement or decrease of the performance can still reflect the effect of the different embeddings. Besides, we do a 10-fold cross-validation to reduce the impact of such cases.

Internal validity. As described in Section 4.2.1, we attempt to represent the source code into graphs, where the nodes are tokens in the source code, and edges are AST paths. There could exist other strategies for representing the source code as a graph. Besides, we rely on the surface forms of the tokens to build the connected graph within a method, as a result, changing the name of a variable would lead to the change of the constructed graph. In particular, using meaningless identifiers (e.g., “v”) may negatively impact the quality of the resulting embeddings and their effectiveness in the downstream tasks. However, we have applied our approach to a variety of real-world software projects. The results demonstrate the effectiveness of our approach when applied to ordinary code written by different developers. Meanwhile, using the same identifiers in different surrounding code contexts would also impact the performance of our approach. For example, the keyword "`public`", can be used as modifiers for different levels of source code (i.e., class, attribute, and method), and ideally, they should have different representations to better capture the properties. However, in our approach, the token 'public' has been assigned only one unique vector representation, which is non-optimal. Another threat to validity is that there is a possibility that the temporal dependencies among code tokens (i.e., the sequence of the source code tokens) may not be captured by the ASTs. However, prior work [11, 21, 63, 97, 169, 204] finds that the structural information performs better for some SE tasks. On the other hand, our way of constructing the training context still can capture such information, if the distance between the sequence of code tokens is within the threshold (i.e., a pre-defined value of the maximum number of AST nodes connecting two leaf nodes, cf., Table 4.1). To better illustrate it, given the following code sequence, "`public static void main`", the temporal dependencies would be "`public`" -> "`static`"-> "`void`"-> "`main`", if we convert the code sequence to an AST, only the structure information of the code sequence is preserved and the sequence information is lost. While, if the distances between these tokens in the source code are within a threshold, by using our method to traverse the AST, we are able to construct the following triples, "`public`" -> "`static`", "`static`" -> "`void`", and "`void`" -> "`main`" (different colors represent different AST paths). And thus we can construct a graph that captures the temporal dependencies, "`public`" -> "`static`" -> "`void`" -> "`main`". Besides, in RQ2, we also treat the source code as a sequence of plain text for embedding generation, which also confirms the findings that, overall, the structural information extracted from the ASTs can benefit code embedding generation for the downstream SE tasks. Another threat is that in RQ2, we examine how the training context (with or without AST) impacts the resulting embeddings and thus the performance of the downstream tasks. Although we only vary the input of the embedding training from the process point of view, the change in the resulting embeddings can impact the training process of the subsequent downstream tasks. More specifically, the changes in code embeddings would lead to different weight values between layers and neurons depending on the embedding during the training process. Thus, the training context is not the sole varying factor of the analysis, and it confounds with other factors such as the training of the downstream tasks. Future work may further investigate the impact of the individual steps (e.g., embedding training) while isolating other steps (e.g., downstream stream task training). In Section 4.3.1, we remove the rare tokens during the preprocessing stage, which may also cause the removal of important tokens, leading to an overfitting model. While, on the contrary, rare tokens mean that there is not enough data for training. As we described in Section 4.2, if the tokens appear only once or twice, the vector (i.e., embeddings) of that token can only be updated limited times (depending on the epoch) which would result in a poor embedding of the code embeddings [19, 149, 182], thus we follow common practice [19, 149, 182] and remove these rare tokens. In RQ1, we provide a qualitative analysis of different embeddings, while the manual inspection may include subjective bias introduced by the individual participants. Future work can consider different graph representations and perform manual analysis to verify our findings.

Construct validity. As described in Section 4.3, we select six different tasks and corresponding models to evaluate the generalizability of the code embeddings. Thus, one of the threats is the quality of the models used in the downstream tasks. In our work, most of the models have a comparable or better performance compared to the work in the literature [72, 88, 179, 204]. Although, in our experiment, the model used in the task of code comment generation performs not as well as the original work [63, 72] (i.e., with a 5.7% performance degradation). This may be caused by the different parameters used for the inference stage and the data separation. Previous work [63, 72] only mentions the parameters for the model training but do not provide the parameters for inference, and unlike what we do in RQ1, they only randomly split the data into training, validation and test sets without a 10-fold cross-validation which also has a non-negligible impact on the results. However, we can still observe the performance changes in the model caused by different code embeddings. Another threat is that the training data used for our embeddings is the *Java-small* dataset. There may exist other datasets that can be used for embedding training. And in order to make a fair comparison with baselines, we only extract the training context based on the methods which may lead to the inadequate use of the class or project-level information from the source code. However, as ASTs can represent the source code with different levels (e.g., method level, statement level, class level, etc.), our method can also be applied to other types of training data. Besides, the edges (i.e., AST paths) in the graph representations are extracted based on the JavaParser tool. JavaParser is a mature tool and has been widely used in various software engineering research. Nevertheless, the quality of the data generated by JavaParser may impact the results of our study. GraphCodeVec requires several hyper-parameters for the training process, such as the dimensions, the number of GCN layers, and the number of training epochs, which may impact the resulting code embeddings. To minimize the bias caused by the hyper-parameter configurations, we follow the practices from prior studies [11, 72, 204] to configure the hyper-parameters. Performing further fine-tuning on these hyper-parameters may further improve the results of GraphCodeVec. In our experiments, we randomly initialize the OOV tokens with real numbers, which may affect the performance of downstream tasks. However, to minimize such influence, we conduct a 10-fold cross-validation for all experiments.

4.7 Conclusions

In this chapter, we introduce a graph convolutional network-based approach, GraphCodeVec, which represents source code as graphs and learns code token embeddings from the context information provided by the graphs. GraphCodeVec trains code token embeddings in an unsupervised way, aiming to improve the generalizability of the learned embeddings. We evaluate GraphCodeVec on an extended benchmark containing six downstream SE tasks. The experiment results show that GraphCodeVec performs comparable or better than all existing code embedding techniques on all SE tasks. Our approach and our pre-trained embeddings can be leveraged by software engineering researchers and practitioners in their downstream tasks that rely on or can be improved by code embeddings. Our work also sheds light on future work that explores different approaches to constructing graph representations of source code and utilizing graph-based deep learning methods to leverage the graph representations.

Part II

Improving the Textual Information in Logging Statements

Chapter 5

Background and Related Work

In this chapter, we first present the background of the textual information in logging statements. We then talk about the prior research that is related to this part.

5.1 Background

Logging statements are inserted by developers in the source code to collect valuable runtime information about software systems. Figure 5.1 shows an example code snippet, which contains a logging statement (line 3). The logging statement has four components: (1) a logging object "`LOG`", (2) a verbosity level "`info`", (3) a dynamic variable "`this.rmAddress`", and (4) a logging text, "`Connected to ResourceManager at `". The content of a logging statement is typically written by developers.

```
1.  private void registerWithRM() throws YarnRemoteException {
2.      this.resourceTracker = getRMClient();
3.      LOG.info("Connected to ResourceManager at " + this.rmAddress);
4.      ...
5.      RegistrationResponse regResponse = this.resourceTracker
            .registerNodeManager(request).getRegistrationResponse();
6.      ...
7.  }
```

Figure 5.1: A code snippet from Hadoop with a logging statement (line 3).

Logging statements produce execution logs at runtime, which play important roles in the daily tasks of developers and other software practitioners [13, 89]. Prior work has leveraged the rich information in logs to support different software engineering activities, including system comprehension [46, 158], anomaly detection [45, 66, 100, 114, 190, 191], and failure diagnosis [132, 164]. In particular, logs are usually the only available resource for diagnosing field failures [197].

5.2 Related Work

Although logs are of much value to software practitioners, the usefulness of logs highly depends on their quality. Both logging too much and logging too little are undesired in practice [89, 199]. There exists a significant challenge for developers to make proper logging decisions. In this section, we categorize prior research into two types: (1) automated logging suggestions and (2) empirical studies on software logging, which align with our proposed two research aspects: (1) proactively suggesting new logging texts and (2) retroactively analyzing existing logging texts.

Automated logging suggestions. Prior research has proposed automated approaches that provide different logging suggestions including the locations of logging statements [47, 88, 193, 206, 210], the verbosity levels [86, 96], the variables to include in a logging statement [108], and the need to update an existing logging statement [87]. The most related work to ours is from He et al. [59], who conduct an empirical study on the usage of natural language descriptions in logging statements and propose an automated logging text generation approach that leverages logging texts from similar code snippets. Their approach has been adopted in the next chapter as the baseline approach (cf., Section 6.3). Recently, pre-trained models of code have achieved new state-of-the-art results for several code-related tasks, such as clone detection, code search, and code completion [54]. Inspired by these advances, Mastropaolo et al. [120] propose to train a Text-To-Text-Transfer-Transformer (T5) model to support the automatic generation of the complete logging statement, including the logging positions, logging levels, and logging texts (the focus of our work). However, although the model is trained on more than 1,000 projects, the generated logging texts only have a BLEU score of 15. Other research aims to detect issues in logging statements. Li et al. [92] perform the first extensive study on applying LLMs for logging statement generation. Chen et al. [23] and Hassani et al. [58] discovered anti-patterns of logging statements from prior log-related code changes and issue reports. Li et al. [94] discuss the issue of duplicate logging statements. Automated tools are designed and implemented to detect these anti-patterns in logging statements.

Despite the above research efforts, providing automated suggestions for logging texts is still challenging. Prior work has highlighted the great importance of the information in the logging texts [89, 198]. Therefore, our work aims to provide automated generation of logging texts to support developers' logging decisions.

Empirical studies on software logging. Empirical studies have been conducted on the practices of logging. The first empirical study on quantitatively characterizing the logging practices was performed by Yuan et al. [199]. Afterwards, follow-up studies by Chen et al. [24] and Zeng et al. [202] extend Yuan et al's study from C/C++ projects to Java projects and Android app projects, respectively. Similarly, Shang et al. [157] conduct a study focusing on the evolution of logging statements. Recently, Li et al. [89] conduct a qualitative study on the benefits and costs of logging based on surveying developers and studying logging-related issue reports. Li et al. [99] conduct a series of interviews with industrial practitioners and investigate their expectations of the readability of log messages. Besides those characteristic studies on logging, empirical studies are also carried out focusing on different aspects of logging practices. The studied topics include the stability of logging statements [68], logging utilities [25] and libraries[67], logging configurations [208], and the relationship between logging practices and software quality [159] and performance [30, 202].

All prior studies provide empirical evidence that shows the challenges in software logging practices, which motivates our work towards retroactive analysis of existing logging texts.

Chapter 6

Proactively Suggesting the Generation of New Logging Texts

Developers insert logging statements in the source code to collect important runtime information about software systems. The textual descriptions in logging statements (i.e., logging texts) are printed during system executions and exposed to multiple stakeholders including developers, operators, users, and regulatory authorities. Writing proper logging texts is an important but often challenging task for developers. Prior studies find that developers spend significant efforts modifying their logging texts. However, despite extensive research on automated logging suggestions, research on suggesting logging texts rarely exists. To fill this knowledge gap, we first propose LoGenText, an automated approach that uses neural machine translation models to generate logging texts by translating the related source code into short textual descriptions. LoGenText takes the preceding source code of a logging text as the input and considers other context information such as the location of the logging statement, to automatically generate the logging text. The LoGenText's evaluation on 10 open-source projects indicates that the approach is promising for automatic logging text generation and significantly outperforms the state-of-the-art approach. Furthermore, we extend LoGenText to LoGenText-Plus by incorporating the syntactic templates of the logging texts. Different from LoGenText, LoGenText-Plus decomposes the logging text generation process into two stages. LoGenText-Plus first adopts a neural machine translation model to generate the syntactic template of the target logging text. Then LoGenText-Plus feeds the source code and the generated template as the input to another neural machine translation model for logging text generation. We also evaluate LoGenText-Plus on the same 10 projects and observe that it outperforms LoGenText on nine of them. According to a human evaluation from developers' perspectives, the logging texts generated by LoGenText-Plus have a higher quality than those generated by LoGen-

Text and the prior baseline approach. By manually examining the generated logging texts, we then identify five aspects that can serve as guidance for writing or generating good logging texts. Our work is an important step towards the automated generation of logging statements, which can potentially save developers' efforts and improve the quality of software logging. Our findings shed light on research opportunities that leverage advances in neural machine translation techniques for automated generation and suggestion of logging statements.

6.1 Introduction

Extensive prior research has shown that writing proper logging statements is an important and challenging task [24, 157, 199, 202]. Besides the typical challenges of deciding where to log [38, 39, 95, 210] and how to choose verbosity levels [86, 165], deciding the textual information in the logging statement is even more challenging [59]. Prior studies find that developers spend significant efforts modifying the textual information in their logging statements [24, 87, 157, 199, 202]. A recent study has shown that developers rely heavily on reading the text in the logging statement while misleading textual information often makes the use of logs counterproductive [89].

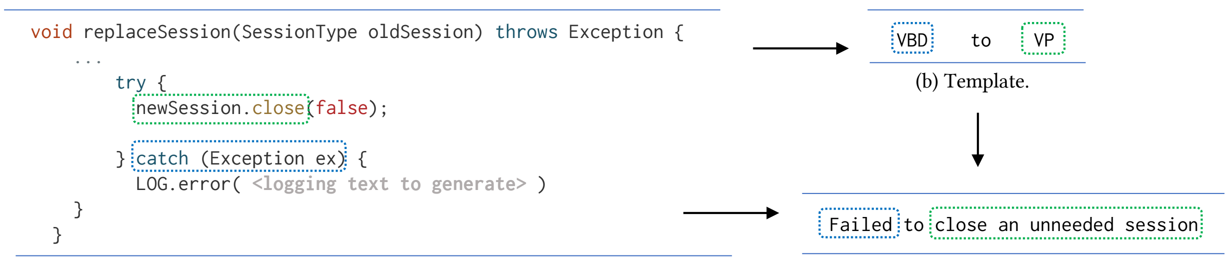
Despite the importance of logging texts, there exists a rare research effort that devotes to assisting developers in writing logging texts. A recent study by He et al. [59] proposes an approach that reuses the texts in the logging statements from similar code snippets. However, since only existing logging texts are directly reused, the texts generated by the prior approach may still require significant revisions by practitioners. Nevertheless, prior work [59] has demonstrated the potential possibility of automatically generating logging texts.

In order to help developers address the challenges of writing logging texts, we propose LoGenText [37], a neural-machine-translation-based approach. LoGenText automatically generates the textual description of a logging statement by translating the related source code into logging texts. Specifically, we adopt a Transformer-based Sequence-to-Sequence model which leverages an encoder-decoder architecture to automate translations and uses the attention mechanism to boost its performance [172]. In LoGenText, the target sequence of the Transformer-based model is a logging text, and the source sequence is its related source code. We consider the source code preceding the logging text as the source input. We also consider incorporating other contexts that may provide relevant information about the logging texts to be generated, including the location of the logging statement, the succeeding source code, and the logging texts in similar code snippets. To incorporate such contexts, we further extend the Transformer by adding additional encoders that integrate the context information into the model [83]. The outputs of these encoders are then formed as a new input to the decoder which generates the logging text as the final output of LoGenText.

We evaluate LoGenText on 10 open-source Java projects from different domains. We first evaluate the automatically generated logging texts by comparing them with the original logging texts inserted by developers using quantitative metrics such as BLEU and ROUGE-L. LoGenText achieves BLEU scores of 23.3 to 41.8 and ROUGE-L scores of 42.1 to 53.9, which outperforms the baseline approach from prior research [59] by a large margin. On the other hand, our evaluation results show that incorporating other context information (e.g., the location of the logging statement) can further improve LoGenText. In order to further understand the effectiveness of LoGenText, we conduct a human-based evaluation that involved 42 participants. The results confirm that LoGenText can provide high-quality logging texts and it significantly outperforms the baseline approach in generating logging texts.

Although LoGenText has achieved superior performance over the baseline approach, it still has limitations. For example, in our previous work [37], we find that there exists a non-negligible gap between the automatically generated logging texts and those written by developers, as LoGenText may generate logging texts that have similar meanings but different syntactic structures from the developer-written logging texts. Meanwhile, recent studies [41, 55, 177, 187, 192] on text generation tasks (e.g., text summarization, sentence generation) show that incorporating the templates of the target sentences can produce promising results in generating better translations, as templates can provide a positive impact for guiding the translation process [192].

Therefore, we further extend LoGenText to LoGenText-Plus to incorporate the template information (i.e., syntactic structures) of the logging texts, which may guide the generation of the logging text. In this work, we use constituency-based parse trees of logging texts from the training set to train an NMT-based model to generate templates. Then, the generated templates are used to guide the generation of the logging texts. Figure 6.1 illustrates the process of generating logging texts based on the source code and templates. In this example, the source code (i.e., Figure 6.1(a)) is used to generate the coarse syntactic template (i.e., Figure 6.1(b)) which contains different levels of information, including the syntactic symbols (i.e., “VBD”, verb with past tense and “VP”, verb phrase) and tokens (i.e., “to”) of the target logging text. Then, the template together with the source code is fed into a Transformer-based model for generating the target logging text. We assume that by using the templates, we are breaking down the task of logging text generation into several stages in a coarse-to-fine manner and the coarse syntactic templates can guide the generation of the target logging texts.



```
void replaceSession(SessionType oldSession) throws Exception {
    ...
        try {
          newSession.close(false);

        } catch (Exception ex) {
          LOG.error( <logging text to generate> )
    }
  }
```

(a) Source code.

```
VBD   to   VP
```

(b) Template.

```
Failed to close an unneeded session
```

(c) Target logging text.

Figure 6.1: An example of the process of generating logging texts based on the source code and templates. "VBD" represents a verb with the past tense, and "VP" represents a verb phrase.

To assess the performance of LoGenText-Plus, we evaluate it on the same 10 projects that were previously used to evaluate LoGenText. We first compare the logging texts generated by LoGenText-Plus with that generated by LoGenText as well as the baseline approach [59] using quantitative metrics. Experiments show that LoGenText-Plus outperforms the baseline approach as well as the best-performing version of LoGenText in nine out of the 10 projects. Besides, we conduct another human evaluation to qualitatively evaluate the quality of the generated logging texts. The results further confirm that LoGenText-Plus can provide higher-quality logging texts.

The contributions of this chapter include:

- Our automated approach LoGenText significantly outperforms the baseline approach in generating logging texts.
- The newly extended approach, LoGenText-Plus, which incorporates the syntactic templates of logging texts further advances the state-of-the-art.
- Our work suggests that automated approaches for logging text generation should not only focus on the preceding code of a logging statement (as done in prior work) but also consider other context information to further improve the performance.
- Our work demonstrates the promising direction of leveraging advances in neural machine translation techniques to generate logging texts.
- Based on the manual evaluation results, we identify five aspects that can be used to generate better logging texts.

Our work is an important step towards the automated generation of logging statements. Our findings shed light on future research opportunities that apply up-to-date neural machine translation techniques in automated generation and suggestion of logging statements. We share our extracted datasets from the 10 open-source projects and the source code used for training our models[1].

Chapter organization. Section 6.2 presents the details of our approach: LoGenText and its extension LoGenText-Plus. Section 6.3 presents the setup of the experiment for evaluating LoGenText and LoGenText-Plus. Section 6.4 and Section 6.5 present the results of evaluating LoGenText and LoGenText-Plus through quantitative metrics and human evaluation. Section 6.6 discusses threats to the validity. Finally, Section 6.7 concludes the chapter.

6.2 Approach

In this section, we describe the details of LoGenText and its extension LoGenText-Plus that leverage neural machine translation (NMT) to automatically generate logging texts.

6.2.1 Approach Overview

6.2.1.1 LoGenText

LoGenText is an NMT-based approach that uses deep neural networks to translate source code into logging texts. The bottom half of Figure 6.2 (i.e., the part delimited by the black dashed lines) illustrates the overall approach of LoGenText which consists of three phases. First, for each logging statement in the source code, LoGenText extracts its logging text, the source code preceding the logging text (i.e., the pre-log code), and the context information from the source code (i.e., **data preparation**). Then, LoGenText feeds the extracted logging text, the pre-log code (i.e., the source), and the context information into a Transformer-based Sequence-to-Sequence (Seq2Seq) model [172] that consists of embedding layers, encoders, and decoders (i.e., **model training**). Finally, the trained model takes the source (the pre-log code) and the context information as input and translates it into the corresponding logging text (i.e., **model inference**).

In the base form of LoGenText, we use the pre-log code of a logging statement to generate its logging text. We evaluate the base form of LoGenText in RQ1 (Section 6.4-RQ1). In RQ2 (Section 6.4-RQ2) and RQ3 (Section 6.4-RQ3), we propose a context-aware

[1]Replication package: `https://tinyurl.com/4njsbu9m`

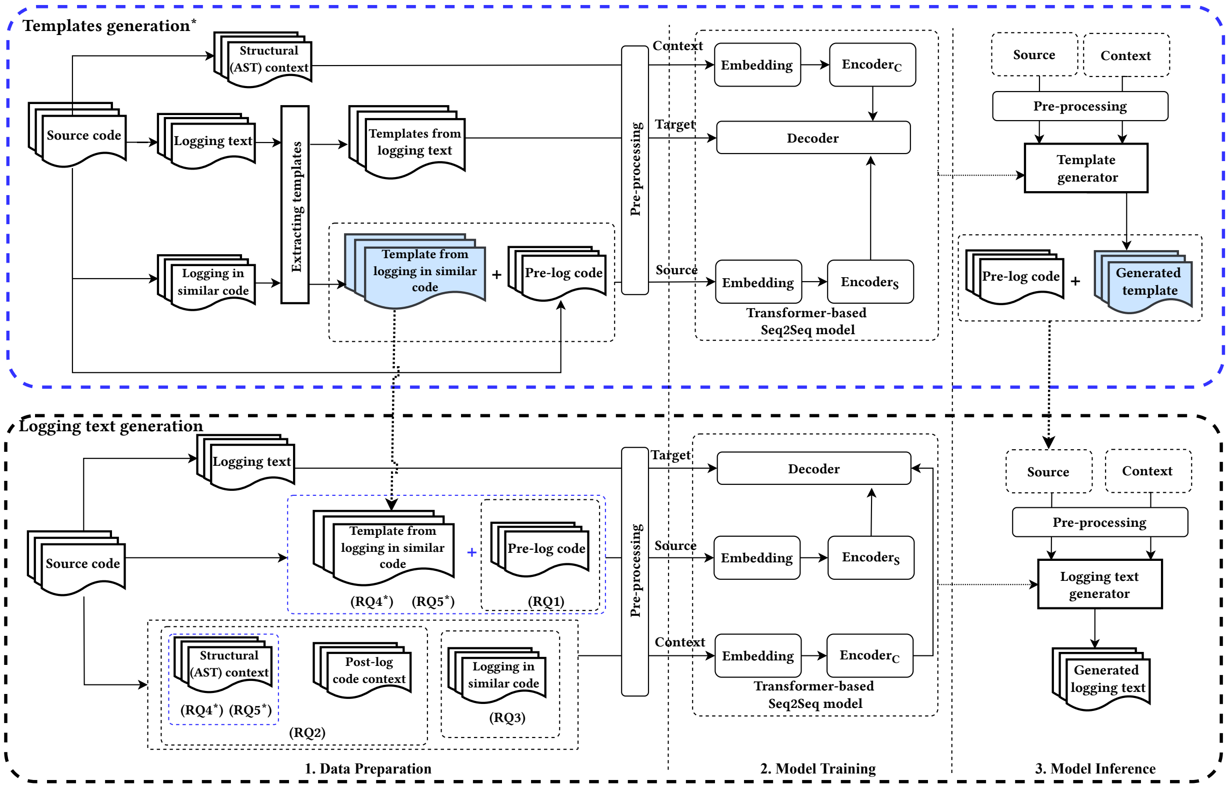


Figure 6.2: An overview of LoGenText and its extension (LoGenText-Plus) which is delimited by the blue dashed lines and highlighted with *. In LoGenText-Plus, to train the template generation model, the templates extracted from logging texts are used as the target sequence, the templates extracted from logging texts in similar code (highlighted in blue) are concatenated with the pre-log code as the source, and the structural (AST) context is used as the context; To train the logging generation model, the logging texts are used as the target sequence, the templates extracted from logging texts in similar code (highlighted in blue) are concatenated with the pre-log code as the source, and the structural (AST) context is used as the context. During inference, the generated templates are concatenated with the pre-log code as the source, and the structural (AST) context is used as the context.

form of LoGenText and discuss the impact of adding the context information, including the location of the logging statement in the abstract syntax tree (AST) (i.e., the structural (AST) context), the source code succeeding the logging statement (i.e., the post-log code), and the logging text in the most similar code snippet, on the performance of LoGenText. The pre-log code is fed as the *source*, while other context information is fed as the *context* to the model.

6.2.1.2 LoGenText-Plus

Besides, we extend LoGenText to LoGenText-Plus by incorporating the template of the target logging text. Different from LoGenText, LoGenText-Plus contains two stages:

Stage 1: template generation, where LoGenText-Plus adopts a Transformer-based model to predict the templates. As shown in Figure 6.2, during **data preparation**, for each logging statement in the source code, LoGenText-Plus first extracts four types of information (i.e., the target logging text, the pre-log code, the structural (AST) context information and the logging text in the most similar code (cf. Section 6.4-RQ3)) [37] which are also used in LoGenText. Then LoGenText-Plus extracts the syntactic template from the logging text as the target sequence for training and from the logging text in similar code which is concatenated with pre-log code as the source sequence. During **model training**, LoGenText-Plus feeds the template from the target logging text (i.e., target sequence), the pre-log code together with the template from the logging text in the similar code (i.e., the source sequence), and the structural (AST) context into a Transformer-based model. Finally, during **model inference**, the trained model (i.e., Template generator) takes the source sequence (i.e., the pre-log code together with the template from the logging text in the similar code) and the structural (AST) context information as input and translates it into the corresponding template (i.e., Generated template).

Stage 2: template-based logging text generation, where LoGenText-Plus adopts another Transformer-based model to predict the final logging text based on the generated template in stage 1. As shown in Figure 6.2, during **data preparation**, similar to that of the template generation stage, for each logging statement in the source code, LoGenText-Plus first extracts four types of information (i.e., the target logging text, the pre-log code, the structural (AST) context information and the logging text in the most similar code). Then LoGenText-Plus extracts the syntactic template from the logging text in similar code which is concatenated with pre-log code as the source sequence. During **model training**, LoGenText-Plus feeds the target logging text (i.e., target sequence), the pre-log code together with the template from the logging text in the similar code (i.e., the source sequence), and the structural (AST) context into the Transformer-based model. Finally, during **model inference**, the trained model (i.e., Logging text generator) takes the source sequence (i.e., the pre-log code together with the generated template in the template generation stage) and the structural (AST) context information as input and translates it into the corresponding logging text (i.e., Generated logging text).

In RQ4 (Section 6.4-RQ4) and RQ5 (Section 6.4-RQ5), we detail LoGenText-Plus and discuss the impact of incorporating different templates on its performance.

6.2.2 Data Preparation

In this part, we describe the steps for preparing data that are required by both LoGenText and LoGenText-Plus.

Data preparation involves three types of information: (1) *logging text*, which refers to the static plain text in the logging statement, (2) *(part) source*, which contains the pre-log code, and (3) *context*, which includes the structural (AST) context, the post-log code, and the logging text in similar code.

The steps for preparing data that are required by both LoGenText and LoGenText-Plus are as follows. Details for step **template extraction** specific to LoGenText-Plus can be found in Section 6.4-RQ4, in which we evaluate LoGenText-Plus.

Extracting the logging text. We first extract the complete logging message (including the logging text and variables) from the logging statement. Since our focus is on the logging text, we then replace the variables with a wildcard (<*vid*>). For example, given the following logging statement from Hadoop, the extracted logging text is "`Removed child queue: <vid>`".

```
LOG.debug("Removed child queue: {}",
    cs.getQueueName());
```

(a) Original logging statement.

```
"Removed child queue: <vid>"
```

(b) Extracted logging text

Extracting (part of) the source data. We use the pre-log code as the main input (i.e., the *source* data) for LoGenText. Specifically, the *source* data includes the code from the method start point to the location right before the logging text of the logging statement. We consider the pre-log code as our main input for logging text generation because a logging statement usually communicates the runtime behavior of the system before the execution of the logging statement [47, 89].

Extracting the context data. We consider three types of data as the *context* input of our neural translation model, including the structural (AST) context, the post-log code context, and the logging text in similar code. We discuss the details of extracting the structural context and the post-log code context in RQ2 where we discuss the impact of such contexts. Similarly, we discuss the details of extracting the logging text in similar code in RQ3.

Pre-processing the logging text and source data. Following the previous approaches for pre-processing the input text data [59, 61, 88], we convert the logging text and source code text into lower cases and tokenize them into token units. We also remove all the non-identifiers (e.g., quotation marks).

A potential challenge is the out-of-vocabulary (OOV) tokens of the source code and logging texts [60, 63]. At testing time, there would be tokens that have never occurred in the training data, which may lead to the poor translation of the NMT systems [111]. One way to alleviate the OOV problem is to enlarge the dictionary size to include more rare tokens. However, due to the fact that user-defined identifiers (i.e., not reserved by the programming language) take up the majority of code tokens, they have a non-negligible influence on the vocabulary of translation dictionary [63]. Thus, using a large dictionary to cover the user-defined tokens would increase the difficulty of training the translation model, as it requires more training data and hardware resources [63]. To address this problem, we employ byte pair encoding (BPE), a data compression technique, to segment the code tokens into subword units [48, 155]. This is based on the intuition that users often define identifiers via combining smaller word units. For example, the token "`getQueueName`" is a combination of three subwords, i.e., "`get`", "`queue`" and "`name`". In this way, our approach can encode all tokens as sequences of subword units.

Note that sometimes, preserving the original case of the source code provides more information, which can be useful for certain code-related tasks. For example, in the task of logging variable recommendations (i.e., which variable in the source code should be logged), the capitalization information can be a strong signal for identifying the variables. However, the authors of BERT also note that typically, the uncased model is better[2] for a range of downstream tasks. In our experiments, our goal is to generate the textual description in the logging statements. Although using BPE tokenization allows us to have relatively good coverage with small vocabularies, unknown tokens still exist. Therefore, we lowercase the source code and logging text to further reduce the out-of-vocabulary (OOV) tokens, especially considering the fact that the capitalization might be inconsistent between the source code and logging text. For example, in the project ActiveMQ[3], the source code contains a variable named "`sendShutdown`", but in the logging statement the token "`shutdown`" is used.

We set the maximum length of both the logging text sequences and the source code sequences to 1,024 (the default value of our Transformer-based model). The tokens of the sequences beyond the maximum length will be truncated; the sequences shorter than the maximum length are padded. 0% of the logging text sequences are truncated and 3.7% to 3.8% of the source code sequences are truncated in the studied projects.

[2]https://github.com/google-research/bert#pre-trained-models

[3]https://github.com/apache/activemq/blob/905f00c843b96996b25017e1b8646de15d703398/activemq-broker/src/main/java/org/apache/activemq/network/DemandForwardingBridgeSupport.java#L324

6.2.3 NMT-based Log Generation

In this work, we consider the logging text generation task as a machine translation task, i.e., translating a code snippet into logging text that communicates the internal behavior of the code snippet. Thus, we can apply neural machine translation (NMT) techniques to solve the logging text generation problem. Formally, given a source sequence $X = (x_1, x_2, \ldots, x_S)$, our goal is to predict tokens in the target logging text $Y = (y_1, y_2, \ldots, y_T)$. Most NMT models use an encoder-decoder architecture. The input to the encoder is the source sequence X, and the output of the encoder is a sequence of distributed representations. The generated representations are then fed into the decoder part, where the tokens in the target sequence are generated one by one [174]. Hence, the objective of the models is to approximate the conditional distribution $\log P\,(Y|X;\theta)$ over the given source-target pairs and model parameters θ.

Our model is also based on an encoder-decoder model, in particular, the Transformer model proposed by Vaswani et al. [172], which has shown outstanding performance in many software engineering tasks (e.g., source code summarization [4] and code completion [104]). Figure 6.3 illustrates the structure of the Transformer translation model that is implemented in LoGenText and LoGenText-Plus. Note that the two models used in the different stages of LoGenText-Plus are both based on the Transformer model. Like many other sequence to sequence models, the Transformer utilizes an encoder-decoder structure, which is explained in detail in the rest of this section.

Source encoder: As Figure 6.3 shows, the source encoder component makes use of N stacked layers. Each layer is broken down into two sub-layers. The first sub-layer is a self attention layer:

$$Attention\,(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{6.1}$$

where Q, K, V are the query, key, and value vectors, $\sqrt{d_k}$ is a normalization factor and d_k is the dimension of the key/query vector, *Attention* is the output of the attention layer. The self attention mechanism allows the model to look at other positions for extra information while encoding the current position.

The residual connection and layer normalization are then applied to the output of the attention layer:

$$LayerNorm\,(Attention + X) \tag{6.2}$$

where X is the vector representation of the input token after positional encoding (explained in the next paragraph). The output is then fed to the second sub-layer, a fully connected feed forward network. Note that the feed forward network is point-wise, which means the

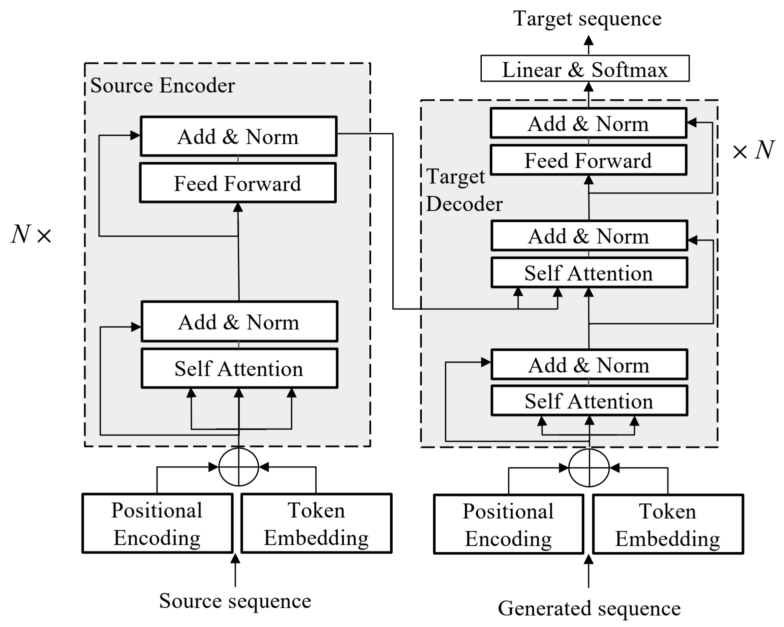


Figure 6.3: An overview of the Transformer translation model.

network is applied independently to individual vectors generated by the attention layer.

Positional encoding: The orders of the tokens in the source sequence are important for a machine translation model. To address this, unlike RNN and its variances, Transformer adopts positional encoding to inject the relative positional information into the token representations. Specifically, a positional vector is added to the input embedding, where the positional vector pe for t^{th} token is calculated as follows:

$$pe_t^i = \begin{cases} \sin(w_k \cdot t) & \text{if } i = 2k \\ \cos(w_k \cdot t) & \text{if } i = 2k+1 \end{cases} \tag{6.3}$$

where k is used for determining whether i is an odd or even number, $i \in \{0, \ldots, d-1\}$ is the encoding index, d is the dimensionality of the input embedding, and $w_k = \frac{1}{10000^{2k/d}}$. The final token representation that is fed into the self attention layer is a sum of the token embedding and the positional encoding.

Context encoder: The structure of the context encoder is the same as the source encoder. As the *context* inputs (i.e., the structural context, the post-log code context, and logging text in similar code) are only discussed in RQ2 and RQ3, we describe the details about how we integrate the *context* into our model in RQ2.

Target decoder: The decoder in Transformer has a similar structure to the encoder. It also consists of N stacked layers, with three sub-layers in each layer (slightly different from the two sub-layers in the source encoder). The additional second sub-layer takes the source encoder's output and the decoder's states which are generated by the first self attention sub-layer. Besides, an attention masking is applied to the first self attention sub-layer. This masking prevents future information from being leaked to the decoder before the prediction and ensures that the predictions only rely on the previous outputs.

Given a source code and logging text corpus D, the goal of training the Transformer model is to find parameters θ that maximize the log-likelihood of the training data:

$$\widehat{\theta} = \arg\max_{\theta} \sum_{\langle X,Y \rangle \in D} \log P\left(Y|X, Y^{pre}; \theta\right) \tag{6.4}$$

where P is the conditional probability of the target sequence Y (i.e., the logging text) given the source sequence X (i.e., the source code) and the previous sequence Y^{pre}.

Note that in LoGenText-Plus, we also consider the template generation as a machine translation task, i.e., translating a code snippet into a template that provide the syntactical information of the logging text. The model shares the same structure as in Section 6.2.3, where the target sequence is changed to the template. As the template information is only used in RQ4 and RQ5, we describe the details about how to generate templates in RQ4 and RQ5.

6.3 Evaluation Setup

6.3.1 Subject Projects

We evaluate our proposed LoGenText and LoGenText-Plus on 10 open-source Java projects. We choose the same subject projects that are used in prior work [59] which studies the characteristics of logging texts. The details of the studied versions of these projects are listed in Table 6.1. The source lines of code of the studied projects range from 330K to 1.7M. These projects have about 2K to 12K logging statements, among which 76.2% to 95.8% have logging texts. Similar to prior work [59], we evaluate LoGenText and LoGenText-Plus on the logging statements with logging texts.

Table 6.1: Details of the studied projects.

Project	**Version**	**SLOC**	**# of logging statements**	**# of logging statements with text**
ActiveMQ	5.16.0	415k	2,185	2,093 (95.8%)
Ambari	2.7.5	490K	4,150	3,651 (88.0%)
Brooklyn	1.0.0	339K	2,937	2,813 (95.8%)
Camel	3.4.2	1.4M	7,046	6,366 (90.3%)
CloudStack	4.14.0	645K	12,015	10,613 (90.3%)
Hadoop	3.3.0	1.7M	12,471	11,270 (88.3%)
HBase	2.3.0	778K	5,534	5,071 (90.4%)
Hive	3.1.2	1.7M	6,845	6,290 (91.6%)
Ignite	2.8.1	1.1M	3,366	3,048 (90.6%)
Synapse	3.0.1	330K	1,978	1,508 (76.2%)
Avg.		890K	5853	5272 (90.1%)

6.3.2 Experimental Settings

6.3.2.1 Model training settings

The goal of LoGenText and LoGenText-Plus is to use the Transformer-based model to automatically generate logging texts with the source code as the input. Our LoGenText and LoGenText-Plus are implemented based on Fairseq [83, 138], a sequence-to-sequence modeling toolkit. We use the same model structure as in the original Transformer model: six stacked layers (i.e., $N = 6$), 512 embedding dimensions for both the source encoder and the target decoder, and 2,048 feed-forward embedding dimensions. We use the Adam optimizer to optimize the model parameters (same as the original Transformer model). To prevent overfitting, we use a dropout rate of 0.1 [44, 54, 118, 119, 120]. More details about the configuration of hyperparameters can be found in our replication package[4].

For each subject project, we split all the instances into 80%/10%/10% training/validation/testing sub datasets[5]. As the number of instances in each subject project is relatively small (i.e., about 1.5K to 11K), it is challenging to fit a Transformer model with more than forty million parameters. To overcome this problem, we adopt a two-stage training strategy (*a.k.a.,* transfer learning (TL)) [52, 135, 195]: for each subject project, (1) we first pre-train a model using all the training sets from 10 projects for 50 epochs, and (2) we then continue to fine-tune the pre-trained model parameters using the target project's training

[4]https://github.com/conf-202x/experimental-result

[5]The sizes of training datasets range from 1K to 9k.

set for another 50 epochs. The idea is inspired by the work of He et al. [59], where the authors have shown that the logging practices are quite different across different projects, and the n-gram patterns in different projects vary a lot. Meanwhile, a large project usually has a long development history (e.g., years). By fine-tuning a model for a specific project using its existing data, we can leverage the model to suggest logging text for its future development activities. Therefore, we intentionally train separate models for each project, aiming to accurately capture the in-project language patterns while avoiding the (negative) impact of other projects. The validation set is used to monitor the performance of the model during training to avoid overfitting.

For inference, we use the beam search with a width of eight, which means at each step, the top eight candidate tokens with the highest scores are kept for the next step. However, the beam search algorithm favors shorter sequences [20, 133]. To address this problem, we adopt the length penalty, which gives favor to longer sequences [188]. In our experiments, we set the value of the length penalty to 2.5. In addition, we set the maximum length and minimum length of the generated logging text to be 100 and 3, respectively, as we find that the lengths of 92.4% to 98.4% of the logging texts in the studied projects fall in this range.

The training of our models is conducted in a cluster of machines each with an NVIDIA V100 Tensor Core GPU.

6.3.2.2 Model evaluation approaches

We evaluate the performance of LoGenText and LoGenText-Plus using a combination of quantitative evaluation and human evaluation.

Quantitative evaluation: We use two widely used machine translation evaluation metrics, BLEU [141] and ROUGE [102], to evaluate the quality of the generated logging text sequences in terms of their similarity to the original logging texts inserted by the developers. The details of these evaluation metrics are described in the research questions that apply these metrics.

Human evaluation: In order to evaluate how developers perceive the generated logging texts, we also performed a human evaluation, which is detailed in Section 6.5.

6.3.3 Baseline Approach

We compare our approach with prior work by He et al. [59], which is by far the state-of-the-art approach for generating logging texts. Their method assumes that similar code snippets tend to have similar logging texts. To generate the logging text for a given code snippet, He et al. [59] perform a search in the training corpus to retrieve the most similar code snippet based on Levenshtein distance [81]. The logging text of the most similar code snippet is used as the logging text for the given code snippet. We re-implement their method as a baseline to compare with our approach.

6.4 Evaluation Results

In this section, we discuss the results of evaluating LoGenText and LoGenText-Plus through answering six research questions. More specifically, the first three research questions (i.e., RQ1-RQ3) are related to LoGenText, and RQ4-RQ6 are newly proposed research questions and related to its extension, LoGenText-Plus.

RQ1: How well can the base form of LoGenText automatically generate logging text?

Motivation.

Prior research [59] has observed that logging texts are predictable and proposes a simple approach (the baseline approach in Section 6.3) based on the intuition that similar code snippets contain similar logging texts. Such a simple approach has demonstrated a promising result. Therefore, in this RQ, we would like to explore whether our NMT-based solution (i.e., LoGenText) can automatically generate logging texts with a better performance than the baseline approach.

Approach.

We evaluate the base form of LoGenText, i.e., using only the source input (pre-log code) to generate the logging texts and compare it with the baseline approach [59]. Following prior work [59], we evaluate the quality of the generated logging texts using two widely used metrics for machine translation evaluation, i.e., BLEU[6] [141, 143] and ROUGE[7] [102]. Both BLEU and ROUGE take the automatically generated logging texts and the reference logging texts (i.e., the original logging texts written by developers) as input and calculate the similarity between them, which outputs a percentage score between 0 and 1. The higher the score, the better the generated logging texts in terms of their similarity to the reference logging texts.

[6]https://github.com/mjpost/sacrebleu
[7]https://github.com/pltrdy/rouge

BLEU (Bilingual Evaluation Understudy) is used to evaluate the match between a generated text and a reference text, which is calculated as follows:

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{6.5}$$

$$BP = \begin{cases} 1 & if\ c > r \\ e^{(1-r/c)} & if\ c \leqslant r \end{cases} \tag{6.6}$$

where p_n is the modified n-gram precision (i.e., the maximum number of n-grams co-occurring in the automatically generated logging text and the reference logging text divided by the total number of n-grams in the generated logging text), w_n are positive weights that can be configured, BP is a brevity penalty, c is the length of the generated logging text and r is the length of the reference logging text. In our evaluation, we choose $N = 4$ and uniform weights $w_n = 1/N$, same as prior work [59]. In addition to the overall BLUE score, we also consider the specific BLEU-n (n = 1, 2, 3, 4) scores, which are the BLUE scores considering only one gram size.

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a set of metrics for evaluating automated generated texts in text summarization and translations. ROUGE is calculated as follows:

$$\text{ROUGE-n} = \frac{\sum_{gram_n \in Ref} Count_{match}(gram_n)}{\sum_{gram_n \in Ref} Count(gram_n)} \tag{6.7}$$

where n is the length of the n-gram ($gram_n$), and $Count_{match}(gram_n)$ is the number of n-grams co-occurring in the automatically generated logging text and the reference logging text, Ref. We calculate ROUGE-1, ROUGE-2 and ROUGE-L. ROUGE-L measures the longest matching sequence of tokens using LCS (Longest Common Subsequence).

Results.

Our base form of LoGenText generally outperforms the baseline approach. Our experimental results of comparing LoGenText with the baseline on the 10 studied projects are presented in Table 6.2. The best results are highlighted in the **bold** font. We can see that the base form of LoGenText provides a ROUGE-L score of 41.1 to 52.3 and a BLEU score of 21.8 to 39.0 for the studied projects. As shown in Table 6.2, LoGenText outperforms the baseline approach for all the projects in terms of ROUGE-L by 5.7% to 22.8% and has a higher BLEU score than the baseline approach by 2.9% to 18.5% in seven out 10 projects. In addition, besides the overall BLEU and ROUGE-L, LoGenText performs better than the baseline approach in almost all different gram sizes (i.e., BLEU-

Table 6.2: Evaluation results of using LoGenText and the baseline approach to generate logging texts in the studied projects (RQ1).

Projects	Methods	BLEU(%)	BLEU-1(%)	BLEU-2(%)	BLEU-3(%)	BLEU-4(%)	ROUGE-L(%)	ROUGE-1(%)	ROUGE-2(%)
ActiveMQ	Baseline	21.0	37.0	22.9	18.4	**16.0**	36.1	36.0	21.6
	LoGenText	**23.0(+9.5%)**	**44.6**	**26.0**	**19.6**	**16.0**	**43.4(+20.4%)**	**43.1**	**25.1**
Ambari	Baseline	19.9	36.8	22.0	17.0	**14.1**	36.8	37.5	22.4
	LoGenText	**22.8(+14.6%)**	**44.0**	**25.6**	**17.8**	13.4	**42.9(+16.5%)**	**44.1**	**24.7**
Brooklyn	Baseline	**26.0**	41.4	25.5	**21.8**	**19.7**	38.1	40.9	23.0
	LoGenText	25.4(-2.1%)	**48.7**	**28.4**	20.8	16.8	**43.6(+14.4%)**	**47.1**	**26.2**
Camel	Baseline	37.9	51.5	39.2	35.6	33.8	47.5	47.9	33.0
	LoGenText	**39.0(+2.9%)**	**58.3**	**43.3**	**38.1**	**35.9**	**52.3(+10.2%)**	**52.5**	**35.3**
CloudStack	Baseline	30.1	46.6	33.5	28.4	25.4	43.9	44.5	30.0
	LoGenText	**34.6(+14.7%)**	**52.4**	**37.3**	**30.0**	**25.6**	**50.1(+14.0%)**	**50.8**	**35.2**
Hadoop	Baseline	19.6	37.2	22.8	18.7	**16.8**	34.1	34.9	20.1
	LoGenText	**21.8(+11.1%)**	**44.4**	**25.4**	**19.1**	16.5	**41.1(+20.5%)**	**42.3**	**23.0**
HBase	Baseline	19.5	38.4	24.2	19.4	15.9	38.4	38.9	26.1
	LoGenText	**23.1(+18.5%)**	**46.1**	**28.2**	**21.6**	**17.2**	**46.5(+21.2%)**	**47.0**	**30.6**
Hive	Baseline	**28.2**	42.9	29.8	**26.2**	**24.0**	42.4	42.9	28.9
	LoGenText	28.0(-0.6%)	**47.4**	**30.8**	25.2	21.7	**46.7(+10.2%)**	**47.2**	**29.8**
Ignite	Baseline	21.5	38.5	23.4	18.4	14.8	37.1	38.0	22.9
	LoGenText	**24.9(+15.6%)**	**50.9**	**30.7**	**23.3**	**18.3**	**45.5(+22.8%)**	**47.2**	**27.1**
Synapse	Baseline	**34.1**	46.7	**36.7**	**31.7**	**27.2**	46.9	46.8	**36.9**
	LoGenText	28.9(-15.3%)	**53.3**	34.7	26.7	21.5	**49.5(+5.7%)**	**50.2**	32.0
Avg.	Baseline	25.8	41.7	28.0	23.6	**20.8**	40.1	40.8	26.5
	LoGenText	**27.1(+5.0%)**	**49.0**	**31.1**	**24.2**	20.3	**46.1(+15.0%)**	**47.2**	**28.9**

Note: The numbers in the brackets indicate the relative change of LoGenText to the baseline approach.

n and ROUGE-n). Our results indicate the promising research direction of using neural translation techniques in the automated generation of logging text.

On the other hand, we also observe that the base form of LoGenText may not always provide a better performance in terms of BLEU scores (e.g., BLEU-4). As shown in Table 6.2, LoGenText performs better than the baseline approach for seven out of 10 projects in terms of BLEU but worse for the other three projects (Brooklyn, Synapse, and Hive). By examining the BLEU scores of different gram sizes (i.e., BLEU-n), we realize that the base form of LoGenText always outperforms the baseline in terms of smaller gram sizes (i.e., BLEU-1 and BLEU-2); in some cases (e.g., for the projects Brooklyn, Synapse, and Hive), the base form of LoGenText may not perform better than the baseline approach in terms of larger gram sizes (i.e., BLEU-3 and BLEU-4). This phenomenon can be explained by the different working mechanisms of these two different approaches. The baseline approach simply reuses logging texts from other code snippets [24], thus it tends to produce long sequences of identical tokens between code snippets, which can result in relatively high larger-gram BLEU scores, especially when there are many duplications of logging texts [94]. In contrast, LoGenText automatically generates new logging texts token by token, thus it may not always produce long sequences of tokens that are identical to the ones written by developers, even though the generated ones may have similar semantic meanings with the written ones, as discussed in our user study in Section 6.5.

Summary

The base form of our NMT-based approach LoGenText generally outperforms the baseline approach that leverages the existing logging texts in similar code snippets. Our results illustrate the promising future research opportunity of formulating automated logging text generation as a neural machine translation task.

RQ2: Can incorporating context information improve the base form of LoGenText in generating logging texts?

Motivation.

Prior studies [14, 76, 115, 116, 174, 175, 183, 203] on NMT show that incorporating the context information (e.g., surrounding text) of the source input may provide promising results in generating better translations. In addition, the context information (e.g., surrounding source code, AST structure of source code) of a particular source code of interest has shown benefits in some software engineering (SE) tasks that rely on neural network-based techniques [11, 21, 63, 169, 204]. Therefore, in this research question, we aim to understand whether the context information (e.g., the post-log code and the structural (AST) information of a logging statement) can help further improve LoGenText in automatically generating logging texts.

Approach.

We propose a context-aware form of LoGenText and consider two types of context information in this research question: the post-log code and the structural (AST) information related to a logging statement. Below we discuss how we extract such context information and incorporate it in LoGenText.

Extracting context information. *Extracting the structural (AST) context:* We use AST extracted by *srcML* [31] to represent the location of a logging statement. The structural information represented by the AST has been applied successfully in many SE tasks, including suggesting *where to log* [210] and *how to choose log levels* [86]. First, we extract the AST of the method containing the logging statement. Then, we convert the AST into a sequence of AST node types (e.g., if statement) following a preorder traversal. We only keep the sequence of AST node types prior to the logging statements.

Extracting the post-log code context: Although a logging statement is usually not directly related to the subsequent code (i.e., post-log code), prior research [59] shows the post-log code may provide some extra information relevant to the logging text. Therefore, we consider the post-log code as the context input instead of the source input in our NMT-based model. Specifically, the post-log code contains the code from the location that immediately follows the logging statement to the end of the containing method. We use the same approach as the pre-log code (cf., Section 6.2) to convert the post-log code into a sequence of code tokens.

Integrating context information in our models. There are mainly two approaches for integrating the context information in NMT-based models: (1) simply concatenating the context and the source as a new input sequence [3, 168], and (2) utilizing a multi-encoder model, where additional neural networks are used to encode the context [83, 174, 203]. Prior work [83, 174] shows that the multi-encoder approach is more effective for incorporating context information in NMT tasks. We experimented with both approaches and we also found that the multi-encoder approach shows better performance in our context. Therefore, we use the multi-encoder approach in this chapter.

The structure of our context integration approach is illustrated in Figure 6.4. The context encoder replicates the original Transformer encoder and takes one type of context information (e.g., AST context, post-log code context) as input. The output of the context encoder together with the output of the source encoder is then fed into a self-attention layer. Then, the outputs of the attention layer and the source encoder are fused by a gated sum. Formally, let S be the output of the source encoder and C be the output of the

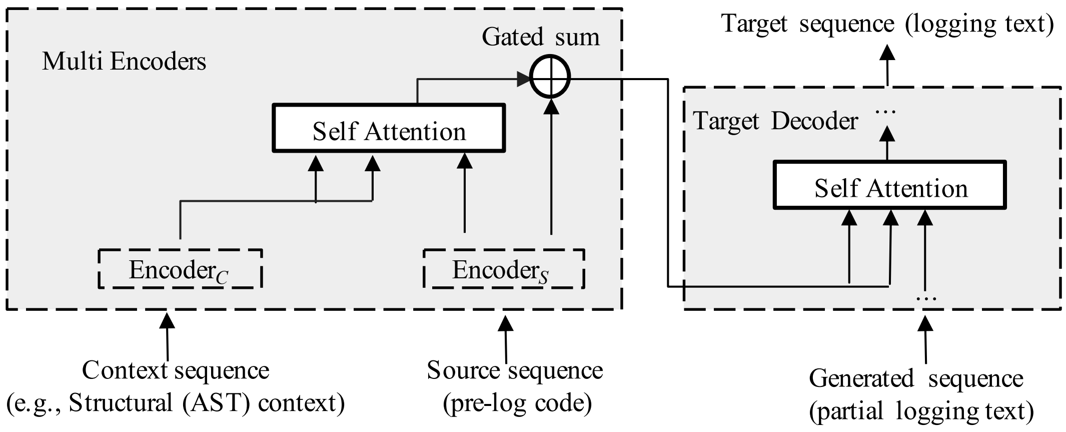

Multi Encoders
Gated sum
Self Attention
Encoder$_C$
Encoder$_S$
Target sequence (logging text)
Target Decoder
Self Attention
Context sequence (e.g., Structural (AST) context)
Source sequence (pre-log code)
Generated sequence (partial logging text)

Figure 6.4: An overview of the multi-encoder Transformer.

attention layer, the output of the gated sum G is

$$G = \lambda \odot C + (1 - \lambda) \odot S \tag{6.8}$$

where the gating weight λ is calculated by

$$\lambda = \sigma \left(W \left[C, S \right] + b \right) \tag{6.9}$$

where $\sigma\left(\cdot\right)$ is the sigmoid function, W is the weight parameters of the model and b is the bias.

In order to understand the impact of different types of context information, we evaluate the performance of the models using each type of context. We use the same metrics used in RQ1 (i.e., BLEU and ROUGE-L) to evaluate the quality of the generated logging texts.

Results.

Incorporating context information can improve the performance of the base form of LoGenText and outperforms the baseline approach in all the studied projects. Table 6.3 shows the results of incorporating different context information. By comparing the context-aware form of LoGenText with the base form, we find that by incorporating the context information using multi-encoder models, we can obtain a performance improvement on almost all the projects. For example, by encoding the structural (AST) context into our LoGenText, we obtain a 29.2% relative (8.4% absolute) increase in terms of BLEU score in project Synapse over the base form of LoGenText. Overall, as shown in Table 6.3, the context-aware form of LoGenText that incorporates the AST context provides a BLEU score of 23.3 to 41.8 and a ROUGE-L score of 42.1 to 53.9 for the studied

Table 6.3: Evaluation results of incorporating contexts (AST, post-log code) in LoGenText for logging text generation (RQ2).

		BLEU(%)										
		ActiveMQ	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
	Baseline	21.0	19.8	26.0	37.9	30.1	19.6	19.5	28.2	21.5	34.1	25.8
	Base LoGenText (RQ1)	23.0	22.8	25.4	39.0	**34.6**	21.8	23.1	28.0	24.9	28.8	27.1
With context	AST	**24.1**	23.8	27.8	**41.8**	**34.6**	**23.3**	23.5	**29.6**	**28.8**	**37.2**	**29.5**
	Post-log code	**24.1**	**24.5**	**28.4**	39.9	34.3	23.1	**24.3**	**29.6**	28.2	34.8	29.1
		ROUGE-L(%)										
		ActiveMQ	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
	Baseline	36.1	36.8	38.1	47.4	43.9	34.1	38.4	42.4	37.1	46.9	40.1
	Base LoGenText (RQ1)	**43.4**	42.9	43.6	52.3	50.1	41.1	**46.5**	46.7	45.5	49.5	46.2
With context	AST	42.5	43.4	44.0	**53.9**	**50.8**	**42.1**	46.4	**48.2**	**47.6**	**53.6**	**47.3**
	Post-log code	42.8	**43.5**	**44.7**	53.6	50.4	41.5	46.3	48.0	46.0	53.4	47.0

Note: Values in bold font indicate the best-performing models.

projects, which are 5.0% to 34.0% and 13.7% to 28.3% higher than the baseline approach, respectively. In addition, unlike the base form of LoGenText which may underperform the baseline approach for certain projects (e.g, Brooklyn and Synapse) in terms of BLEU scores, sizes (i.e., BLEU-3 and BLEU-4), our context-aware form of LoGenText can provide better BLEU scores than the baseline approach for all the studied projects. The results demonstrate that LoGenText can benefit from the extracted context information.

Meanwhile, we observe that for some projects (e.g., Synapse and Camel), different types of context can result in diverse performance. In particular, for the Synapse project, incorporating AST and post-log code results in BLEU scores of 37.2 and 34.8, respectively. This finding suggests that practitioners should be careful with the selection of contexts for different projects, as they may produce diverse results. On the other hand, we also observe that leveraging the AST context performs better than the post-log context in seven out of the 10 projects and has the largest improvement over the base form of LoGenText on average. This observation further confirms the success of applying AST information in suggesting logging activities [86, 210].

We also find that incorporating additional context may not always improve the performance of LoGenText significantly. As shown in Table 6.3, by adding context using the multi-encoders model, the performance on the project CloudStack (using AST context) remains the same as that without the context. This may be due to the fact that CloudStack has a much higher number of pre-log code tokens for each generated logging text (information used in the base form of LoGenText) than other projects, leading to less value in adding the context information.

Additionally, to gain a deeper understanding of why utilizing AST context is more beneficial, we conduct a comprehensive manual analysis. First, we sort the cases in descending order based on the BLEU score gap between utilizing the AST context and utilizing the post-log code. This sorting allows us to identify the cases where the utilization of AST context has the most significant impact on performance improvement. Subsequently, we select the top 10 cases from each project, resulting in a total of 100 cases.

We find that most (i.e., 90%) of the logging statements, where utilizing AST outperforms relying on the post-log code, are describing the preceding source code (i.e., pre-log code). As a result, the succeeding code (i.e., post-log code) would be a source of noise and has a negative impact on the generation of the logging text.

Moreover, to understand how the post-log code would be a source of noise during the logging text generation process, we further manually analyze the characteristics of the post-log code. We find that the noise mainly comes from two aspects: (1) For a testing case, there exist training cases that share exactly the same post-log code, but a different pre-log code. Therefore, utilizing the post-log code would cause LoGenText to (partly) copy from such existing logging texts. It is intuitive that the post-log codes are similar, as developers may put return or exception statements at the end of a method; (2) The post-log code contains irrelevant tokens and thus misleads the generation of the logging text.

In short, all the noisy information can be summarized as the introduction of irrelevant code to the source input. As a result, LoGenText cannot effectively focus on the most important source code to generate the logging text. Moreover, we find that even though in some cases, the logging texts are describing the succeeding source code, they are only related to one or two lines of post-log code. Therefore, using all the post-log code as the context in LoGenText would sometimes decrease the performance of our approach.

Summary

Incorporating context information (AST and post-log code) can improve the performance of the base form of LoGenText for generating logging texts, and different context information may have a diverse impact on the studied projects.

RQ3: Can incorporating logging text from similar code improve the base form of LoGenText in generating logging texts?

Motivation.

Prior work [59] proposes a preliminary logging text generation approach that simply reuses the logging text from the most similar code snippet (i.e., our baseline approach) and achieves promising results. Their results suggest that the logging statement in a similar code may provide additional information about the logging text to be generated. Although we demonstrate better performance of LoGenText than the baseline, it may be the case that the information captured by LoGenText and that captured by the baseline approach

Table 6.4: Evaluation results of incorporating logging text from similar code in LoGenText for logging text generation (RQ3).

		BLEU(%)										
		ActiveMQ	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
	Baseline	21.0	19.8	26.0	37.9	30.1	19.6	19.5	28.2	21.5	34.1	25.8
	Base LoGenText (RQ1)	23.0	22.8	25.4	39.0	**34.6**	21.8	23.1	28.0	24.9	28.8	27.1
With context	Logging text from similar code	**25.8**	**25.3**	**27.5**	**41.6**	34.4	**22.8**	**24.0**	**29.2**	**26.6**	**34.0**	**29.1**
		ROUGE-L(%)										
		ActiveMQ	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
	Baseline	36.1	36.8	38.1	47.4	43.9	34.1	38.4	42.4	37.1	46.9	40.1
	Base LoGenText (RQ1)	43.4	42.9	43.6	52.3	50.1	41.1	46.5	46.7	45.5	49.5	46.2
With context	Logging text from similar code	**44.8**	**44.1**	**43.9**	**53.9**	**50.7**	**41.8**	**46.6**	**47.5**	**46.4**	**53.1**	**47.3**

NNote: Values in bold font indicate the best-performing models.

do not overlap. Given that including the information provided by the baseline may further improve the results, in this research question, we aim to explore the impact of incorporating logging text in similar code on automated logging text generation and examine whether we can improve the base form of LoGenText by utilizing such logging information.

Approach.

Similar to prior work [59], we leverage the logging texts from similar code snippets in the generation of logging texts.

Extracting logging text from similar code. For each logging statement, we extract its pre-log code and search for the most similar code snippet in the training dataset. Specifically, for a given pre-log code snippet, we follow prior work [59] and use the Levenshtein distance [81] to calculate the similarity between it and other code snippets in the training dataset. We then extract the logging text in the most similar code snippet.

Incorporating logging text from similar code. We adopt the same multi-encoder approach as in RQ2 to incorporate the retrieved logging text from similar code. In particular, the logging text in a similar code snippet is encoded using a context encoder, and then a gated sum is applied to the outputs of the context encoder and the source encoder, the output of the gated sum is then fed to the target decoder.

Similar to RQ1 and RQ2, we evaluate the performance of LoGenText that incorporates the logging text from the similar code using the BLEU and ROUGE-L metrics.

Results.

Incorporating logging text from similar code can improve the performance of the base form of LoGenText. As shown in Table 6.4, we find that by incorporating the retrieved logging text from similar code using a context encoder, the performance of the base form of LoGenText can be increased in nine out of the ten studied projects (e.g., the average BLEU score increases from 27.1 to 29.1). The results indicate that the logging in similar code may contain useful knowledge for the logging text to be generated in the NMT model. However, incorporating the logging text from similar code (with an average BLEU of 29.1) is less effective than the LoGenText that incorporates the AST context (with an average BLEU of 29.5, cf. RQ2).

Similar to our results in RQ2, incorporating logging text from similar does not improve the performance on the CloudStack project over the base form of LoGenText. Similarly, this result may be due to the fact that CloudStack has a large number of pre-log code tokens for each generated logging text (information used in the base form of LoGenText), which may lead to less value of incorporating the additional logging information from similar code.

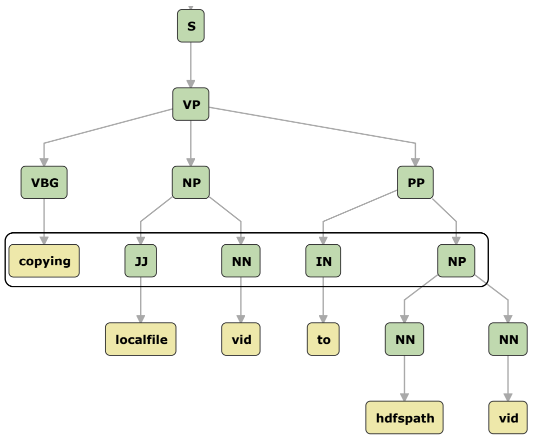
Summary

Incorporating logging text from similar code can provide additional information to the base form of LoGenText. However, it cannot further improve the best-performing version of LoGenText that incorporates the AST context.

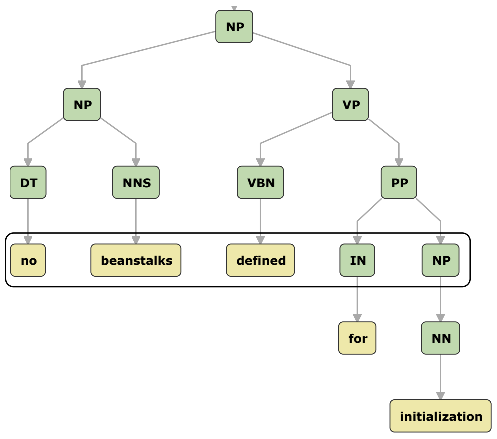
RQ4: Can incorporating the template information into LoGenText-Plus improve LoGenText in generating logging texts?

Motivation.

Prior studies [41, 55, 177, 187, 192] on text generation tasks (e.g., text summarization, sentence generation) show that incorporating the template information of the target sentences can provide promising results in generating better texts. For example, Yang et al. [192] use syntax-based templates to guide the translation procedure and outperform the baseline models in the task of neural machine translation. In addition, based on the manual inspection of the two generated logging texts in the section of human evaluation provided by Ding et al. [37], we find that the syntactic structures of the two logging texts are different from each other and carry distinct information for each logging text. Such variety and specificity raise the question of whether we can extract syntactic templates from the syntactic structures and adopt templates to guide the automatic logging text generation process. Figure 6.5 are two different constituency-based parse trees produced by Stanza [146] for two logging texts. Based on these two syntactic trees, we may construct two syntactic templates (the construction process is elaborated in the following section ***Approach***.), "copying JJ NN IN NP" and "no beanstalks defined IN NP", where JJ NN IN NP are non-terminal symbols of the parse tree, representing different token syntactic abstractions (e.g., NP refers to a noun phrase and NN means noun). As illustrated in Figure 6.5, templates are abstract representations of logging texts that encompass the syntactic characteristics of logging texts and may serve as a guide when generating logging texts. Therefore, in this research question, we aim to understand whether the syntactic template information can help further improve LoGenText in automatically generating logging texts.



(a) The constituency-based parse tree of the logging text, "`copying localfile vid to hdfspath vid`".



(b) The constituency-based parse tree of the logging text, "`no beanstalks defined for initialization`".

Figure 6.5: Two constituency-based parse trees for the logging texts. The no-terminal (i.e., syntactic tags) and terminal (i.e., tokens in logging text) symbols delimited by the black lines can be selected to construct the templates.

Approach.

To answer our research question, we propose LoGenText-Plus, which uses the pre-log code and the logging template as the source and AST as the context to generate the logging text. Unlike LoGenText, LoGenText-Plus not only extracts the three types of information (i.e., logging text, the pre-log code, and the context information) used in LoGenText but also considers the syntactic template of logging texts. As stated in Section 6.2.1.2, LoGenText-Plus contains two stages: (1) template generation and (2) template-based logging text generation.

Stage 1: template generation. LoGenText-Plus uses a Transformer-based Seq2Seq model to generate the templates for the given source input. To effectively incorporate the template and AST information, LoGenText-Plus considers concatenating the template from the logging text in the similar code with the pre-log code (i.e., the source input in LoGenText) as the new input to the source encoder, and AST as the context to the context encoder. Below, we describe how we extract the template and incorporate it into LoGenText-Plus.

Extracting the template from the logging text in the similar code. Similar to RQ3, in this step, we assume that similar code snippets would have similar logging templates, and incorporating the logging template in similar code would ease the process of predicting the target templates. For example, "`failed to unregister vid`" is a logging text from the project ActiveMQ, and the logging text in its similar code is "`failed to dispose of vid`". Both can have the same syntactic template "`failed TO VP`". In RQ3, we have extracted logging text from similar code based on the given pre-log code snippet. In this step, we first use *Stanza* [146], an open-source NLP library, to perform constituency parsing for each logging text from similar code. Constituency parsing is the task of analyzing phrase structures (e.g., simple declarative clauses, verb phrases, noun phrases, etc.) for a given sentence. Figure 6.5 shows two parsed trees, where the terminal symbols (or, leaf nodes in the tree) are tokens in the logging text, and non-terminal symbols are syntactic categories (e.g., "S" for simple declarative clause, "NP" for a noun phrase and "VP" for a verb phrase).

After having the consistency-based parse tree, we choose a certain depth of the constituency-based parse tree to construct the template. Then, all the symbols (including both terminal and non-terminal symbols) at the pre-defined depth are collected as the template. For example, assuming the depth is four, then the template for the logging text, "copying localfile vid to hdfspath vid", is "copying JJ NN IN NP". The depth ranges from one to the maximum depth of the generated tree (e.g., six for both examples in Figure 6.5), resulting in different templates for the logging text. Among all the templates, one special case is that we set the depth to a number larger than the maximum depth of the tree, the templates are exactly the same as the logging texts. With the decrease of the depth, the complexity of the templates (e.g., the length of the template and the number of unique tokens in the template) is also reduced.

In this research question, we start with the depth of one due to the following reasons (1) each template contains only one token and should be easier to predict, and (2) even though the template contains only one token, it still can convey different syntactic information (e.g., "S" for simple declarative clause and "NP" for noun phrase as shown in Figure 6.5). Note that this may be different from the text generation tasks (e.g., machine translation) in NLP, where the target text is usually a complete sentence (that is if we set the depth to one, the template may always be "S".). On the contrary, for the logging text in the source code, some developers prefer to use noun phrases while others may use complete sentences to monitor the status of the software. Moreover, the experiment in RQ5 also demonstrates that our choice is optimal for automatic logging text generation.

Concatenating the pre-log code and the template. In previous research questions (i.e., RQ1-RQ3), the source is the pre-log code. However, in this step, we consider concatenating the pre-log code and the template from the logging text in a similar code as the new source input to the source encoder of the Transformer-based model. We adopt this incorporation strategy due to the following considerations: (1) by doing the input concatenation, the newly formed input sequence is almost a complete code structure (i.e., the pre-log code and the template of the logging text from similar code). In other words, in the newly formed input sequence, the template acts as a placeholder, which needs to be refined or replaced by the output of the Transformer-based model. (2) Besides, under this setting, we can still integrate the AST information using our multi-encoder strategy which has been proven to be useful for generating the logging text [37].

Extracting the template from the target logging text. The goal of this stage is to predict the template based on the source input, thus, we need to construct a new target sequence, which is the template of each logging text. As we have already collected the logging text from each logging statement in Section 6.2.2, in this step, we share the same way of extracting the template from the logging text in a similar code to extract the template from the logging text.

By now, we have the source sequence (i.e., the concatenated pre-log code and the template from the logging text in similar code), target sequence (i.e., the template from the target logging text), and context information (i.e., AST information extracted in RQ2). Next, we describe how we integrate the context information into the model used for generating the template.

Integrating context information in our models. In RQ2, LoGenText utilizes another encoder to encode the context information. As shown in Figure 6.4, the outputs of the context encoder and source encoder are converted into a new representation by an attention layer, which is finally fused with the output of the source encoder by a gated sum. The input of the target decoder, G, is a deep hybrid of both the source and the context inputs. However, this design may have one limitation, that is the deep hybrid happens at the encoder part, as a result, the context information may not be passed into the decoder part effectively and the influence of the context may vanish after the attention and gated sum operations. Hence, in this step, we adopt another design to incorporate the context information at the decoder part [83, 203].

The structure of our new context integration approach is illustrated in Figure 6.6. Similar to the context encoder in the multi-encoder Transformer used in RQ2, the context encoder replicates the original encoder Transformer and takes the AST as the input. The output of the context encoder and the output of the source encoder are then passed to the target decoder separately, where the two outputs together with the previously generated template sequences are fed into self-attention layers (i.e., context attention layer and source attention layer as shown in Figure 6.6), respectively. Then, the outputs of the attention layers are fused by a gated sum. Formally, let S' be the output of the source attention

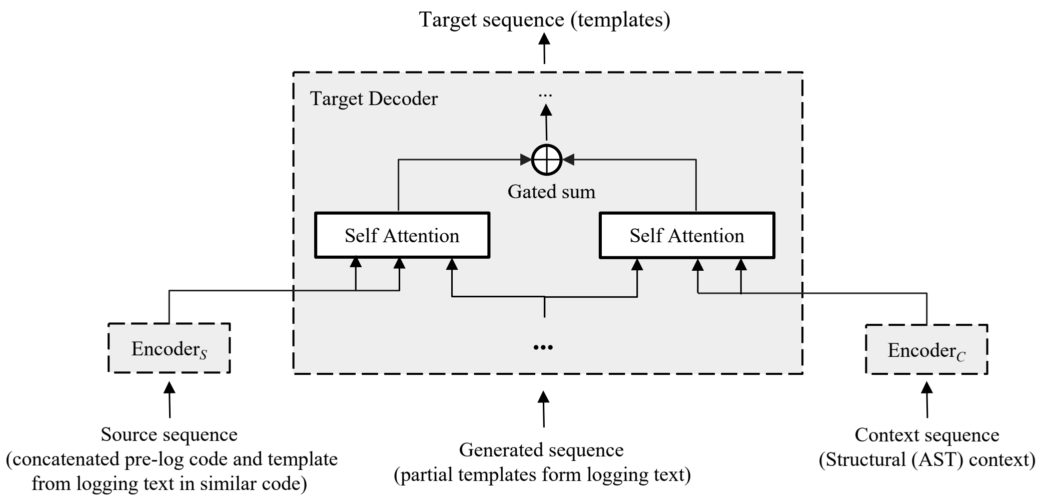

Target sequence (templates)
Target Decoder
...
Gated sum
Self Attention
Self Attention
...
Encoder$_S$
Encoder$_C$
Source sequence
(concatenated pre-log code and template
from logging text in similar code)
Generated sequence
(partial templates form logging text)
Context sequence
(Structural (AST) context)

Figure 6.6: An overview of the new multi-encoder Transformer used for template generation. The left "Self Attention" is the source attention layer and the right "Self Attention" is the context attention layer.

layer and C' be the output of the context attention layer, the output of the gated sum G is

$$G = \lambda \odot C' + (1 - \lambda) \odot S' \tag{6.10}$$

where the gating weight λ is calculated by

$$\lambda = \sigma \left(W \left[C', S' \right] + b \right) \tag{6.11}$$

where $\sigma(\cdot)$ is the sigmoid function, W is the weight parameters of the model and b is the bias.

Finally, the template generation stage produces a template for each of the newly constructed source input (i.e., Generated templates in Figure 6.2), which is later used for the template-based logging text generation (i.e., the model inference in Figure 6.2).

Stage 2: template-based logging text generation. LoGenText-Plus adopts the same model structure as the model used in template generation for template-based logging text generation. During the model training of logging text generation, LoGenText-Plus concatenates the template from the logging text in the similar code with the pre-log code (i.e., the source input in LoGenText) as the new input to the source encoder, and AST as the context to the context encoder, while the target sequence is the logging text instead of the template. During the model inference, LoGenText-Plus concatenates the pre-log code with the template produced in stage 1 as the new input to the source encoder. The output

Table 6.5: Evaluation results of using LoGenText-Plus, LoGenText and the baseline approach to generate logging texts in the studied projects (RQ4).

BLEU(%)	**ActiveMQ**	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
Baseline	21.0	19.8	26.0	37.9	30.1	19.6	19.5	28.2	21.5	34.1	25.8
Base LoGenText (RQ1)	23.0	22.8	25.4	39.0	34.6	21.8	23.1	28.0	24.9	28.8	27.1
LoGenText with AST (RQ2)	24.1	23.8	27.8	**41.8**	34.6	23.3	23.5	29.6	**28.8**	37.2	29.5
LoGenText-Plus	**26.6**	**25.5**	**31.2**	40.1	**35.0**	**23.8**	**23.7**	**30.3**	**28.8**	**37.9**	**30.3**

ROUGE-L(%)	**ActiveMQ**	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
Baseline	36.1	36.8	38.1	47.4	43.9	34.1	38.4	42.4	37.1	46.9	40.1
Base LoGenText (RQ1)	43.4	42.9	43.6	52.3	50.1	41.1	**46.5**	46.7	45.5	49.5	46.2
LoGenText with AST (RQ2)	42.5	43.4	44.0	53.9	**50.8**	42.1	46.4	**48.2**	**47.6**	53.6	47.3
LoGenText-Plus	**44.4**	**44.0**	**46.6**	**54.0**	50.1	**42.5**	46.2	47.6	47.3	**54.3**	**47.7**

Note: Values in bold font indicate the best-performing models.

is the final prediction of the logging text. Note that during model training, we intentionally use the template from the logging text in the similar code instead of the template from the corresponding logging text, because the former contains noise that can be used to simulate the prediction errors in the generated template. Otherwise, during model training, if we use the template from the corresponding logging text, the model would pay more attention to the template part. However, during inference, the generated template may contain errors, which would mislead the model, thus, resulting in a poor quality of the generated logging text. Our experiment results also confirm our assumption.

In order to understand the impact of the syntactic template information, similar to previous RQs, we evaluate the performance of LoGenText-Plus on all the subject projects, where we train separate models for each project (cf. Section 6.3.2.1).

Besides, we conduct another experiment to study how the diversity across different projects would impact our approach. We first train a single model using the combined ten training datasets and evaluate its performance on each of the ten individual projects. Furthermore, we extend the evaluation to include a new dataset from the project Cassandra, allowing us to assess the generalizability of our approach to unseen data. We select Cassandra as it is widely studied in the literature [94, 96, 109, 110, 200, 205].

We use the same metrics used in RQ1 (i.e., BLEU and ROUGE-L) to evaluate the quality of the generated logging texts.

Results.

Overall, our newly proposed approach LoGenText-Plus outperforms the baseline approach as well as the best-performing version of LoGenText that incorporates the AST context. The experimental results on the 10 studied projects are provided in Table 6.5 with the best results highlighted in **bold**. In particular, LoGenText-

Plus outperforms LoGenText in eight out of 10 projects in terms of BLEU score. For example, we obtain over 10% relative increase (i.e., 12.3%) for the project Brooklyn in terms of BLEU score and a 5.9% increase in ROUGE-L score. Our results indicate the effectiveness of incorporating templates in guiding the automated generation of logging text.

In addition to the overall BLEU and ROUGE-L, LoGenText-Plus can provide relatively good performance for relatively small projects. As shown in Table 6.5, LoGenText-Plus achieves a BLEU score of 25.5 to 37.9 for the ActiveMQ, Ambari, Brooklyn, and Synapse projects, which are the smallest projects with less than 500K SLOC. LoGenText-Plus has the largest BLEU improvements on three of these four smallest projects (i.e., 10.3%, 7.1%, 12.3% relative increases on projects ActiveMQ, Ambari, and Brooklyn respectively) over the best-performing form of LoGenText. It is widely recognized that deep neural networks usually require larger training data to generalize better [52, 91]. However, our results indicate that our LoGenText-Plus could alleviate the (negative) impact of limited training data and effectively generate logging texts for smaller projects.

However, we also observe that LoGenText-Plus may not always improve the performance significantly. For example, as Table 6.5 shows, the improvement of BLEU score on some relatively larger projects (e.g., HBase, Hadoop, and Ignite) is limited, and LoGenText-Plus performs even worse than LoGenText on the project Camel. This phenomenon may be explained from two aspects: (1) The size of the project: the project contains more lines of source code, which provides more source information and training sets for learning a good model, as a result, the impact of the template can be mitigated, and (2) The different context integration strategies: as we stated in RQ4-***Approach***, we adopt another integration strategy, where we move the context integration from the encoder part to the decoder, trying to enlarge the impact of the context. On the other hand, it should be noted that the context may contain noise, as a result, the impact of the noise is also increased, which makes the performance even worse. Thus, we further conduct another experiment on the 10 projects using the integration strategy proposed by LoGenText. The results are presented in Table 6.6. The results confirm that the integration strategy does have an impact on the performance of logging text generation. Specifically, we get a BLEU score of 42.3 on project Camel, which is higher than LoGenText-Plus and LoGenText (i.e., 40.1 and 41.8, respectively). This observation also matches previous studies that show that enlarging the context may lead to performance degradation of NMT models due to the noise introduced by the enlarged context [203, 207].

Additionally, we find that training separate models for each project (cf. Section 6.3.2.1) can benefit the performance of our approach. Table 6.7 shows the results of the different training strategies, as well as the performance of the model trained on the 10 studied

Table 6.6: Evaluation results of incorporating templates with different multi-encoder Transformers for logging text generation (RQ4).

	BLEU(%)										
	ActiveMQ	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
Integrating context at decoder	**26.6**	**25.5**	**31.2**	40.1	35.0	**23.8**	**23.7**	30.3	28.8	**37.9**	**30.3**
Integrating context at encoder	26.0	23.3	31.0	**42.3**	**35.1**	23.6	23.3	**30.6**	**29.4**	36.0	30.1
	ROUGE-L(%)										
	ActiveMQ	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
Integrating context at decoder	**44.4**	**44.0**	**46.6**	54.0	50.1	**42.5**	46.2	47.6	47.3	**54.3**	47.7
Integrating context at encoder	43.5	43.3	46.5	**54.5**	**50.7**	41.7	**46.4**	**48.8**	**48.2**	53.5	47.7

Note: Values in bold font indicate the best-performing models.

Table 6.7: Evaluation results of different training strategies for logging text generation in the studied projects and a new project (RQ4).

	BLEU(%)											
	ActiveMQ	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**	**Cassandra**
Separate models	**26.6**	**25.5**	**31.2**	**40.1**	**35.0**	**23.8**	**23.7**	**30.3**	**28.8**	**37.9**	**30.3**	**20.1**
Single model	22.1	21.9	27.2	36.9	32.8	20.9	22.0	27.5	26.5	34.1	27.2	11.3
	ROUGE-L(%)											
	ActiveMQ	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**	**Cassandra**
Separate models	**44.4**	**44.0**	**46.6**	**54.0**	**50.1**	**42.5**	**46.2**	**47.6**	**47.3**	**54.3**	**47.7**	**42.0**
Single model	41.5	42.1	42.9	50.1	49.1	40.2	45.4	46.4	46.5	51.5	45.6	31.2

Note: Values in bold font indicate the best-performing models. The single model of Cassandra is trained on the 10 studied projects.

projects, when applied to the new project, Cassandra. As the table shows, by training only one single model on all the training datasets, the performance of LoGenText-Plus decreases on almost all the projects and reaches an average BLEU score of 27.2, which is lower than that of the separate models (i.e., 30.3). The results may be due to the fact that the training data from other projects would bring some noise, and thus may negatively impact the performance of the model. Furthermore, when evaluating the single trained model on the unseen dataset (i.e., Cassandra), the model gives a BLEU score of 11.3. Meanwhile, we also evaluate the model that is trained on the project Casandra and the model has a BLEU score of 20.1. For comparison, we further evaluate the baseline approach under these two settings, which gives BLEU scores of 8.8 and 15.7 respectively. On the one hand, the results confirm the findings from previous work [59] that the language patterns in different projects vary a lot. On the other hand, the results reveal one of the limitations of our approach: although our approach has better performance than the baseline, there is still a non-negligible performance drop when the training and testing datasets are drawn from different distributions. The results call for future research that can alleviate such performance decreases across different distributions.

Although LoGenText-Plus exhibits improved performance compared to LoGenText, and both approaches surpass the baseline approach, it is important to acknowledge that there is still potential for further improvement in both approaches. To better understand the limitations and instances requiring enhancement, we conduct another manual analysis on instances where both LoGenText and LoGenText-Plus do not achieve satisfactory performance. By closely examining these cases, we aim to identify the specific challenges and factors contributing to suboptimal outputs.

In particular, we check the distribution of the BLEU scores for the generated logging texts and find that, on average, approximately 35 cases yield a BLEU or ROUGH score of zero. To gain further insights into these cases, we randomly selected 10 cases for each project, which we would thoroughly analyze.

We have categorized the characteristics of the cases into two main categories, each with several subcategories[8]:

Limited source input. Both LoGenText and LoGenText-Plus rely on the pre-log code as the source input, and in some cases, the pre-log code may lack sufficient information for logging text generation, resulting in unsatisfactory outputs. We have identified two common scenarios (1) The logging statement is put at the beginning of the method of which the method name is very simple and common and does not provide meaningful information. For instance, Figure 6.7a presents an example where the corresponding code (i.e., lines 2 and 3) only contains a few tokens and does not provide much information. 2) The logging statement describes the post-log code, while in our approaches, we use the preceding code as the input, which would result in the wrong output. As shown in Figure 6.7b, line 8 is the corresponding code that the logging statement (i.e., line 4) is describing, but our approaches ignore such information.

Similar source input in the training set. There are some test cases that share a very similar input with the training cases, where both approaches may simply copy the logging texts from the training cases. Specifically, (1) There are two consecutive logging statements in the original source code and one of them is used as the training case. For example, line 3 of Figure 6.7c appears in the training set and has the same source input as line 4 (i.e., the logging text to generate). (2) There are two logging statements that are close to each other in the original code. For example, in Figure 6.7d, the logging statement in the if block (i.e., line 4) is used as the training set, and when generating the logging statement in the else block (i.e, line 7), our approaches may simply copy the logging text from line 4, due to the minimal difference in source input.

Based on these findings, future work may consider (1) incorporating the data flow information to discriminate similar source input, (2) utilizing the method call graph to

[8]Note that these categories may not be strictly exclusive. For instance, in Example (a), the code (i.e., lines 2 and 3) is very common and there is a high possibility that it appears in the training set.

```
@Override
public void run() {
  try {
--------------Candidate log start---------------
Original log: LOG.debug("Executing task #{}", taskId)
Generated log-ast: logsearchfilenamerequestrunnable starting
Generated log-plus: persisting metric metadata
--------------Candidate log end---------------
...
```

(a)

```
...
try {
--------------Candidate log start ---------------
Original log: LOG.debug("Setting subscriptions: {}", ...)
Generated log-ast: dispose old state
Generated log-plus: disposition disconnect
--------------Candidate log end ---------------
connected.putSubscriptions(this.subscriptions);
...
```

(b)

```
private void printHelp() {
    ...
    logger.info("Default values:");

---------------Candidate log start----------------
Original log: logger.info(DOUBLE_INDENT + "HOST_OR_IP
                                    =" + DFLT_HOST)
Generated log-ast: default values
Generated log-plus: default values
---------------Candidate log end----------------
...
```

(c)

```
...
if (jobScheduler != null) {
    jobScheduler.removeJob(jobId);
    LOG.info("Removed scheduled Job " + jobId);
} else {
---------------Candidate log start----------------
Original log: LOG.warn("Scheduler not configured")
Generated log-ast: removed scheduled job vid
Generated log-plus: removed scheduled job vid
---------------Candidate log end----------------
...
```

(d)

Figure 6.7: An illustration of error cases generated by LoGenText and LoGenText-Plus.

identify more context for the logging text at the beginning of a method, and (3) identifying more relevant source code, while avoiding the introducing of noise.

Summary

LoGenText-Plus generally outperforms the baseline approach that leverages the existing logging texts in similar code snippets as well as the best-performing version of LoGenText that incorporates the AST context. Our results illustrate the effectiveness of using templates for guiding the generation of the logging text.

RQ5: How does the granularity of logging templates impact the performance of LoGenText-Plus in generating logging texts?

Motivation.

In RQ4, we have introduced the approach of how to build the templates from the consistency-based parse tree and have shown the effectiveness of using these templates. On one hand, we start building the templates with the depth of one, which produces simple yet effective templates. On the other hand, choosing different depths can result in diverse templates, which may convey different levels of information for guiding the generation of logging texts. For example, as shown in Figure 6.5b, if we set the depth to one, the template only has one symbol "NP", which means that the logging text is a noun phrase. The template is very short and cannot capture too much information. Meanwhile, if we set the depth to six (or a larger number), the template is exactly the same as the logging text, which violates our assumption. Therefore, in this research question, we aim to explore the impact of incorporating different templates derived from the tree with different depths on automated logging text generation.

Approach.

In this RQ, we adopt the same approach as in RQ4 to construct and incorporate the templates. In particular, we extract the templates with depths from one to nine and concatenate them with the pre-log code, separately. Based on the newly constructed source input, we train different models and evaluate the performance of LoGenText-Plus that incorporates different templates using the BLEU and ROUGE-L metrics.

Results.

The performance of LoGenText-Plus on the subject systems can be further improved by incorporating templates with different depths compared to that of LoGenText-Plus using templates with a depth of one. As shown in Table 6.8, we find that by incorporating the new templates constructed with different depths, the performance of LoGenText-Plus can be increased in half of the 10 studied projects (e.g., the average BLEU score increases from 29.5 (i.e., LoGenText with AST, cf. RQ2) to 30.9 (i.e., the best-performing depths)). The biggest improvement is observed in the project ActiveMQ. When setting the depth to two, LoGenText-Plus achieves a BLEU score of 28.9, which is 19.9% and 8.7% higher than LoGenText (i.e., 24.1, cf. RQ2) and LoGenText-Plus with the depth of one (i.e., 26.2, cf. RQ4). The results indicate that the templates constructed with different depths may capture various types of knowledge to help with automatic logging text generation. With proper depth, the template can further improve LoGenText-Plus for the studied projects.

However, similar to our findings in RQ2, incorporating templates with different depths can result in diverse performance for each project. Figure 6.8 shows the distribution of performance results produced by LoGenText-Plus with different templates. To quantify the variance of the differences, we also calculate the coefficient of variation (CV) for each project. The results show that the projects with a relatively small size (SLOC), are more sensitive to the templates. As Figure 6.8 shows, the four smallest projects, ActiveMQ, Ambari, Brooklyn, and Synapse projects have the largest variances, 5.0%, 4.0%, 5.2%, 4.7%, respectively. While for large projects, the impact of utilizing different templates is relatively weak. For example, for the project Synapse, using the template with a depth of six can only have a BLEU score of 33.4, compared to a BLEU score of 38.0 when using the

Table 6.8: Evaluation results of incorporating templates constructed with different depth in LoGenText-Plus for logging text generation (RQ5).

BLEU(%)											
Depth	**ActiveMQ**	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
1	26.6	**25.5**	31.2	40.1	**35.0**	**23.8**	23.7	**30.3**	28.8	37.9	**30.3**
2	**28.9**	24.4	**31.5**	**40.8**	34.2	22.9	24.2	29.0	28.9	36.4	30.1
3	28.2	24.6	29.3	40.6	34.5	22.1	24.1	29.1	**30.5**	37.4	30.0
4	24.8	23.4	28.2	40.2	**35.0**	21.9	24.3	29.0	29.2	35.0	29.1
5	25.9	23.2	30.5	39.6	33.2	22.3	22.7	28.8	28.1	**38.0**	29.2
6	26.4	23.2	29.0	**40.8**	34.0	21.9	23.7	28.4	27.7	33.4	28.9
7	25.6	23.9	30.4	39.3	33.8	21.8	24.2	28.9	27.6	34.6	29.0
8	26.1	22.2	27.6	39.6	33.7	21.2	23.3	28.8	28.5	34.1	28.5
9	27.8	23.6	27.3	38.9	33.2	21.5	**24.6**	29.0	26.9	35.7	28.8
Best	**28.9**	**25.5**	**31.5**	**40.8**	**35.0**	**23.8**	**24.6**	**30.3**	**30.5**	**38.0**	**30.9**
ROUGE(%)											
Depth	**ActiveMQ**	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**	**Avg.**
1	44.4	**44.0**	46.6	**54.0**	50.1	**42.5**	46.2	47.6	47.3	54.3	**47.7**
2	44.6	42.2	**46.7**	52.3	**50.8**	41.0	46.6	47.3	48.3	52.4	47.2
3	**44.8**	43.2	45.9	53.3	50.2	41.2	47.0	47.4	**48.6**	**55.2**	**47.7**
4	43.5	42.7	45.5	52.1	50.4	41.5	46.9	47.3	47.7	53.6	47.1
5	42.4	43.5	44.4	53.0	48.9	40.7	45.2	**48.0**	47.2	54.0	46.7
6	43.5	42.4	45.1	53.2	49.9	40.1	46.8	47.2	47.0	53.6	46.9
7	42.7	43.8	45.8	52.4	49.9	41.3	**47.7**	47.7	46.5	52.3	47.0
8	42.7	42.4	44.7	53.0	50.0	40.2	46.2	47.2	48.5	51.7	46.7
9	44.5	43.3	43.5	52.5	49.8	40.7	**47.7**	47.5	46.3	53.5	46.9
Best	**44.8**	**44.0**	**46.7**	**54.0**	**50.8**	**42.5**	**47.7**	**48.0**	**48.6**	**55.2**	**48.2**

Note: Values in bold font indicate the best-performing models.

template with a depth of five, but for the project Hive, the biggest performance different is 1.9 (i.e., 30.3 when depth is one vs. 28.4 when depth is six). One explanation for this phenomenon may be that larger projects may contain more source data, and thus, produce more powerful models and mitigate the differences between different templates.

In addition, as shown in Table 6.8, with the increases in depth, the average BLEU score tends to decrease. In particular, when the depth is one, LoGenText-Plus has the best BLEU and ROUGE scores on average. Besides, for most of the projects, LoGenText-Plus achieves the best performance when the depth is less than five. For example, the BLEU scores on projects ActiveMQ, Brooklyn and Camel reach the highest with a depth of three. This is reasonable given that with the increases in depth, more terminal symbols (tokens in logging text) are captured, resulting in more complex templates. Figure 6.9 shows the distribution of the length of templates under different depths as well as the vocabulary size (i.e., the number of unique tokens in templates). It is obvious that both the length of templates and the vocabulary size increase significantly with the increases in depth. As a result, the task of template generation becomes more difficult, which means

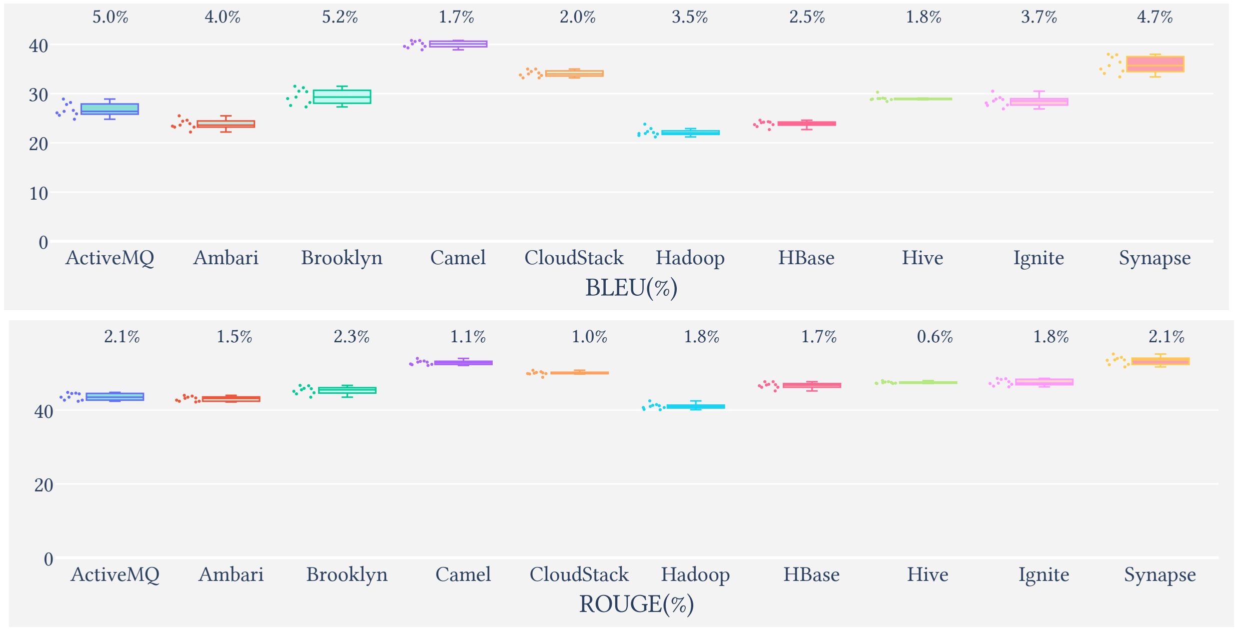


Figure 6.8: The distribution of the results produced by LoGenText-Plus with different templates. The horizontal axis represents all the studied projects; the vertical axis is the performance (i.e., BLEU or ROUGE) in different projects. The numbers on top of each box are the corresponding coefficient of variance.

that LoGenText-Plus may not generate accurate templates, and thus negatively impacting the performance of generating the logging texts.

Discussion

In the above sections, we have quantitatively demonstrated the superiority of LoGenText-Plus on the 10 subjects. Based on the extensive experimental results, in this part, we would like to elaborate more on the design choice as well as the potential limitations of LoGenText-Plus.

Strengths and limitations

In LoGenText-Plus, we divide the logging text generation task into two stages: template generation and template-based logging text generation. Therefore, the advantages of LoGenText-Plus can be discussed from two aspects:

- Design paradigm: Instead of directly predicting the logging texts, we are trying to solve the problem with a coarse-to-fine strategy. Intuitively, predicting the templates is a little easier compared to predicting the logging texts directly, as (1) the size of

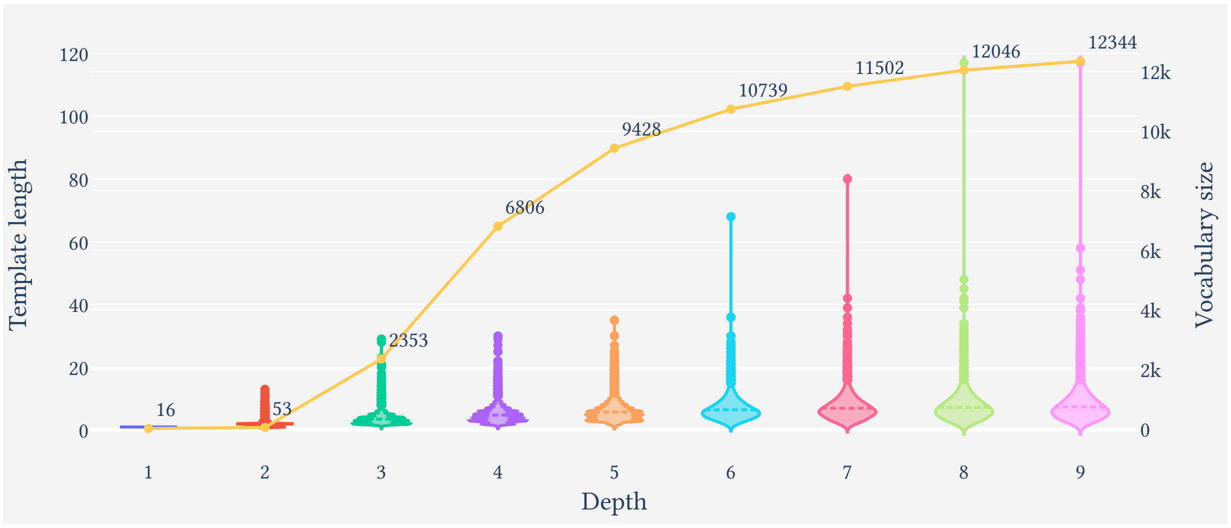


Figure 6.9: The distribution of the length of templates (i.e., violin plot) constructed with different depths and the vocabulary size (i.e., line plot). The horizontal axis represents the different depths; the left vertical axis is the template length (i.e., the number of tokens in each template); the right vertical axis is the vocabulary size (i.e., the number of distinct tokens in all the templates).

the symbols (about 60 tags) in the templates is much smaller than the number of tokens (i.e., natural language words plus source code tokens) in the logging texts, and (2) the templates are relatively shorter than the corresponding logging texts. In particular, if the depth is set to one, the template only contains one element. As a result, we are actually converting the template generation task to a multi-class classification problem, which should be easier to tackle.

- Enriched information: By incorporating the templates of the logging texts, we are explicitly introducing more knowledge into the model. Previous works [37, 59] mainly focus on the information extracted from the source code, such as the abstract syntax trees, and code sequences, while few works consider utilizing the information from the target sequence. In this work, the template is constructed from the constituency-based parse tree of the logging text which provides a syntactical representation of the logging texts. The low-level syntactic information (i.e., syntactic tags) with the high-level semantic information (i.e., some tokens in the logging text) in the template complements the source code and thus may be useful for the generation of the logging texts. This finding also explains the experimental results in RQ4 that LoGenText-Plus can provide relatively good performance for relatively small projects, as LoGenText-Plus can bring more external knowledge to the trained model.

However, LoGenText-Plus also has limitations. Although using the template may provide useful information, there is still a chance that the generated templates contain noise, and thus may harm the performance of LoGenText-Plus. As the experimental results in RQ5 show, with the increases in depths, predicting the templates tends to be a more challenging task. As a result, the generated templates may be of poor quality with a risk of misleading the generation process of the logging texts. Moreover, we also observe that for some projects, LoGenText-Plus does not bring significant benefits, especially when the size of the projects is large enough. For example, LoGenText-Plus has the same BLEU score on the Ignite project with LoGenText. This may be due to the fact that larger datasets may produce more powerful models and mitigate the differences caused by the external information (i.e., syntactic information from the logging texts.). In future work, to further improve the performance of LoGenText-Plus, we can focus on generating more accurate templates.

Another limitation may come from the fact that we use the pre-log code as the source input, while there may exist some logging texts describing the succeeding source code. Due to the lack of information on the corresponding source code, our approach may fall short of generating satisfactory logging texts. In this work, the design choice of only considering the pre-log code as the main source input is based on the findings from previous work by He et al. [59], where they find that using the code surrounding (i.e., code preceding and succeeding) the logging text would result in worse results, compared to only using the code preceding the logging text.

We also conduct another two experiments on LoGenText and LoGenText-Plus, where we concatenate the pre-log code and post-log code as the source input and utilize the AST as the context. We find that both LoGenText and LoGenText-Plus have a performance degradation, with an average BLEU score of 27.6 and 26.7 respectively, compared to the initial BLEU scores of 30.1 and 30.3 without the use of post-log code. The results show that incorporating more contexts can not always guarantee a better model, which conforms to the results in the work of He et al. [59]. This phenomenon can be explained by the fact that developers prefer to insert logging statements to describe the actions in the preceding source code [40, 59], and thus, the use of post-log code may bring in some noise. Especially for LoGenText-Plus, as it involves the use of post-log code in two stages: template generation and logging text generation. The errors in the generated template may propagate to a later stage, and mislead the generation of the logging texts. Meanwhile, there is a possibility that the use of all the pre-log code might not be optimal, as the logging statements may only describe a few lines of code (e.g., the example in our introduction section). Future work may consider improving the performance by identifying the most relevant source code as input.

Summary

The performance of LoGenText-Plus on the subject systems can be further improved by incorporating different templates with different depths. However, the selection of the templates is a trade-off, as incorporating templates with different depths can result in diverse performance for each project.

RQ6: Can we harmonize the wording in the generated logging text with the logging text written by developers using n-gram dictionaries?

Motivation.

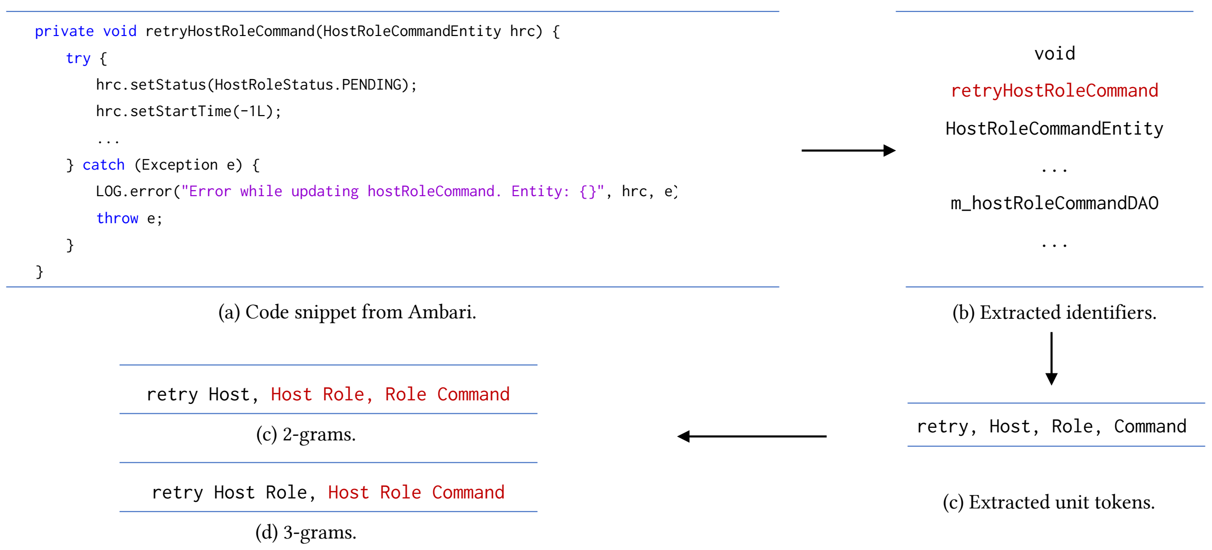
Prior work [37] proposes an NMT-based model to automatically generate the logging text based on the source code (i.e., LoGenText) and achieves promising results. However, while reviewing the generated logging texts, they find that the generated text sequence and the text sequence in the developer-written logging text may not always be consistent. For example, the logging text in Figure 6.5a uses the term "localfile", while the generated logging text uses the noun phrase "local file". Although these two terms have very similar meanings for developers, the use of different words may cause inconsistency in the wording and affect downstream log-related tasks (e.g., log parsing, and log compression). Meanwhile, it is obvious that the term "localfile" should be an identifier from source code (e.g., method names, or variables) and is constructed by two natural language tokens ("local" and "file"). Therefore, in this research question, we aim to explore whether we can refine the wording and make it consistent with the original logging text written by developers based on the extracted identifiers in the source code.

Approach.

In this RQ, we first build n-gram dictionaries from the source code and then extract the token sequences of a certain length (i.e., n-grams) from the generated logging text. Then, we check whether we can compose the extracted sequence into bigger units based on the dictionary. We here choose the n-gram model, as it has been proven to be useful for processing log data [32].

Building n-gram dictionaries from the source code. *Pre-processing source code.* In this step, we try to extract identifiers from the source code. We first apply srcML[9] to the method that contains the logging text. srcML converts the source code into an XML tree, where the leaf nodes are the tokens in the source code. We then use Beautiful Soup to select the nodes with a "name" tag, which is used to mark the identifiers of the source code. For example, Figure 6.10a shows a code snippet from the project Ambari and the extracted

[9]https://www.srcml.org/



```
private void retryHostRoleCommand(HostRoleCommandEntity hrc) {
    try {
        hrc.setStatus(HostRoleStatus.PENDING);
        hrc.setStartTime(-1L);
        ...
    } catch (Exception e) {
        LOG.error("Error while updating hostRoleCommand. Entity: {}", hrc, e)
        throw e;
    }
}
```

(a) Code snippet from Ambari.

```
void
retryHostRoleCommand
HostRoleCommandEntity
...
m_hostRoleCommandDAO
...
```

(b) Extracted identifiers.

```
retry, Host, Role, Command
```

(c) Extracted unit tokens.

```
retry Host, Host Role, Role Command
```

(c) 2-grams.

```
retry Host Role, Host Role Command
```

(d) 3-grams.

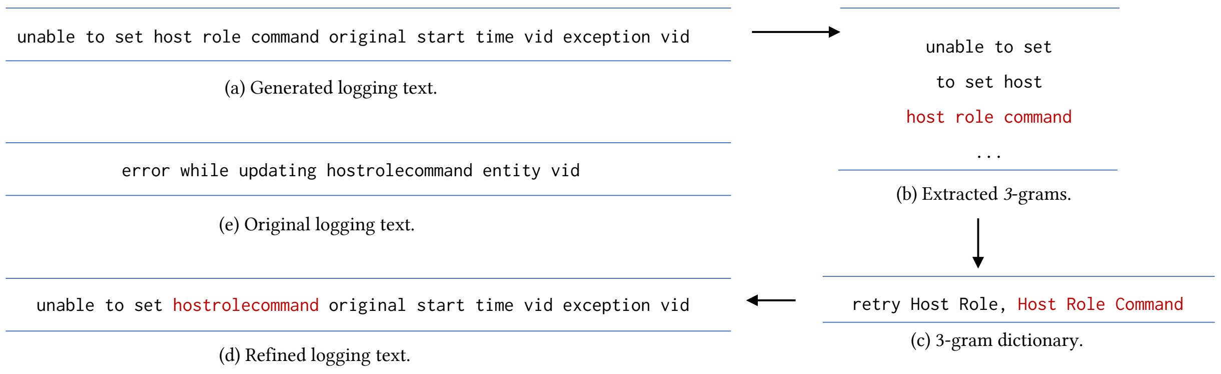
Figure 6.10: An overview of the process of building n-gram dictionaries from the source code.

identifiers are shown in Figure 6.10b. Then, we tokenize the identifiers based on the camel case convention. Figure 6.10c is the tokenized token units of "retryHostRoleCommand", which are later used for building the dictionaries.

Building n-gram dictionaries from tokens. In this step, we build n-gram dictionaries based on the extracted token units. In our approach, an n-gram is a contiguous subsequence of n token units from a tokenized identifier. For example, given the sequence of token units "retry, Host, Role, Command", we can build a dictionary with three 2-grams or a dictionary with two 3-grams, as shown in Figure 6.10. We build such n-gram dictionaries for each method that contains the logging text.

Composing n-grams in the generated logging text together into bigger units. *Identifying n-grams in generated logging text.* Similar to the last step, we extract n-grams for each generated logging text. For each n-gram from the logging text, we check whether it appears in the pre-constructed dictionary in the last step. If the n-gram is found in the dictionary, we consider it as an n-gram that may be combined into one new token. We filter out the n-grams that never appear in the dictionaries. Figure 6.11a is the generated logging text based on the given source code in Figure 6.10a. Based on the dictionary built from the source code (i.e., Figure 6.11c and Figure 6.10d), we filter out all the 3-grams except the "host role command".

Composing identified possible n-grams. From the last step, we have obtained a list of n-grams that may be combined into new tokens. However, not all the n-grams are meaningful



```
unable to set host role command original start time vid exception vid
```

(a) Generated logging text.

```
unable to set
to set host
host role command
...
```

(b) Extracted 3-grams.

```
retry Host Role, Host Role Command
```

(c) 3-gram dictionary.

```
unable to set hostrolecommand original start time vid exception vid
```

(d) Refined logging text.

```
error while updating hostrolecommand entity vid
```

(e) Original logging text.

Figure 6.11: An overview of the process of composing n-grams in the generated logging text together into bigger units.

or frequently used in the logging text. If we simply combine all these identified n-grams, we may have a lot of False Positives, especially when n is small (e.g., 2). For example, in Figure 6.11, if we use a 2-gram dictionary (i.e., Figure 6.10c), we may combine "host role" or "role command", which results in worse wording in the generated logging text. To avoid such improper combinations, we define two rules (1) syntactic analysis, and (2) logging practice. The details are shown below:

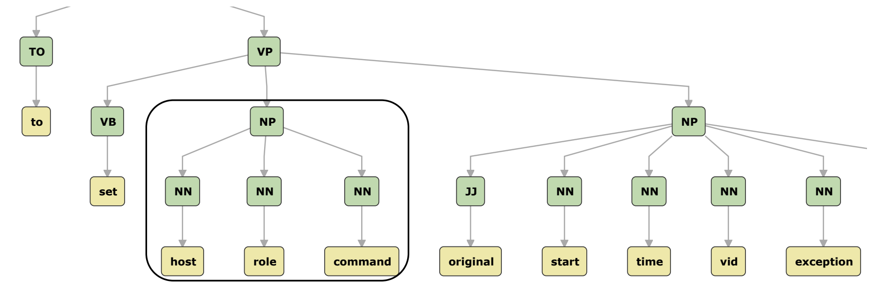


Figure 6.12: The constituency-based parse tree of the logging text in Figure 6.11a. The tokens delimited by the black lines are three siblings and the only children of the node "NP".

- Syntactic analysis-based rule (rule 1): The n-gram to be combined should be the only siblings of the same parent (e.g., "NP", "VP") in the consistency-based parse tree. Figure 6.12 illustrates the constraint that "host, role, command" are siblings (i.e., three nouns) with the same parent node (i.e., "NP"), and their parent only has these three children. Thus, they can be used together to compose a bigger token unit.

- Logging practice-based rule (rule 2): Meanwhile, some developers prefer to use separate tokens of identifiers in the logging text. Therefore, to make the combined new tokens conform to the existing logging text conventions, we propose our second rule: for a 2-gram, the newly composed bigger token unit in the existing logging texts (i.e. the training corpus) should be more frequently used than that of the 2-gram (i.e., the number of occurrences of the newly composed bigger token unit should be larger than that of a 2-gram); for 3-grams, the newly combined token should appear at least once in the existing logging text.

Note that in our experiment, to balance the recall and precision, we further adjust the scope of the rules, that is we limit the identifiers for building dictionaries to the method name.

To examine the effect of our dictionary-based combination strategy, we first compare the generated logging texts and the logging texts written by developers, and then we manually study the results.

Results.

Although we can refine the generated logging text with the post-processing strategy, the improvement is limited. We first manually check the generated logging texts and compare them with the original logging texts extracted from the source code. For a generated logging text, if it contains a combined n-gram that can be found in the original logging texts, then it is considered a ground truth. We find there do exist n-grams in the generated logging text that can be possibly combined, but the number is relatively small (i.e., Ground truth in Table 6.9). As shown in Table 6.9, the number of ground truth is small, which means that there are a few generated logging texts containing a n-gram that should be combined to harmonize the wording (e.g., the generated logging text in Figure 6.11a).

To examine the impact of our proposed two rules, we have conducted another three ablation experiments. First, we remove the two rules and combine the n-grams only based on the dictionary (i.e., None in Table 6.9). As Table 6.9 shows, a much larger size of the generated logging texts is detected, while a small number of the detected logging texts are relevant to the ground truth (i.e., True Positives). To filter out the irrelevant logging texts (i.e., False Positives), we apply the two rules to the detected logging texts. As a result, we successfully detect two logging texts that can be further refined. The results show that by using our proposed post-processing strategy we can further improve the generated logging texts, but the improvement is incremental. Future research may consider modifying the architecture of the Transformer-based model and incorporating the n-gram dictionary into the model (e.g., N-Grammer [152]) during the training or inference stage to improve the quality of the generated logging texts.

Table 6.9: Evaluation results of applying different constraints for composing n-grams in the generated logging text together into bigger units (RQ6). **Ground truth** represents the number of the generated logging texts containing n-grams that should be combined. **Detected** is the number of the detected logging texts by our strategy. **Relevant** represents the number of the detected ground truth. **Both, None, Rule 1, and Rule 2** represent when applying both constraints, no constraint, the first constraint and the second constraint respectively.

Rules		**ActiveMQ**	**Ambari**	**Brooklyn**	**Camel**	**CloudStack**	**Hadoop**	**HBase**	**Hive**	**Ignite**	**Synapse**
	Ground truth	1	4	0	4	6	6	4	2	2	2
Both	Detected	1	1	0	0	0	3	2	1	0	1
	Relevant	1	1	0	0	0	0	0	0	0	0
None	Detected	12	24	21	48	165	82	30	30	25	8
	Relevant	1	2	0	1	4	4	2	1	1	0
Rule 1	Detected	5	13	9	16	96	42	20	16	7	4
	Relevant	1	1	0	0	0	2	1	1	1	0
Rule 2	Detected	1	1	0	0	3	5	2	1	0	1
	Relevant	1	1	0	0	0	1	0	0	0	0

Finally, we manually study the detection results and identify two possible reasons for incorrectly detected logging texts: (1) Inconsistent writing convention. For example, although this research question is motivated by the logging text in Figure 6.14, we still fail to refine this generated logging text by combining the 2-gram "local file". We check the method and find both "localfile" and "local file" are used in the logging statements of this method, but "localfile" only appears once in this extracted logging text, thus the combination operation is ignored. (2) Missed n-grams in the dictionary. This is reasonable, as we limit the identifier to the method name for building the dictionaries, we have the chance to miss some n-grams. However, selecting the identifiers is a trade-off. If we chose more identifiers, as a result, we would produce more false positives. On the contrary, fewer identifiers would cause the miss of the possible combinations.

Summary

It is possible to harmonize the wording in the generated logging text with the post-processing strategy that leverages the token sequence (i.e., n-grams) in the source code, however, the improvement is limited. Future research may consider incorporating the obtained n-grams from the source code into the logging text generation model during the training or inference stage, to improve the quality of the generated logging texts.

6.5 Human Evaluation

Our approach LoGenText and its extension LoGenText-Plus are evaluated in the last section based on quantitative metrics (i.e., BLEU and ROUGE scores) that measure the similarity between the original and the generated logging texts. However, the quantitative metrics may not directly reflect how developers perceive the quality of the generated logging texts. Therefore, in this section, we conduct two separate human evaluations to further evaluate LoGenText and LoGenText-Plus.

6.5.1 Evaluation of LoGenText

We invited 42 participants in our human evaluation. The participants include a mix of 23 graduate students who major in computer science or software engineering and 19 full-time employees (e.g., software developers, software researchers, and data analysts) in software-related companies across the globe. These companies include multimedia, video games, social media, telecommunications and finance. 10 participants have at least five years of experience in software development and 32 participants have around 10 years of experience.

Our human evaluation for LoGenText contains two tasks: **task 1)** evaluating the *similarity* between the automatically generated logging texts and the original logging texts extracted from source code; **task 2)** evaluating the logging texts separately from three aspects [189], i.e., *relevance*, *usefulness* and *adequacy* based on the given source code. For task 1, each participant was given 15 logging statements that were randomly sampled from the 10 projects to evaluate. We presented the participants with the original logging texts, the logging texts generated by the baseline, and the logging texts generated by LoGenText. Since our results in Section 6.4 show that the context-aware form of LoGenText incorporating the AST context has the best overall performance, we used it to generate logging texts for our human evaluation. We named the logging text from the original logging statement as *log-ref* and the two generated logging texts as *log-1* and *log-2*. We asked the participants to rate the similarity between the generated logging texts (*log-1* and *log-2*) and the original logging texts (*log-ref*). In order to avoid the bias caused by the order of the two generated logging texts, we randomly assigned the one generated by LoGenText or by the baseline as *log-1* or *log-2*. Each generated logging text is evaluated based on a scale from 0 to 4 where 0 means no similarity and 4 means perfect similarity. For task 2, each participant was randomly given three logging statements to evaluate. We presented each participant with the original logging text, the logging text generated by the baseline, the logging text generated by LoGenText, and the surrounding method of the logging statement that highlights the location of the logging statement. We randomly assigned the three logging texts as *log-a*, *log-b* and *log-c*. We asked the participant to rate the three logging texts based on the given code snippet from three aspects, i.e., *relevance*, *usefulness* and *adequacy*. *Relevance* refers to how relevant the logging text is to the given source code. *Usefulness* refers to how useful the logging text is for collecting valuable run-time information of the source code. *Adequacy* refers to how the logging text is acceptable in quality or quantity with regard to the given source code. Each logging text is evaluated based on a scale from 0 to 4 where 0 means irrelevant/useless/unacceptable and 4 means perfect relevance/usefulness/adequacy.

LoGenText generates logging texts that are significantly more similar to the original logging texts than that generated by the baseline approach. Figure 6.13 presents the distribution of the user ratings in our evaluation. We find that LoGenText generates more logging texts with ratings of 3 and 4 while fewer logging texts with ratings of 0 and 1 than the baseline approach. We conducted a Wilcoxon signed-rank test [186]

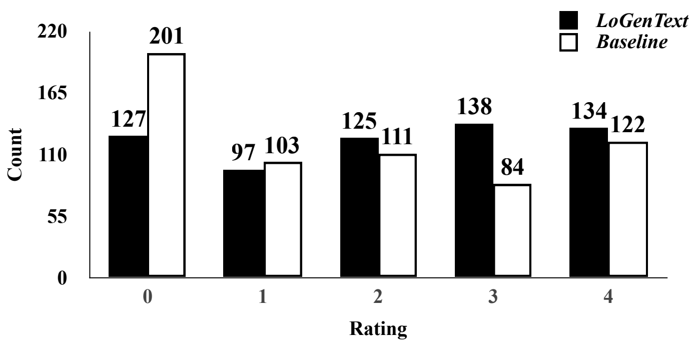


Figure 6.13: Distribution of the rating results (in task 1) in terms of the similarity between the generated logging texts and the reference logging texts.

to statistically compare the ratings of the logging texts generated by LoGenText and the baseline approach. With a p-value $\ll 0.00001$[10], we can confirm that the difference between the ratings of the logging texts generated by the two approaches is statistically significant. On the other hand, despite the significant improvement over the baseline approach, we still observe that more than one-third of the automatically generated logging texts by LoGenText receive a rating of 0 or 1. The results suggest opportunities for future research that further improves the automated logging generation.

In order to reflect on the results of our research questions that leverage quantitative metrics BLEU and ROUGE to evaluate LoGenText (cf., Section 6.4), we analyze the relationship between the results of the quantitative measurement and the human evaluation.

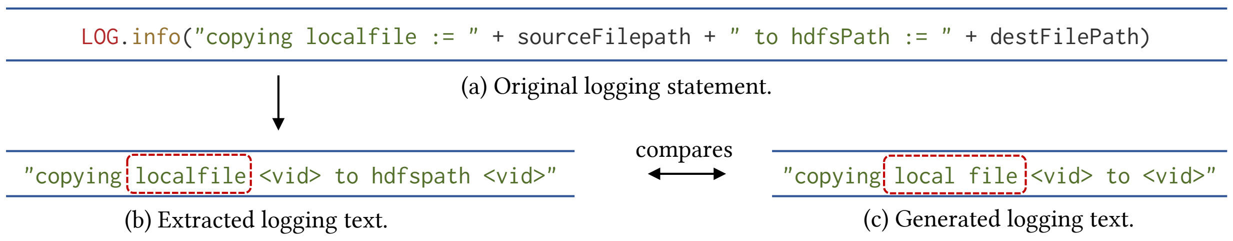
[10]We also conducted the Vargha and Delaney A index, due to the significant overlap between the ratings, the effect size is negligible.

Specifically, we group the logging texts generated by LoGenText by each rate, then evaluate the BLEU and ROUGE scores of the logging texts in each group. As shown in Table 6.10, higher BLEU and ROUGE scores are both associated with higher user ratings. Such results confirm the validity of our findings in our research questions that leverage the quantitative metrics.

Table 6.10: Comparing the human ratings (in task 1) and the BLEU and ROUGE scores of the logging texts generated by LoGenText.

Rating	**0**	**1**	**2**	**3**	**4**
BLEU	14.3	20.6	27.4	36.4	78.5
ROUGE-L	21.4	29.7	37.4	46.4	87.3

We manually examine the generated logging texts for which the participants assigned a very high rating (i.e., 3 or 4) while the BLEU and ROUGE values are relatively low (i.e., lower than the median), in order to further understand the quality of the generated logging texts. In particular, there are 79 (12.5%) cases where the human ratings are high (i.e., 3 or 4) while the BLEU scores are lower than the median. We find two main reasons contributing to such inconsistency: (1) **Using shorter words.** In the generated logging texts, the generated words are often short and easy to follow. For example, in a logging statement from Ambari,



```
LOG.info("copying localfile := " + sourceFilepath + " to hdfsPath := " + destFilePath)
```

(a) Original logging statement.

```
"copying localfile <vid> to hdfspath <vid>"
```

(b) Extracted logging text.

```
"copying local file <vid> to <vid>"
```

(c) Generated logging text.

Figure 6.14: An example generated logging text from Ambari.

the original logging text uses the term "*localfile*"; while our generated logging text uses the term "*local file*". Although these two terms have a very low similarity in terms of BLEU and ROUGE, they have a very similar meaning. (2) **Using synonyms.** Another reason for the inconsistency is the use of synonyms. For example, a logging text from Hadoop says "*no beanstalks defined*" while our generated logging text says "*no beanstalk definitions found*". Both logging texts have similar meanings but with different choices of words, which results in a high human rating but low BLEU and ROUGE-L values.

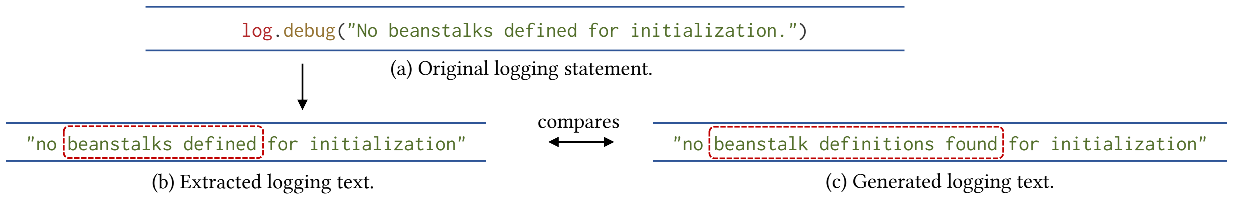

log.debug("No beanstalks defined for initialization.")
(a) Original logging statement.
"no beanstalks defined for initialization"
(b) Extracted logging text.
compares
"no beanstalk definitions found for initialization"
(c) Generated logging text.

Figure 6.15: An example generated logging text from Hadoop.

LoGenText outperforms the baseline approach in all three aspects. Table 6.11 shows the mean and median of relevance, usefulness and adequacy scores of the reference logging texts and the logging texts generated by LoGenText and the baseline approach. We can see that LoGenText outperforms the baseline approach on all three aspects with an average score of 2.67, 2.41 and 2.15, respectively. Similar to task 1, we also conducted a Wilcoxon signed-rank test and the difference is statistically significant for each aspect. However, there is still a non-negligible margin between the logging texts generated by LoGenText and the reference logging texts. The results call for future research that narrows down the gap between the logging texts written by developers and the automatically generated logging texts. On the other hand, the mean scores of the reference logging texts are 3.37, 3.19 and 3.02 respectively, which indicate that some logging texts inserted by the developers can still be further improved and call for high-quality logging texts to record the software execution information.

Table 6.11: Comparing the mean and median ratings of the logging texts in task 2. The median ratings are in the brackets following the mean ratings.

	Relevance	**Usefulness**	**Adequacy**
Reference	3.37 (4)	3.19 (4)	3.02 (3)
Baseline	2.09 (2)	1.89 (2)	1.75 (2)
LoGenText	**2.67 (3)*****	**2.41 (3)*****	**2.15 (2)****

Note: ***: p-value$<$0.001; **: 0.001$<$p-value$<$0.01. The effect size is small for three aspects.

Summary

The logging texts generated by LoGenText have a higher quality than that generated by the baseline approach in terms of relevance, usefulness, adequacy, and similarity to the logging texts written by developers. Our results also suggest future research opportunities for improving automated logging generation.

6.5.2 Evaluation of LoGenText-Plus

In the last section, we have conducted a human evaluation to compare LoGenText and the baseline and the results show that LoGenText outperforms the baseline approach in all aspects. Therefore, in this section, we conduct another human evaluation to compare baseline and LoGenText with LoGenText-Plus. Considering the number of cases to evaluate, we invited 10 out of the 42 participants and nine of them are from academia and one from industry.

Our human evaluation for LoGenText-Plus contains two tasks: **task 1)** comparing the quality of the logging texts generated by LoGenText with that of LoGenText-Plus based on the given source code, and **task 2)** comparing the quality of the logging texts generated by the baseline approach with that of LoGenText-Plus. To be consistent with the evaluation for LoGenText, we use the same dataset as that of task 2 in Section 6.5.1. In order to avoid redundancy, we filter the dataset to remove cases where LoGenText-Plus and LoGenText as well as the baseline approach, generate the identical logging texts. As a result, approximately half of the samples are filtered out, leaving us with 69 out of 126 samples for task 1 and 96 out of 126 samples for task 2. Similar to the methodology used in task 2 in Section 6.5.1, each participant was presented with the two generated logging texts as *log-1* and *log-2* and the surrounding method of the logging statement that highlights the location of the logging statement. Note that the names *log-1* and *log-2* were randomly assigned to avoid bias. Then each participant was asked to examine whether *log-1* is better than *log-2* based on the given code snippet. We listed three options for each comparison, *TRUE*, *FALSE*, and *NA*. *TRUE* means that *log-1* is better than *log-2*, *FALSE* means that *log-2* is better, and *NA* means the two logging texts are hard to compare (e.g., both are similar or useless). Besides, each participant was asked to provide reasons why they made the decision.

Overall, LoGenText-Plus generates better logging texts compared to that generated by LoGenText and the baseline approach. Figure 6.16 presents the distribution of the user ratings in our evaluation, where "Neutral" means that the two generated logging texts are hard to compare. We find that LoGenText-Plus generates more logging texts that are better than LoGenText (i.e., 53.6% vs. 34.8%) and the baseline approach (i.e., 57.3% vs. 28.1%), which shows the improvement of LoGenText-Plus over LoGenText and the baseline. However, we still observe that around 30% of the automatically generated logging texts by LoGenText and the baseline approach receive a better rating. The results suggest opportunities for future research to further improve LoGenText-Plus.

Besides, to uncover the reasons why LoGenText-Plus or LoGenText receives a higher rating, we further manually examine the comments provided by participants as well as the

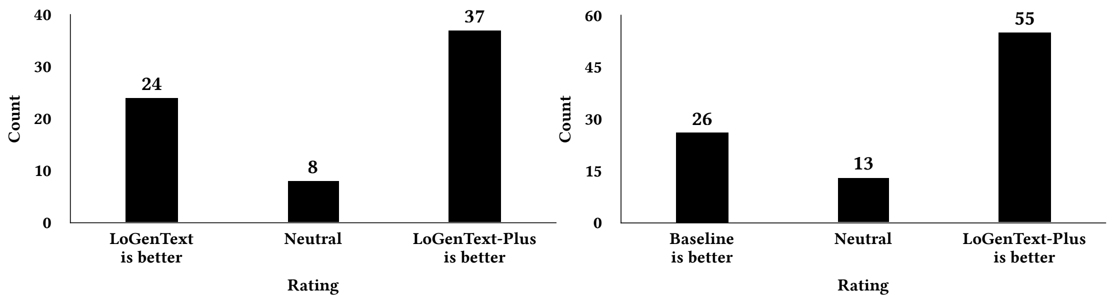


Figure 6.16: Comparing the human ratings of the logging texts generated by LoGenText-Plus, LoGenText, and the baseline approach. "Neutral" means that the two generated logging texts are hard to compare.

generated logging texts. In other words, the goal of this step is to find out what kind of logging texts (i.e., the characteristics) are considered with a higher quality by developers, aiming to provide guidance for writing or generating good logging texts.

We summarize five main reasons that a developer may favor a logging text. The reasons together with examples are presented in Table 6.12. We discuss each reason in detail in the rest of this section.

More relevant to the source code. The logging text should be more relevant to the source code, which means that the actions that happened in the source code should be similar or the same as the described actions in the logging text. For example, as shown in Table 6.12a, the method is about the action "shut down", while the generated log-2 is describing the action of "connection", which is irrelevant to the source code, thus, log-1 is selected as a better logging text.

More descriptive and useful information to the source code. We find that some logging texts are very short and simple, and thus cannot detailly describe what is happening in the software system. For example, as shown in Table 6.12b, the generated log-1 provides more information, as it not only tells the action taken (i.e., "load") before the logging statement but also shows the result of this action (i.e., an exception occurs).

More succinct and less confusing/redundant words. On one hand, the logging text should provide enough information for failure diagnosis. On the other hand, the logging text should also be more succinct and avoid confusing or redundant descriptions. For example, in the generated logging text log-2 in Table 6.12c, the last few words are a little meaningless and may confuse developers while analyzing the logs. Besides, our approach can insert "<vid>"s to the generated logging text as variable placeholders. However, one participant commented that some "<vid>"s may be unnecessary.

Table 6.12: The summary of the five main reasons that a developer may favor a logging text. The "Generated log-1/2" represents two logging texts by either LoGenText-Plus or LoGenText. "Original log" denotes the logging statement written by developers. The better logging text is highlighted in bold green.

Reason	Example
More relevant to the source code	`public void shutdown() {`
	`---------------Candidate log start----------------`
	`Generated log-1: shutting down connection to zk <vid>`
	`Generated log-2: connection to zookeeper <vid>`
	`Original log: LOG.debug("CamelDestination shutdown()")`
	`---------------Candidate log end----------------`
	`...`
	(a) CamelDestination.java from Camel
More descriptive and useful information to the source code	`...`
	`load(key.file(), props);`
	`} catch (IOException e) {`
	`---------------Candidate log start----------------`
	`Generated log-1: load of <vid> failure exception <vid>`
	`Generated log-2: load of <vid>`
	`Original log: LOG.error("Failed to load: " + key + ", reason:" + e.getLocalizedMessage())`
	`---------------Candidate log end----------------`
	`...`
	(b) ReloadableProperties.java from ActiveMQ
More succinct and less confusing/redundant words	`...`
	`if(filterPart.size() != 2) {`
	`---------------Candidate log start----------------`
	`Generated log-1: invalid filter specification <vid> skipping`
	`Generated log-2: invalid filter specification filters count <vid> skipping split s`
	`Original log: LOG.warn("Invalid filter specification " + filterClass + " - skipping")`
	`---------------Candidate log end----------------`
	`} else {`
	`...`
	(c) ThriftServer.java from HBase
More accurate	`...`
	`try {`
	`    newSession.close(false);`
	`} catch (Exception ex) {`
	`---------------Candidate log start----------------`
	`Generated log-1: failed to close an old session ignoring`
	`Generated log-2: failed to close session <vid>`
	`Original log: LOG.error("Failed to close an unneeded session", ex)`
	`---------------Candidate log end----------------`
	`...`
	(d) TezSessionPool.java from Hive
More specific and focusing on critical actions in source code	`public void deleteAllMessages() throws IOException {`
	`...`
	`    getAdapter().doDropTables(c);`
	`    getAdapter().setUseExternalMessageReferences(isUseExternalMessageReferences());`
	`    getAdapter().doCreateTables(c);`
	`---------------Candidate log start----------------`
	`Generated log-1: deleted apache activemq <vid>`
	`Generated log-2: executing sql <vid>`
	`Original log: LOG.info("Persistence store purged.")`
	`---------------Candidate log end----------------`
	`...`
	(e) JDBCPersistenceAdapter.java from ActiveMQ

More accurate. Another important factor for a better logging text is to use more accurate descriptions. The logging text should avoid providing the wrong information, which may mislead the developers. For example, as shown in Table 6.12d, the generated log-1 uses “an old” to describe the object “session”. However, based on the source code in the “try” block, the “session” should be a “new” session, instead of the “old”.

More specific and focusing on critical actions in source code. We notice that there may exist several statements inserted before the target logging statement in the source code. For such cases, the logging text should focus on more critical statements and describe the specific statements with less general words. For example, as shown in Table 6.12e, there are a list of database-related actions, including “doDropTables” and “doCreateTables”, which is exactly what the generated log-2 describes, “executing sql <vid>”. However, based on the feedback from the participants as well as the original logging statement, the logging text “executing sql” is too general, while “deleted” is more specific and describes the more critical action “doDropTables”.

Besides, as shown in Figure 6.16, there are also some cases where log-1 and log-2 are hard to compare. This may be caused by two reasons: (1) **The two generated logging texts are inaccurate or even wrong.** For example, in Figure 6.17, the actual action in the source code is “scanning” file, instead of the generated “restoring” or “deleting”. 2)

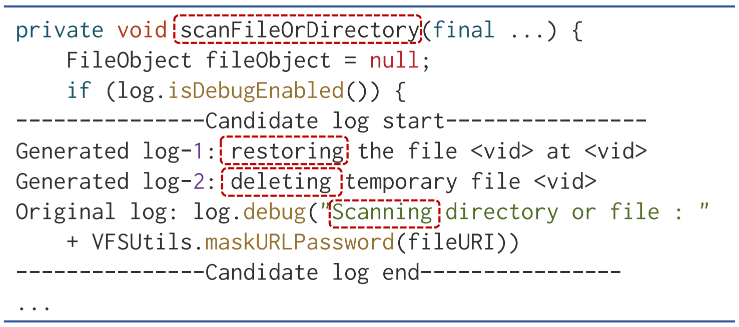

```
private void scanFileOrDirectory(final ...) {
    FileObject fileObject = null;
    if (log.isDebugEnabled()) {
---------------Candidate log start----------------
Generated log-1: restoring the file <vid> at <vid>
Generated log-2: deleting temporary file <vid>
Original log: log.debug("Scanning directory or file : "
    + VFSUtils.maskURLPassword(fileURI))
---------------Candidate log end----------------
...
```

Figure 6.17: An example generated logging text from Synapse.

The two generated logging texts have a very similar meaning. For example, in Figure 6.18, the two generated logging texts are almost the same, except for the missing preposition “to” in generated log-2, of which the influence can be ignored. Therefore, they are considered to convey the same information.

```
...
---------------Candidate log start----------------
Generated log-1: failed to transfer <vid> to <vid> retryable error attempt <vid> vid <vid>
Generated log-2: failed transfer <vid> to <vid> retryable error attempt <vid> vid <vid>
Original log: log.warn("Failed to transfer " + urlToInstall + " to " + machine + ", not a
    retryable error so failing: " + e)
---------------Candidate log end----------------
```

Figure 6.18: An example generated logging text from Brooklyn.

Summary

Overall, the logging texts generated by LoGenText-Plus have a higher quality than that generated by LoGenText. Besides, we identify five possible reasons that a developer may favor a logging text. The reasons can be used as a guideline for practitioners to improve the process of automated logging generation.

6.6 Threats to Validity

Internal Validity. In this work, we compare our approach with prior work by He et al. [59]. Meanwhile, pre-trained models of code have achieved new state-of-the-art results for several code-related tasks, such as clone detection, code search, and code completion [54]. Therefore, we also try to select UniXcoder [54] as a comparison, which is most recently released and has shown to have better performance than CodeBERT [44], CodeT5 [180], and GraphCodeBERT [53]. We conduct the experiments under two settings: (1) zero-shot logging statement completion, and (2) fine-tuning the model on our training dataset. Similar to our experiments, we use the pre-log code as input. Under the zero-shot setting, there is only an average of 26.6% logging statements generated among all the test inputs, with an average BLEU score of 12.9. Besides, we also fine-tune UniXcoder on our training dataset of each project, and there is an average of 34.7% logging statements generated among all the test inputs, with an average BLEU score of 22.2, which is slightly worse than that of the baseline approach and our proposed approaches. The reasons may come from (1) the lack of training data on logging statements and (2) the lack of optimized training objectives for logging statement-related tasks. Future work may consider designing logging statement-specific pre-training objectives and pre-training the model using the dataset curated for logging. Meanwhile, Mastropaolo et al. [120] propose to train a T5 model to support the automatic generation of complete log statements, including the generation of logging texts, where to log, and which level to log. However, as mentioned in the work of Mastropaolo et al. [120], the generated logging texts have a BLEU score of 15, which is also lower than the average result (i.e., 30.3) reported in our thesis. However, we believe that the performance of the T5 model can be further improved by, for example, (1) training on a larger corpus (currently, it is only trained with 6M Java methods) and (2) including the AST or other types of information extracted from source code, which has shown to be useful for code-related tasks. In RQ2, we attempt to include two types of context information to further improve LoGenText. Similarly, we design two strategies in RQ3 to incorporate logging texts from similar code snippets. There could exist other context information and other strategies for integrating the context information, while our findings do not in any way claim to generalize the usefulness of other types of context information or other integration strategies. We evaluate the effectiveness of LoGenText and LoGenText-Plus based on both quantitative metrics (i.e., BLEU and ROUGE) and human ratings. The quantitative metrics may not reflect the actual quality of the generated logging from developers' perspective, while the human ratings may include subjective bias introduced by the individual participants. However, to mitigate this effect, we try our best to invite more than 40 participants. The number of participants is much larger than that of previous research [176]. Future work should consider further evaluating LoGenText and LoGenText-Plus by using them in a real-life industrial setting.

External Validity. In this chapter, we evaluate LoGenText and LoGenText-Plus based on 10 subject systems. All of the subject systems are open-source systems that are mainly written in Java. In addition, all of the subject systems are server or desktop applications, while logging practices on mobile devices are found to be different [202]. Evaluating LoGenText and LoGenText-Plus on other systems that are written in other languages, with closed-source code, or running on mobile devices, may further demonstrate the effectiveness and limitations of our approach.

Construct Validity. Our data (e.g., logging texts, pre-log code, post-log, and ASTs) are extracted based on the srcML tool [31]. srcML is a mature tool and has been widely used in various software engineering research. Nevertheless, the quality of the data generated by srcML may impact the results of our study. LoGenText and LoGenText-Plus require several hyperparameters for the training process, such as the dimensions, the number of layers, and the number of attention heads, which may impact the results of generating logging texts. To minimize the bias caused by the hyperparameter configurations, we follow the practices from prior studies [83, 172] to configure the hyperparameters. Performing further fine-tuning on these hyperparameters may even further improve the results from LoGenText and LoGenText-Plus. In our evaluation, the data from each project is randomly split into 80%/10%/10% training, validation, and testing datasets. The evaluation results may show some differences with other splits of the training, validation, and testing datasets. Besides, we find that there exists a duplication of the data samples between the training and testing datasets, which may also impact the evaluation results. Specifically, there are 1 to 29 or 0.5% to 6.6% duplicate logging statements in the studied subjects. However, we did not remove these duplicates as the number of duplicate instances is relatively small, and we want to evaluate our approach in a real-life situation where duplicate logging statements do exist [94, 98]. Future work may consider exploring how duplicate logging statements would impact the tool.

6.7 Conclusion

In this chapter, we present our approach, LoGenText, and its improved version, LoGenText-Plus, which automatically generates the textual descriptions of logging statements based on neural machine translation models. By comparing the generated logging texts with the actual logging texts in the source code, we find that both LoGenText and LoGenText-Plus show promising results in the automated generation of logging texts. Our approach LoGenText-Plus, which leverages the logging template information, outperforms the state-of-the-art LoGenText and the baseline approach in terms of both quantitative metrics (BLEU and ROUGE) and human ratings. Our research sheds light on promising research opportunities that exploit and customize neural machine translation models for the automated generation of logging statements, which will reduce developers' efforts in logging development and maintenance and potentially improve the overall quality of software logging.

Chapter 7

Retroactively Analyzing Existing Logging Texts

Prior work shows that misleading logging texts (i.e., the textual descriptions in logging statements) can be counterproductive for developers during their use of logs. One of the most important types of information provided by logs is the *temporal information* of the recorded system behavior. For example, a logging text may use a *perfective aspect* to describe a fact that an important system event has *finished*. Although prior work has performed extensive studies on automated logging suggestions, few of these studies investigate the temporal relations between logging and code.

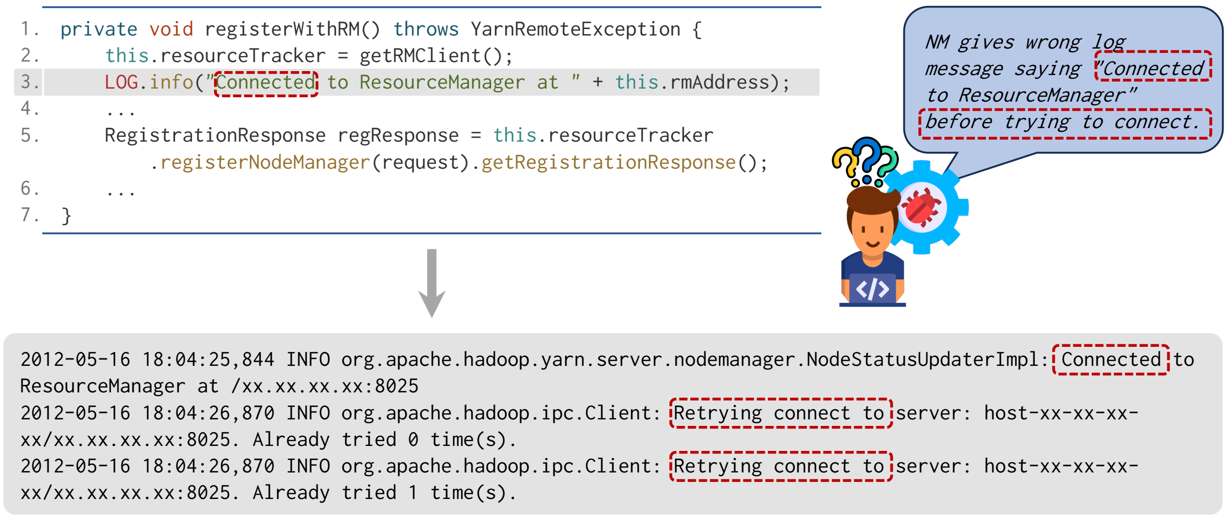
In this chapter, we make the first attempt to comprehensively study the temporal relations between logging and its corresponding source code. In particular, we focus on two types of temporal relations: (1) logical temporal relations, which can be inferred from the execution order between the logging statement and the corresponding source code; and (2) semantic temporal relations, which can be inferred based on the semantic meaning of the logging text. We first perform qualitative analyses to study these two types of logging-code temporal relations and the inconsistency between them. As a result, we derive rules to detect these two types of temporal relations and their inconsistencies. Based on these rules, we propose a tool named TempoLo to automatically detect the issues of temporal inconsistencies between logging and code. Through an evaluation of four projects, we find that TempoLo can effectively detect temporal inconsistencies with a small number of false positives. To gather developers' feedback on whether such inconsistencies are worth fixing, we report 15 detected instances from these projects to developers. 13 instances from three projects are confirmed and fixed, while two instances of the remaining project are pending at the time of this writing. Our work lays the foundation for describing temporal relations between logging and code and demonstrates the potential for a deeper understanding of the relationship between logging and code.

7.1 Introduction

Logging texts provide high-level human-readable information and are usually written to describe the behaviors of the corresponding source code. Well-written logging texts can provide developers and other software practitioners with valuable information for system comprehension or failure diagnosis. Thus, it is important for developers to write proper logging texts for their logging statements. Recent work shows that incorrect logging texts often make the use of logs counterproductive [23, 58, 89, 98]. For example, according to a Hadoop issue report[1], the logging text of the logging statement in Figure 7.1 is misleading: the logging text indicates a perfective action ("connected"), while the source code corresponding to the action (line 5) is placed after the logging statement, causing an inconsistency between the textual description of the logging statement and its logical relationship with the corresponding source code. When the logging statement is executed and a log message is produced, the log message will provide misleading information that the "connection" has been established while it is not. To avoid such confusion, the word "connected" in the logging text was changed to "connecting" (indicating a progressive action) in the patch that fixed the issue.

Prior work has performed extensive studies on software logging, including the studies that perform empirical investigations of logging practices [24, 27, 59, 98], that characterize and detect logging-related issues, as well as the studies that propose automated tools to support where to log [38, 39, 95, 210], what to log [37, 59, 120], and how to choose log levels [86, 96]. While a few studies indicate and discuss the importance of the relationship between logging and code [59, 89], none of them offer a satisfactory solution to address this issue. For example, a recent study by He et al. [59] finds that developers insert logging statements to describe three types of program operations (i.e., completed, current, and next operations). In their study, only the relative position between the logging statement and its corresponding code is considered, while the underlying intention in the developer-written logging text is ignored. However, both aspects are critical for logging quality and various log analysis tasks. As explained previously in the Hadoop logging example (Figure 7.1), the inconsistency between these two aspects can mislead practitioners who rely on analyzing logs for various tasks (e.g., debugging). Specifically, four of the logging benefits (e.g., knowing the status of an ongoing event) observed in a recent survey [89] would be impaired with such inconsistency. Moreover, by carefully examining real-life log

[1] https://issues.apache.org/jira/browse/MAPREDUCE-4262



```
1.  private void registerWithRM() throws YarnRemoteException {
2.      this.resourceTracker = getRMClient();
3.      LOG.info("Connected to ResourceManager at " + this.rmAddress);
4.      ...
5.      RegistrationResponse regResponse = this.resourceTracker
            .registerNodeManager(request).getRegistrationResponse();
6.      ...
7.  }
```

NM gives wrong log message saying "Connected to ResourceManager" before trying to connect.

```
2012-05-16 18:04:25,844 INFO org.apache.hadoop.yarn.server.nodemanager.NodeStatusUpdaterImpl: Connected to
ResourceManager at /xx.xx.xx.xx:8025
2012-05-16 18:04:26,870 INFO org.apache.hadoop.ipc.Client: Retrying connect to server: host-xx-xx-xx-
xx/xx.xx.xx.xx:8025. Already tried 0 time(s).
2012-05-16 18:04:26,870 INFO org.apache.hadoop.ipc.Client: Retrying connect to server: host-xx-xx-xx-
xx/xx.xx.xx.xx:8025. Already tried 1 time(s).
```

Figure 7.1: A code snippet from Hadoop with a logging statement (line 3), generated logs, and its related issue report.

anti-patterns [24] and log bugs data [58], we observe many real-world cases (over 20) fixing such inconsistency.

Therefore, in this work, we make the first attempt to comprehensively study the relationship between logging and its corresponding code. In particular, we focus on the temporal relations. Specifically, we study the logging-code temporal relations from two perspectives: (1) **logical temporal relations**, which can be inferred from the execution order between the logging statement and its corresponding source code; and (2) **semantic temporal relations**, which can be inferred based on the semantic meaning of the logging text. While the logical temporal relations describe *what the actual order is in the code*, the semantic temporal relations describe *what the order is inferred from the generated logs*. When there is an inconsistency between these two relations (i.e., **temporal inconsistency**), it is misleading to practitioners who rely on the clues provided by logs to understand system runtime behaviors [46, 158]. In this work, we first perform qualitative analyses to study these two types of logging-code temporal relations and the issues of temporal inconsistency. Based on the observations from our qualitative analyses, we also propose a tool (named TempoLo) to automatically detect the issues of temporal inconsistency in the source code.

We evaluate our tool on four open-source projects. Our evaluation contains three parts: (1) applying our tool to the logging statements used in our qualitative study, which shows that the tool can cover a majority (78.8%) of the manually identified temporal inconsistencies; (2) applying our tool to the remaining logging statements that are not manually analyzed, which shows that the tool can successfully detect another 326 inconsistencies with a relatively small number of false positive cases (48 cases and almost half are caused by the dependent NLP library); and (3) applying the tool to another dataset of temporal inconsistencies which we collected from the commit history of the studied projects, which shows that our tool can detect 83.3% of the inconsistencies. To gather developers' feedback on whether such inconsistencies are worth fixing, we report 15 detected instances from these projects to developers. 13 instances from three projects are confirmed and fixed, while two instances of the remaining project are still pending at the time of this writing.

The contributions of this chapter include:

- We provide empirical observations on the temporal relations between logging and code.
- We derive rules to detect the logical and semantic temporal relations between logging and code, as well as rules to detect logging-code temporal inconsistencies.
- We implement a tool that can automatically detect logging-code temporal inconsistencies in the source code.

Our work is an important step toward analyzing the relationship between logging and code. Our empirical observations and tool can raise developers' and researchers' awareness of the importance of the temporal relations between logging and code, to avoid and identify temporal relation-related bugs. Our research also sheds light on promising research opportunities that exploit other types of relations between logging and code (e.g., semantic inconsistencies) to improve software logging or detect logging anti-patterns in the source code, which will potentially improve the overall quality of software logging.

Chapter organization. Section 7.2 presents the background of temporal relations. Section 7.3 describes our subject projects, and data collection, and gives an overview of our study. Section 7.4 describes the approaches and results of our qualitative study of the temporal relations between logging and code. Section 7.5 presents the implementation and evaluation of our tool that automatically detects logging-code temporal inconsistencies which is based on the observations from our qualitative study. Section 7.6 discuss the threats to the validity of our results. Finally, Section 7.7 concludes the chapter.

7.2 Background

In this section, we present the concepts of temporal relations that are widely studied in the natural language community.

Before describing temporal relations, we first focus on the concept of “event” which is a fundamental term in natural language. **Event** is defined as a situation that happens or occurs [145], and it is often expressed in verbs (as shown in the example below) to describe an action or a transition. Another important concept is the semantic relation, also called **temporal relation**, that holds between relevant events. Given the following example,

> *The server* ***stopped*** *unexpectedly, we are* ***starting*** *it again.*

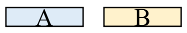
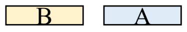
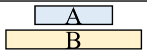
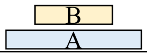
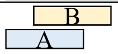
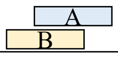
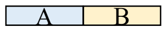
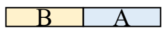
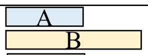
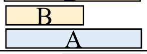
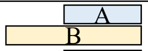
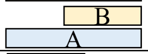
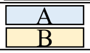
According to the definition of events, in this sentence, there are two events, **stopped** (E1) and **starting** (E2). The occurring order of these two events is the temporal relation. In this example, event E2 should happen after event E1.

Identifying the events and the temporal relations among them plays a vital role in many natural language processing (NLP) tasks, such as temporal information extraction (IE) [103], question answering [154] and knowledge base (KB) construction. To better annotate the events and temporal relations, researchers have proposed several representation schemas (e.g., Allen’s interval algebra [9], STAG [156], and TimeML) [145]. In our work, we mainly focus on Allen’s interval algebra [9], as it has become the standard representation [34, 103].

Allen [9] first proposed the interval-based algebra for representing temporal relations that may exist between any event pair in natural language. The representation consists of a set of 13 distinct and exhaustive interval relations (i.e., the relations between two time intervals) and are listed in Table 7.1 [34, 130], where A and B are two relevant events. The previous example describes two events, E1 and E2, and the corresponding temporal relation should be E2 **after** E1 (or E2 **met-by** E1, as it is possible that E2 starts when E1 ends.).

Since the appearance of Allen’s algebra, researchers also propose modified temporal relations for capturing temporal relations in different domains [65, 130, 156, 173]. For example, Styler IV et al. [65] combine Allen’s algebra and TimeML schema and identify five temporal relations for clinical narratives. Besides, Mostafazadeh et al. [130] find that using a number of four relations is sufficient to handle the inter-event temporal relation in ROCStories corpus [129]. Considering the wide research of temporal relations for NLP, we would like to examine whether we can formally define our own set of temporal relations to model the relations between the logging statements and source code.

Table 7.1: Allen's 13 temporal relations.

Relation (A to B)	Visualization	Explanation
Before		A ends before B starts
After		A starts after B ends
During		A starts and ends while B is ongoing
Contains		B starts and ends while A is ongoing
Overlaps		A starts before B and ends during B
Overlapped-by		B starts before A and ends during A
Meets		A ends at the point B begins
Met-by		B ends at the point A begins
Starts		Share the start point, but A ends before B ends
Started-by		Share the start point, but B ends before A ends
Finishes		A and B share end point, but A begins later
Finished-by		A and B share end point, but B begins later
Equals		A and B start and end at the same time

7.3 Study Setup

In this section, we describe the setup of our study[2]. Figure 7.2 shows an overview of our study. Overall, our study contains two parts: (1) a qualitative study of the temporal relations between the logging statement and source code, and the logging-code temporal inconsistencies, and (2) the implementation and evaluation of a tool for automatically identifying the temporal inconsistencies. Below, we first present our subject projects and the collection of data for the qualitative study, then we provide an overview of our qualitative study and the implementation and evaluation of our tool.

7.3.1 Subject Projects

We base our case study on four open-source Java projects: Hadoop, Tomcat, JMeter and ActiveMQ. The four selected projects are from different application categories: Hadoop is a distributed server-side data processing system; Tomcat is a server-side application

[2]The replication package is available at https://github.com/senseconcordia/TempoLo-replication-package

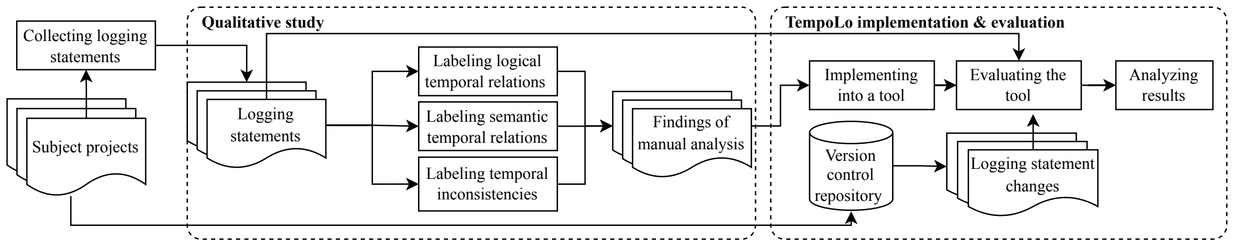

Collecting logging statements
Subject projects
Qualitative study
Logging statements
Labeling logical temporal relations
Labeling semantic temporal relations
Labeling temporal inconsistencies
Findings of manual analysis
TempoLo implementation & evaluation
Implementing into a tool
Evaluating the tool
Analyzing results
Version control repository
Logging statement changes

Figure 7.2: An overview of our study.

used for powering web applications; JMeter is a client-side software for conducting load testing; ActiveMQ is a middleware project that provides useful messaging service for both the server and client-side projects. We choose the subject projects since they are widely used and actively maintained, and have been studied in prior research [23, 24]. The details of the studied versions of these projects are listed in Table 7.2. The source lines of code of the studied projects range from 145K to 1.8M. These projects have about ∼2K to ∼13K logging statements, among which 94.3% to 97.6% have logging texts.

Table 7.2: Details of the studied projects.

Project	**Version**	**SLOC**	**# of logging statements**	**# of logging statements with text**
ActiveMQ	5.6.0	412K	2,139	2,087 (97.6%)
Hadoop	3.4.0	1.8M	13,204	12,463 (94.3%)
JMeter	5.5.0	145K	1,932	1,842 (95.3%)
Tomcat	10.1.0	349K	2,590	2,477 (95.6%)
Total		2.7M	19,865	18,868 (95.0%)

7.3.2 Data Collection: Logging Statements and Logging Statement Changes

As shown in Figure 7.2, we collect logging statements data from the source code of the subject projects to perform our qualitative study of the temporal relations between logging and code and to evaluate our tool that detects temporal inconsistencies. In addition, we collect logging statement change data from the version control repositories of the subject projects to evaluate our tool. Below, we describe our data collection processes.

7.3.2.1 Collecting logging statements

We collect logging statements from the latest versions (as indicated in Table 7.2) of the four subject projects at the time of writing this chapter[3]. We use static analysis and regular expressions to identify the logging statement and the method that contains the logging statement. More specifically, we use JavaParser [134] to find out methods that are invoked in each Java file as logging is typically a method call (e.g., log.info()), then use regular expressions to filter out the logging statements using keywords (e.g., "log", "logger", "logging"). The statistics of the collected logging statements are shown in Table 7.2.

To conduct our manual analysis, we then do random sampling with 95% confidence level and a 5% confidence interval [36] on the collected logging statements. We finally select 326, 373, 321, and 335 logging statements (1,355 samples in total) for ActiveMQ, Hadoop, JMeter and Tomcat, respectively. Note that we have sampled from each subject individually, rather than combining all the logging statements into one dataset for sampling, though it would dramatically reduce our manual labeling efforts (from 1,355 samples to 377 samples). The main reason is that the distribution of the logging statements across different projects is highly imbalanced (i.e., the differences in the number of logging statements in different projects), thus to better understand the characteristics of the temporal relations between the logging statements and the target source code from a wider perspective, we do the sampling for each project individually.

7.3.2.2 Collecting logging statement changes

We also construct an oracle dataset from the commit history of our subject projects to further evaluate the performance of the tool. Our idea is that: if developers change the position of a logging statement relative to its corresponding code, it indicates a potential temporal inconsistency in the first place. We collect all the commits of the four studied projects and keep the commits that have logging statement changes. Then, we manually examine the logging statement changes that fix temporal inconsistencies. We detail the steps in Section 7.5.4.3.

7.3.3 Overview of the Qualitative Study

With the sampled logging statements (see Section 7.3.2.1), we manually label the temporal relations between the logging statement and the target source code. Four participants jointly perform the labeling process.

[3]The time of the data collection is July 2021.

We analyze the logging-code temporal relations from two perspectives: (1) **logical temporal relations**, where our assumption is that the logging statement and its corresponding source code can be considered as a different but relevant event pair and the temporal relation (in other words, execution order) of the event pair can be inferred from their relative position in the code snippet; and (2) **semantic temporal relations**, where the action described in the logging text and the execution of the logging statement is regarded as an event pair, and the temporal relations can be inferred from the semantic meaning of the logging text, which is similar to existing NLP tasks [103, 173].

We then manually identify the temporal inconsistencies. We consider that there exists a temporal inconsistency if the two types of relations violate each other. We detail our qualitative study steps in Section 7.4.

7.3.4 Overview of Our Tool Implementation and Evaluation

In this step, based on the observations from our qualitative study, we implement a tool named TempoLo that automatically detects the logging-code temporal relations and the temporal inconsistencies for a given logging statement and its containing method. The tool implements three functionalities: (1) logical temporal relation detection, (2) semantic temporal relation detection, and (3) temporal inconsistency detection.

To evaluate the effectiveness of TempoLo, we apply it to the unsampled logging statements (the remaining logging statement after sampling for manual labeling) to examine its ability to detect new temporal inconsistencies. As there is no readily available oracle dataset for temporal inconsistencies, we also construct an oracle dataset from the commit history of the subject projects for further evaluation. Finally, we reported the detected inconsistencies as bugs through issue reports and pull requests to the developers of the four studied projects. We detail the steps and results in Section 7.5.

7.4 A Qualitative Study

In this section, we describe the steps of our qualitative study that aims to understand the temporal relations between logging and code, as well as to identify the inconsistencies between the temporal relations inferred from the code (i.e., logical temporal relations) and that inferred from the semantic meaning of the logging text (i.e., semantic temporal relations).

7.4.1 Studying Logical Temporal Relations

In this section, we manually label and analyze the logical temporal relations.

7.4.1.1 Labeling logical temporal relations

We follow a three-step manual labeling process:

Step 1. We start by manually labeling a random sample of 406 logging statements (i.e., 30% of the total 1,355 sampled logging statements). In this step, we employ four participants to do the labeling. The logical temporal relation of each logging statement is labeled by two annotators separately. Therefore, each annotator is assigned 203 logging statements to label. Each logging statement together with its corresponding code snippet is provided to the annotators using a URL that locates the logging statement in the corresponding GitHub repository. The annotators decide the most appropriate relation for each logging statement from Allen's 13 temporal relations.

Step 2. Once the 406 logging statements are labeled, the four annotators compare their labeling results. The results in this step have a substantial agreement for the labeling of the logical temporal relations (Cohen's Kappa of 0.81). Each logging statement is labeled by two annotators, and thus, when there is any disagreement regarding the labeling, the other two annotators would join and discuss until they reach a consensus.

Step 3. Based on the common understanding of the labels obtained from the last step, the remaining 949 from the total sample of 1,355 logging statements are equally distributed to the four annotators, then, each participant labels around 237 logging statements individually.

Note that there exist some logging statements that do not have explicit temporal relations with the source code, and the annotators simply label the temporal relation as "N/A". For example, considering the code snippet in Table 7.3(a), there is no clear target source code for the logging statement, and thus, it is impossible to infer a temporal relation.

7.4.1.2 Identified logical temporal relations

In total, we have identified five logical temporal relations between the logging statement and its corresponding source code, as shown in Table 7.3. As compared with Allen's 13 relations (cf., Section 7.2), we do not include the relations **contains, overlaps, overlapped-by, starts, started-by, finishes, finished-by** and **equals**. The reason is that we consider

Table 7.3: An overview of the logical temporal relations.

Relation	**Code snippet**

N/A	`private void handleBrowse(SampleResult ...`
	`    LOGGER.debug("isBrowseOnly");`
	`    StringBuilder sb = new ...`
	`    }`
	(a) JMSSampler.java (JMeter)

During	`public void setRunning(boolean running, String host) {`
	`    log.info("setRunning({}, {})", running, host); ...`
	`    }`
	(b) JMeterMenuBar.java (JMeter)

Before	`public void run() { ...`
	`    log.debug("Sampler start");`
	`    ...`
	`    sample();`
	`    ...}`
	(c) AbstractPerformanceSampler.java (ActiveMQ)

After	`public void onAMQPData(Object command) ...`
	`    frame = header.getBuffer();`
	`    ...`
	`    LOG.trace("Server: Received from client: {} bytes", ...`
	`    }`
	(d) AmqpConnection.java (ActiveMQ)

Meets	`void waitForAuthentication() ...`
	`    LOG.debug("Waiting for authentication response");`
	`    handler.waitForAuthentication();`
	`    }`
	(e) Application.java (ActiveMQ)

Met-by	`protected void unregisterProducer( ...`
	`    managementContext.unregisterMBean(key);`
	`    } catch (Throwable e) {`
	`        LOG.warn("Failed to unregister MBean {}", key);`
	`        ...`
	`        }`
	(f) ManagedRegionBroker.java (ActiveMQ)

the execution of a logging statement as a time point and the corresponding code as a time interval. Below, we discuss each of our identified logical temporal relations.

During: The logging statement executes while the target source code is ongoing. As shown in Table 7.3(b), the corresponding code of the logging statement is the whole method, and the logging statement executes while the method is ongoing, thus, we label the relation as **during**.

Before: The logging statement executes before the corresponding source code. As shown in Table 7.3(c), the corresponding code of the logging statement is the method "`sample`", and there is other source code between the logging statement and the corresponding code, thus, we label the relation as **before**.

After: The logging statement executes after the corresponding source code. As shown in Table 7.3(d), the target code of the logging statement is the assignment operation "`frame = header.getBuffer();`", and there is another source code between the logging statement and the corresponding code, thus, we label the relation as **after**.

Meets: The logging statement executes right before the corresponding source code. As shown in Table 7.3(e), the corresponding code of the logging statement is the method "`waitForAuthentication()`", and there is no other source code between the logging statement and the corresponding code, thus, we label the relation as **meets**.

Met-by: the logging statement executes right after the corresponding source code. As shown in Table 7.3(f), the corresponding code of the logging statement is the method "`unregisterMBean(key)`", and there is no other operations between the logging statement and the corresponding code, thus, we label the relation as **met-by**.

7.4.1.3 Findings from labeling logical temporal relations

During the labeling process, the most challenging part is locating the target source code of the logging statement. However, we find that the main verb in the logging text, which typically shows the action that occurs in a sentence, can be effectively used for locating the corresponding source code. Therefore, we list observations for locating the corresponding source code based on the logging text. The number in the parentheses is the proportion of these matches respectively.

Finding L.1: The main verb in the logging text (partially) matches the corresponding source code (40.4%). For instance, in Table 7.3(e), the main verb of the logging text "Waiting for authentication response" is "waiting", which partially matches "waitForAuthentication" in the statement "`handler.waitForAuthentication();`". As the logging statement is logically executed right before the corresponding code statement, we can label the relation as **meets**.

Finding L.2: The direct object or subject of the main verb in the logging text (partially) matches the corresponding source code (18.3%). The direct subject (object) is usually a noun or noun phrase that performs (receives) the action of the main verb. For instance, in Table 7.3(c), the logging text is "Sampler start" and the main verb is "start", whose subject "Sampler" partially matches the statement "`sample();`". There are other statements between the logging statement and the corresponding code, we, therefore, label the relation as **before**.

Finding L.3: The synonyms of the main verb in the logging text match the corresponding source code (11.1%). Developers also use synonyms of the main verb to describe the source code. As shown below, the corresponding code for the logging statement invokes the "append" method; while developers choose to use its synonym, "add" to describe the action.

```
private void initUserTags(BackendListenerContext ...
    userTagBuilder.append(',').append(...tagToStringValue(tagName))
    ...
    log.debug("Adding '{}' tag with '{}' value ", tagName, tagValue);
    ...
}
```

Finding L.4: The main verb in the logging text implicitly matches an operator of the corresponding source code (17.3%). For example, in Java, the phrase "instantiating a class" means calling the constructor of a class and creating an object of that class, which is done by using the "new" keyword. Below is an example, where "Instantiating" is used to describe the "new" operation.

```
private void startWebApp() {
    ...
    AHSWebApp ahsWebApp = new AHSWebApp(...
    LOG.info("Instantiating AHSWebApp at " +
    ...
}
```

Finding L.5: The open clausal complement of the main verb in the logging text matches the target source code (10.0%). The open clausal complement of the main verb is usually a predicative or clausal complement, which can also convey actions of the source code. For instance, in Table 7.3(f), the logging text is "Failed to unregister MBean" and the main verb is "Failed", whose open clausal complement, "unregister", matches the method "`unregisterMBean()`".

7.4.2 Studying Semantic Temporal Relations

In this section, we study semantic temporal relations that can be inferred from the semantic meaning of logging text.

7.4.2.1 Labeling semantic temporal relations

To get the labels of the logging texts, we follow the same three-step manual labeling process as in Section 7.4.1.1. We also achieve a substantial agreement on the semantic temporal relations (Cohen's Kappa of 0.83). Besides, there also exist logging statements of which the textual descriptions cannot be used to infer temporal relations. For instance, in Table 7.4(a), there is no explicit event that can be extracted, and the participants simply label the temporal relation as "**N/A**".

Table 7.4: An overview of the semantic temporal relations.

Relation	**Logging statement**
N/A	`(a)log.debug("Arg: {}", arg)`
During	`(b)log.info("setRunning({}, {})", running, host);`
Before	`(c)LOG.debug("Active scan starting")`
After	`(d)log.info("Thread started:{}", ...);`

7.4.2.2 Identified semantic temporal relations

In total, we have identified three semantic temporal relations inferred from the logging text, as shown in Table 7.4.

Considering the intrinsic similarity between this labeling process and the 2007 TempEval challenge in NLP [103, 173] as well the fact that it is hard to be certain about whether two events occur exactly during the same time or starting/ending right after/before each other [130] (e.g., we are unable to ensure that the "starting" would exactly occur right after the logging statement.), we thus follow the previous work [173] and restrict the original Allen's 13 interval relations to a set of three (**before**, **after** and **during**) temporal relations, which based on observation are enough to capture the semantic temporal relations. Below, we discuss our identified set of three semantic temporal relations.

During: The logging statement is called while the event described in the logging statement is ongoing. As shown in Table 7.4(b), the event is the "`setRunning`", and we can infer that the logging statement should execute while the event is ongoing, thus, we label the relation as **during**.

Before: The logging statement is called before the event. As shown in Table 7.4(c), the event is "`starting`", which should occur after the execution of the logging statement, thus, we label the relation as **before**.

After: The logging statement is called after the event. As shown in Table 7.4(d), the logging statement should be executed after the event "`started`", thus, we label the relation as **after**.

7.4.2.3 Findings from labeling semantic temporal relations

We discuss the findings that can be used for inferring the semantic temporal relations. We find that the tense and aspect of the main verb are two major pieces of evidence that can be used to infer semantic temporal relations. Tense and aspect are two important concepts in natural language embodying the linguistic encoding of time [1, 56]. There are two tenses in English, past and present, and two aspects, perfective and progressive, indicating that the action is complete or ongoing, respectively.

Finding S.1: If the main verb has a past tense or perfective aspect, the relation is often after. As shown in Table 7.4(d), the main verb "started" has a tense of past, and thus is labeled a relation, **after**. However, we find that in some cases, just tense solely can not determine the semantic relation, for instance, the expression "have stopped" has an **after** relation, but the tense is present. In this situation, we consider the aspect, as the aspect is perfective.

Finding S.2: If the main verb has a present tense or progressive aspect, the relation is often before. In Table 7.4(c), the main verb "starting" has a present tense and progressive aspect, and the temporal relation is labeled as **before**. Note that the main verb with a past tense can still have a progressive aspect, for example, given the verb phrase "was stopping", it has a progressive aspect but past tense, for such cases, we label them as **after**.

Finding S.3: If the logging text is in CamelCase and does not contain any explicit tense or aspect, the relation may be during. In Table 7.4(b), the logging text "setRunning" is in CamelCase and does not contain any verbs and thus the temporal relation is labeled as **during**.

7.4.3 Studying Temporal Inconsistencies

We have labeled the temporal relations for the collected logging statements in previous steps. As one of our goals is to uncover the patterns of temporal inconsistencies (i.e., the inconsistencies between the logical and semantic temporal relations), in this section, we first describe how to identify the temporal inconsistencies and then describe the findings.

Our way to identify the inconstancy is straightforward: if the temporal relation inferred from the source code (i.e., logical temporal relation) violates the temporal relation inferred from the textual description of the logging statement (i.e., semantic temporal relation), we label the logging statement as a logging statement with a temporal inconsistency. Below, we describe three intuitive rules for such inconsistencies (see Table 7.5).

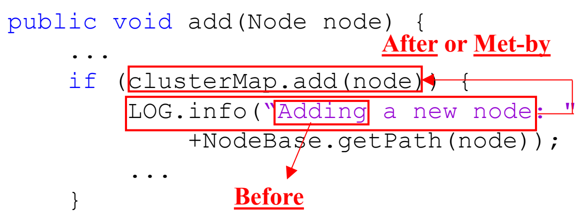
Rule 1: The temporal relation inferred from the source code is After or Met-by, but the temporal relation inferred from the logging text is Before. For instance, in Table 7.5(a), the logging statement "`LOG.info("Adding a new node: ");`" starts right after its target source code "`clusterMap.add(node)`", thus has a **met-by** relation inferred from the code. However, the semantic temporal relation inferred from the text of the logging statement is **before**, as the logging statement should execute earlier than the "Adding" event. The correct event expression should be "Added", or the logging statement should be moved to the line above the code "`if (clusterMap.add(node)){`".

Rule 2: The temporal relation inferred from the source code is Before or Meets, but the temporal relation inferred from the logging text is After. For instance, in Table 7.5(b), the logging statement "`log.warn("Existing AuthManager {} superseded by {}")`" executes before its target source code "`setProperty()`", and thus has a **before** relation inferred from the code. However, the temporal relation inferred from the text of the logging statement is **after**, as the execution of the logging statement should occur later than the "superseded" event.

Rule 3: The temporal relation inferred from the source code is During, but the temporal relation inferred from the logging text is After or Before. For instance, in Table 7.5(c), both the logging statements execute while the target method is ongoing, and thus they have a **during** relation. However, the temporal relations inferred from the texts of the two logging statements are **after** and **before**, respectively, as the execution of the logging statements should occur later and earlier than the "stopped" and "starting" events, respectively. Ideally, the first logging statement should execute after the "`registerHost()`" method, and the second one should execute before the "`registerHost()`" method. However, as we stated in Section 7.4.1.2, the execution of a logging statement is considered as a time point and thus, both should be moved to the end and start point of the method, respectively.

Table 7.5: An overview of the rules for labeling the inconsistencies of temporal relations.

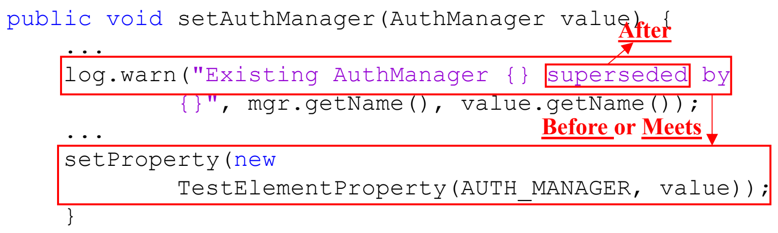
Rule 1: The temporal relation inferred from the source code is **After** or **Met-by**, but the temporal relation inferred from the logging text is **Before**.



```
public void add(Node node) {
    ...
    if (clusterMap.add(node)) {
        LOG.info("Adding a new node: "
            +NodeBase.getPath(node));
        ...
    }
```

(a) NetworkTopologyWithNodeGroup.java (Hadoop)

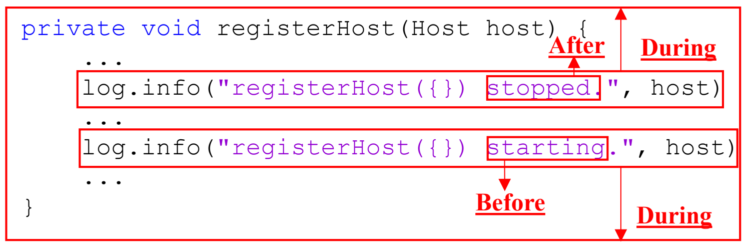
Rule 2: The temporal relation inferred from the source code is **Before** or **Meets**, but the temporal relation inferred from the logging text is **After**.



```
public void setAuthManager(AuthManager value) {
    ...
    log.warn("Existing AuthManager {} superseded by
            {}", mgr.getName(), value.getName());
    ...
    setProperty(new
            TestElementProperty(AUTH_MANAGER, value));
    }
```

(b) HTTPSamplerBase.java (JMeter)

Rule 3: The temporal relation inferred from the source code is **During**, but the temporal relation inferred from the logging text is **After** or **Before**.



```
private void registerHost(Host host) {
    ...
    log.info("registerHost({}) stopped.", host)
    ...
    log.info("registerHost({}) starting.", host)
    ...
}
```

(c) Modified MapperListener.java (Tomcat)

7.4.4 Summary of Our Qualitative Study

In previous sections, we have described our findings from the qualitative study. Here, we discuss the distribution of the logical and semantic temporal relations, as well as the temporal inconsistencies that we identified in the sampled dataset. Table 7.6 shows the statistics of the temporal relations and the identified inconsistencies in the four studied projects.

Table 7.6: The statistics of the manually labeled temporal relations.

	Types	**ActiveMQ**	**Hadoop**	**JMeter**	**Tomcat**	**Total**
Logical temporal relations	During	4	5	7	0	16
	Before	11	17	8	12	48
	After	24	44	24	34	126
	Meets	82	93	64	44	283
	Met-by	178	171	171	192	712
	N/A	27	43	47	53	170
Semantic temporal relations	During	5	5	5	0	15
	Before	93	119	75	61	348
	After	194	199	179	213	785
	N/A	34	50	62	61	207
Temporal inconsistencies	Rule 1	4	13	3	3	23
	Rule 2	4	1	4	1	10
	Rule 3	-	-	-	-	0
	Total	8	14	7	4	33

Note: We did not find any inconsistency that matches Rule 3, as Rule 3 is deduced from the correct cases during labeling.

Developers prefer to insert the logging statement right after/before the corresponding source code. Table 7.6 shows that about 73.4% of the sampled logging statements are located next to (either right before or after) the corresponding source code (i.e., with **meets** or **met-by** logical temporal relations).

Compared to inserting the logging statements before the target source code, developers prefer to see logs after the target source code being executed. Table 7.6 shows that about 61.8% of the sampled logging statements are inserted (right) after the target source code (i.e., with **met-by** or **after** logical temporal relations). This observation is consistent with the findings of previous work [24, 37] that logging statements are more relevant to their pre-log code.

There exist a non-negligible amount of inconsistencies (i.e., 2.4%) between the logical and semantic temporal relations, which can potentially confuse the end

users and make the use of logs counterproductive [89]. Our Rule 1 can detect more than twice the temporal inconsistencies as Rule 2. This gap may be caused by the fact that some developers just insert the base form (i.e., with present tense) of a verb in the logging text and put the logging statement after the target source code, paying little attention to the tense or aspect of the action itself. Note that Rule 3 is deduced from the correct cases. In particular, we find that developers may put the method name together with a verb indicating the start of the execution at the first line of the method body. Therefore, we did not detect any inconsistency by Rule 3 during our labeling.

Inspired by our qualitative study and the non-negligible amount of inconsistencies, we decide to implement our findings into a tool to assist developers in automatically detecting the logging-code temporal inconsistencies in the source code.

7.5 TempoLo: Automatically Detecting Temporal Inconsistencies between Logging and Code

In this section, we propose a tool, TempoLo, which automatically detects inconsistencies between the logical and semantic logging-code temporal relations, based on our findings from Section 7.4. Formally, given a logging statement and the method that contains the logging statement, the proposed tool can automatically analyze both their logical and semantic temporal relations and detect whether there is a temporal inconsistency. TempoLo is built as a static code analyzer that could be integrated into an IDE in practice, of which the storage needed is in a matter of KBs and time cost is almost negligible. We detail our implementation and evaluation in the rest of this section.

7.5.1 Detecting Semantic Temporal Relations

As we discussed in Section 7.4.2.3, the aspect and tense of the main verb can be effectively used to detect semantic temporal relations. In this section, we describe how to extract the logging text, and its main verb with tense and aspect.

7.5.1.1 Extracting the logging text

Following the approach used in prior work [37], we first extract the logging texts and variables from the logging statements which are collected in Section 7.3.2. Since our focus is on the main verb of the logging text, we replace the variables with a wildcard (`VID`). For instance, the extracted logging text of the logging statement in Table 7.4(d) is "`Thread started: VID`". Note that some projects (e.g., Tomcat) use an internationalization/localization helper class to fill the text in the logging statement instead of inserting the logging text directly. For example, Tomcat provides multiple local string files with different languages and uses a helper class and a key (e.g., in "`sm.getString("requestFacade.nullRequest")`", "sm" is the helper class, and "requestFacade.nullRequest" is the key) to retire the specific string. In such cases, we first convert the keys into the corresponding logging text using the underlying resource bundle (e.g., locale-specific resource).

7.5.1.2 Identifying the main verb and its tense and aspect

In this step, we use spaCy [62], an open-source NLP library, to perform dependency parsing and part-of-speech (POS) tagging on the extracted logging text. Dependency parsing is the task of defining the dependency relations (e.g., subject, object) between the tokens of a sentence. Part-of-speech (POS) tagging is the task of categorizing each token in the sentence with different types (e.g., verbs). Figure 7.3 shows the result (a dependency parsing tree) of performing dependency parsing and POS tagging on an example logging text. The arrows and labels at the top are the syntactic dependency relations and the tags at the bottom are the identified part-of-speech tags. Based on the dependency parsing tree, we consider the following two types of tokens as the main verb: (1) a token with a verb tag and its head (i.e., parent node) in the dependency tree is the token itself, or, (2) the first token with a verb tag. For example, in Figure 7.3, the head of the first token "Starting" is itself, and it has a tag "verb", thus "starting" is considered as the main verb of the logging text. Once we get the main verb, we can get the tense and aspect of the main verb using spaCy.

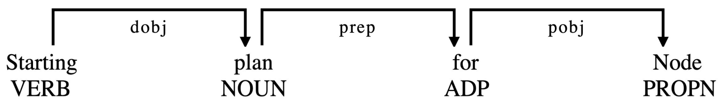


Figure 7.3: A dependency parsing tree of an example logging text.

7.5.1.3 Identifying the semantic temporal relation

To identify the semantic temporal relation, we apply the findings (i.e., Finding S.1 - Finding S.3) in Section 7.4.2.3 to the results from the previous step: if the main verb has a past tense or perfective aspect, we label the relation as **after**, and if the main verb has a present tense or progressive aspect, we label the relation as **before**. If the logging text is in CamelCase and does not contain any explicit tense or aspect, we label the relation as **during**.

7.5.2 Detecting Logical Temporal Relations

As we discussed in Section 7.4.1.3, locating the target source code of the logging statement plays a vital role in detecting logical temporal relations. In this section, we use the same approach (cf., Section 7.5.1) to extract the logging text and identify the main verb of the logging statement, after which, we perform lemmatization on the main verb to get its base form using spaCy. Below, we describe how we locate the source code using the lemmatized main verb.

7.5.2.1 Extracting all the potential statements

During our manual analysis, we also observe that the target source code is mainly method calls, assignment, return, if, else, or break statements. Therefore, in this step, we try to extract such statements as the potential statements. We first apply srcML to the method that contains the logging statement. srcML converts the source code into an XML tree, in which the tags provide the information of the potential statements. For example, all method calls are wrapped with a "call" tag. We then adopt XPath and Beautiful Soup[4] to extract the statements.

7.5.2.2 Locating the corresponding statements

Once we have extracted statements, we need to locate the corresponding statement. We implement the five rules observed (i.e., Finding L.1 - Finding L.5) in Section 7.4.1.3, to identify the corresponding statement[5]. To collect the synonyms of a given main verb, we adopt WordNet, which is a large lexical database of English [125].

7.5.2.3 Identifying the logical temporal relation

After we locate the target source code, we extract its line number, which is an attribute of the code component in the XML tree. Then we compare it with the line number of the

[4] https://www.crummy.com/software/BeautifulSoup/bs4/doc/

[5] We also implemented an ML-based approach to locate the target source code, but the top-1 accuracy is not satisfactory (only around 50%). Therefore, we opt to use the rule-based approach instead of the ML-based approach. Please refer to the replication package for more details.

logging statement and identify the logical temporal relations defined in Section 7.4.1.2.

7.5.3 Detecting Temporal Inconsistencies

Based on the result from the previous two steps, we implement the rules observed in Section 7.4.3 to detect temporal inconsistencies. For each logging statement, TempoLo would scan the pair of the detected logical and temporal relations to check whether they violate each other. Note that on one hand, there may exist some cases that do not have a verb in the logging text or we cannot match any target source code based on the rules, for such cases, we do not label their temporal relations. On the other hand, there is a possibility that one main verb may match more than one target source code in Section 7.5.2.2, and thus we collect multiple logical temporal relations for the logging statement. For these cases, we keep all the logical temporal relations, and if all the relations violate the semantic temporal relation, we then label it as a temporal inconsistency. By doing this, we can have a relatively acceptable accuracy and low false positive rate.

7.5.4 Evaluation

7.5.4.1 Evaluation on the manually identified logging statements

We first apply TempoLo to our manually identified logging statements. As shown in Table 7.6, we manually identify a total of 33 temporal inconsistencies, we run our tool on this dataset and can successfully detect 26 logging statements out of the 33 (i.e., 78.8%, with a false negative rate of 21.2%) having an inconsistent temporal relation. The results verify the usefulness of the findings and the correctness of the implementation. We then manually check the undetected cases, in order to further understand the reasons that may cause our tool to fail. We find two main reasons contributing to such detection errors: (1) **Mismatch of the target code.** In the step of determining the logical temporal relation of the logging statement, our tool returns the incorrect target source code. An example is shown below.

```
public NodePlan plan(DiskBalancerDataNode node)
    ...
    LOG.info("Starting plan for Node : {}:{}", node.getDataNodeName(),
    ...
}
```

The main verb of the logging text is identified as “starting”. The term “get” in the method call “getDataNodeName()” is one of the synonyms of “start”. Thus, our tool made a mistake by matching the logging text to the ”getDataNodeName()” method. (2) **No main**

verb found in the logging text. Our tool relies on spaCy to determine the main verb as well as the tense and aspect of the verb. There exist some cases in which spaCy may not return an incorrect result. For example, given the following logging statement, "`log.trace("Registering key for read:"+ key)`", we can infer that the main verb is "registering" with a progressive state. However, spaCy recognizes "registering" as a noun.

7.5.4.2 Evaluation on the remaining unsampled logging statements

In this part of the evaluation, we would like to check whether TempoLo can detect unseen temporal inconsistencies while having a low false positive rate. Therefore, we evaluate TempoLo on the remaining unsampled logging statements (the remaining logging statements after sampling for manual labeling). We manually check the detected cases, in order to understand the false positive rate and the reasons that may cause our tool to falsely report a correct case as an inconsistency.

TempoLo reports a total of 326 logging statements with temporal inconsistencies. Two participants manually check each of them to determine whether it is a true positive or false positive. The two participants achieve an agreement ratio of 78.3% and finally identify 48 false positives after reaching a consensus. In general, the false positives are caused either by (1) an incorrect main verb or (2) an incorrect target source code. Below, we detail the reasons that may lead to false positives:

(1) Incorrect identification of the main verb. If the logging text contains more than one verb, TempoLo may not detect the correct main verb. For example, TempoLo wrongly identifies "ask" as the main verb for the logging statement "`LOG.info("Requested container ask: "+ ...);`" (the correct one should be "requested"). This error is essentially caused by spaCy, as our way to detect the main verb relies on the dependency parsing tree generated by spaCy. Besides, TempoLo may wrongly return a non-verb token as the main verb. For example, in the logging statement, "`LOG.info("included nodes = "+ ...);`", TempoLo identifies "included" as the main verb, while it is an adjective. This error is caused by the POS tagging of the tokens, which is provided by spaCy. It is reasonable that spaCy has low accuracy in analyzing logging texts (which is different from sentences in NLP) even though we use the state-of-the-art transformer-based model. Future work can train a model on the annotated software engineering data to improve the performance of spaCy for software engineering.

(2) Incorrect match of the target source code. TempoLo may not be able to locate the target source code. For example, in the logging statement "`LOG.info("..., moving files from ... "`", developers use "moving files" to describe the actions of the source code, however, the target source code is "`mergePaths()`". Moreover, some descriptions in the

logging statements mean no action in the source code. For example, some developers use "stopping" in the catch block to indicate the end of the method, and TempoLo cannot detect such cases. To address this problem, developers can manually construct a project-orientated dictionary that maps the token in the logging text to the statements in the source code.

7.5.4.3 Evaluation on the logging statements collected from commit history

Finally, we evaluate whether TempoLo can detect historical issues of temporal inconsistencies between logging and code. Since such issues may not be all reported in the issue tracking system, we construct an oracle dataset from the commit history to further evaluate TempoLo. Below, we describe how we construct the dataset.

We first clone the git version control repositories of our subject projects and use "`git log`" to extract all the code commits. We then use "`git show`" to analyze the changes. In order to make sure the commit changes are relevant to logging statements, we focus on two types of commits: (1) only the locations of the source code are changed and the changed code contains logging statements; and (2) logging statements are only modified by changing their temporal information (e.g., verb tense in the sentence) in the logging text. After this step, we gathered a total of 1,273 code changes from our studied projects. Intuitively, not all commits are indeed related to logging statements. Therefore, two participants manually examine each of the commits to confirm that the commit is related to a logging statement. The two participants achieve an agreement ratio of 86.0% and collect 59 commits that are logging statement-related code changes. Furthermore, as the focus of this study is the temporal inconsistency, we find that a majority of the 59 commits are caused either by (1) regular code changes (e.g., new code is added or deleted, causing the change of the logging statement) or (2) logging efficiency (e.g., moving the logging statement out of loops). Therefore, we continue filtering the commits and finally, we extract a total of six logging statement changes that are related to temporal inconsistencies.

We then evaluate the tool on the six collected logging statements on their versions before the commit and TempoLo can successfully detect five of them (i.e., 83.3%, which is similar to the accuracy when being evaluated on the manually sampled dataset, with a false negative rate of 16.7%.). The one inconsistency case that our tool failed to detect is as follows:

```
+   log.info("Ending thread " + ...);
allThreads.remove(thread);
- log.info("Ending thread " + ...);
```

Our tool fails to detect this case since we cannot successfully match the action “remove” with the event “ending”.

7.5.4.4 Reporting issues to developers

To gather developers’ feedback on whether such inconsistencies are worth fixing, we report our detected instances to developers. To avoid spamming developers, we iteratively and gradually report issues to developers (e.g., by issue reports or PRs). We first only select and report two instances for each project to developers to know whether developers care about this kind of issue. Then if developers confirm the instances (e.g., by fixing the issue or accepting the PR), we further report more instances for that project. In the instance selection process, we prioritize the instances without ambiguity (e.g., we avoid instances that contain verbs that are the same in the present and past tense (e.g., “read”)). We have reported 15 instances covering the four projects. So far, all 13 instances from three projects are confirmed and fixed, and two instances of the other project are still under discussion. There are two main strategies for fixing the reported inconsistencies: (1) moving the logging statement to the proper position, and (2) correcting the tense or aspect of the main verb of the logging statement[6].

Summary

TempoLo can successfully detect the temporal inconsistencies in the source code with a low false positive rate of 14.7%. 13 out of 15 reported inconsistency instances have been fixed by developers and received positive feedback.

7.6 Threats to Validity

External Validity. As the study involves four Java open-source projects, the number of studied subjects and programming languages may pose a threat to the study’s validity. To mitigate this, careful consideration goes into the selection of subjects. These four analyzed projects are well-known and have gained considerable attention from developers and researchers, based on the stars on GitHub and existing research papers [23, 24]. Besides, we did obtain consistent empirical results across the four studied projects. Furthermore, Java is a mature programming language that provides built-in logging and third-party logging frameworks. Many studies [23, 25, 37, 86] focus only on Java logging because of

[6]The details of the pull requests can be found in our replication package.

its abundance of logging usage. Considering the universality of logging, the findings from Java logging studies may be applicable to other programming languages, which will require further research.

Internal Validity. Since the study involves a manual study, its validity can be influenced by the knowledge and experiences of the participants. As a way to mitigate human bias, we use peer review to reach a consensus as a baseline for further review. Participants are all professional researchers in the field of software engineering.

Construct Validity. This study leverages several third-party tools to preprocess the source code, such as JavaParser and spaCy. These tools could have their limitations. For example, almost half of the detected false positives are caused by spaCy. However, all of the tools used in this study were used in previous studies [11, 101] and are well recognized in the Computer Science community. For example, JavaParser has ∼4K Github stars and spaCy has ∼23K stars at the time of writing this chapter. We observe a 15% (48/326) false positive rate, while we admit that after all, our tool is a static analysis-based technique. 85% is rather on-par or above most static analysis-based tools. Besides, as our tool can pinpoint the location of inconsistencies, the cost of manually verifying a false positive is relatively low.

7.7 Conclusion

In this chapter, we have formally defined two sets of temporal relations between the logging statement and the corresponding source code: logical and semantic temporal relations. Based on the defined temporal relations, we have concluded three rules for detecting the temporal inconsistencies that can jeopardize the quality of logging. We then implement the rules as a tool to automatically detect such inconsistencies. By analyzing the results, we find that our tool can successfully detect the temporal inconsistencies in the source code with a relatively low false positive rate. We have reported some detected inconsistencies to the developers of each of our subject projects and received positive feedback. Moreover, our research sheds light on the promising research opportunity of formalizing other logging-code relations to assist in various downstream software engineering tasks (e.g., improving the quality of the automatically generated logging texts [37] and more accurately representing the actual temporal status of the events described in the logs [17]). Future research may build more and improve existing log analysis approaches by incorporating the defined temporal relationship.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

Researchers have been working on applying techniques from NLP to deal with code. However, source code and natural language are by nature different. Thus, directly applying the NLP techniques may not be optimal, and how to effectively modify these techniques to adapt to software engineering (SE) tasks remains a challenge.

In this dissertation, we have explored two research directions as a first step and proposed different approaches to optimize existing NLP techniques to encode the code-specific features. In Chapter 3, we discuss the challenges of applying word embedding techniques to learn distributed code representations and proposed StrucTexVec to analyze the impact of different training contexts on the quality of code embeddings. We find that the structural information extracted from the source code has a non-negligible effect. Then, to better capture such kind of structural information, in Chapter 4, we propose to represent the source code as a graph and leverage the GCN model to train the code embeddings from the context information provided by the graph representations. Then, based on the distributed code representation, we have also discussed another important intersection between the source code and natural language text, the textual information in logging statements. We have tried to improve the current logging texts from two aspects (1) proactively suggesting the generation of new logging texts, and (2) retroactively analyzing existing logging texts. In Chapter 6, we first propose an automated deep learning-based approach towards generating accurate logging texts by translating the related source code into short textual descriptions to reduce development efforts. Inspired by the temporal relations in NLP, we make the first attempt to model the temporal relationships between the existing logging texts and their corresponding code, aiming to detect the related temporal issues and thus improve the quality of the logging activities.

In summary, this dissertation validates our research hypothesis and shows that by encoding code-specific information, the performance of SE tasks can be further improved.

8.2 Future Directions

While this dissertation presents our current research toward the goal of providing suggestions to developers in optimizing NLP techniques to assist in SE tasks, there are still many research opportunities. Below, we discuss some directions for future work, focusing on code embeddings and logging practices:

8.2.1 Analyzing Code Information's Impact on Embedding Quality

In Chapter 3 and Chapter 4, we proposed approaches to encode the textual and structural (AST) information extracted from the code. Meanwhile, there exist other types of code-specific information that can be incorporated for training code embeddings. For example, bytecode, control flows and data flows can also be extracted and learned by our proposed framework. Therefore, future work may consider analyzing the impact of different types of code-specific information on different downstream software engineering tasks.

8.2.2 Learning Multi-modal Code Embeddings

In Chapter 4, one of our observations is that the code embeddings cannot perform very well on the code comment generation task. One possible reason is that our code embeddings are purely trained on the source code, while this task involved both natural language text (i.e., comment) and source code. Future work may consider incorporating both source code and comments during the training phase to create embeddings that capture the relationships between code and comments. This could involve a joint training approach where the model learns to represent the code and comments in a shared space.

8.2.3 Improving the Existing Works on Suggesting Logging Texts

Currently, we use all the pre-log code as the source input. However, in our recent work, by manually examining the logging statement, we find that sometimes, the logging text is only related to a part of the source code, either before or after the logging statement. Thus, how to better build the input for the generation of the logging texts (i.e., selecting the relevant part of the source code as input) is worth further exploring.

8.2.4 Logging for Different Computing Platforms

In the second part of this dissertation, we have proposed approaches to provide support for traditional software systems. Logging support for other computing platforms is also needed. For example, there are few studies that have worked on heterogeneous computing platforms, and in the future, researchers may investigate the potential of providing logging support for heterogeneous computing platforms (e.g., FPGA), especially considering the limitation of the FPGA architecture (e.g., lack of external memory to keep a large amount of runtime data, cannot print the message to the users).

8.2.5 Suggesting Analysis-aware Logging Statements

Most of the existing works provide logging suggestions based on historical logging decisions instead of the usage of logs in downstream tasks. This kind of suggestion would potentially cause a gap between the inserted logging statements and the usage of the produced logs, which may (negatively) impact the analysis process of these logs. Therefore, to better support the downstream log analysis tasks, inserting analysis-aware logging statements should be a direction for further improving the logging activities. For example, we can provide logging suggestions for performance monitoring.

References

[1] Mood & modality and dialogue sentiment | towards data science. https://towardsdatascience.com/mood-modality-and-dialogue-sentiment-b06cd36eca88, 2020.

[2] Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and DaeHun Nyang. Large-scale and language-oblivious code authorship identification. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[3] Ruchit Agrawal, Marco Turchi, and Matteo Negri. Contextual handling in neural machine translation: Look behind, ahead and on both sides. In *European Association for Machine Translation (EAMT)*, 2018.

[4] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*, 2020.

[5] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. Learning natural coding conventions. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2014.

[6] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. Suggesting accurate method and class names. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015.

[7] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning (ICML)*, 2016.

[8] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 2018.

[9] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 1983.

[10] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[11] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2019.

[12] Liliane Barbour, Foutse Khomh, and Ying Zou. Late propagation in software clones. In *International Conference on Software Maintenance (ICSM)*, 2011.

[13] Titus Barik, Robert DeLine, Steven Mark Drucker, and Danyel Fisher. The bones of the system: a case study of logging and telemetry at microsoft. In *International Conference on Software Engineering (ICSE): Companion*, 2016.

[14] Rachel Bawden, Rico Sennrich, Alexandra Birch, and Barry Haddow. Evaluating discourse phenomena in neural machine translation. In *North American Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2018.

[15] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade - Second Edition*. 2012.

[16] Francesco Bertolotti and Walter Cazzola. Fold2vec: Towards a statement-based representation of code for code comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2023.

[17] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[18] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. PHOG: probabilistic model for code. In *International Conference on Machine Learning (ICML)*, 2016.

[19] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics (TACL)*, 2017.

[20] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. Audio chord recognition with recurrent neural networks. In *International Society for Music Information Retrieval Conference (ISMIR)*, 2013.

[21] Lutz Büch and Artur Andrzejak. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019.

[22] Shaofeng Cai, Yao Shu, Gang Chen, Beng Chin Ooi, Wei Wang, and Meihui Zhang. Effective and efficient dropout for deep convolutional neural networks. *arXiv preprint arXiv:1904.03392*, 2019.

[23] Boyuan Chen and Zhen Ming (Jack) Jiang. Characterizing and detecting anti-patterns in the logging code. In *International Conference on Software Engineering (ICSE)*, 2017.

[24] Boyuan Chen and Zhen Ming Jack Jiang. Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation. *Empirical Software Engineering (EMSE)*, 2017.

[25] Boyuan Chen and Zhen Ming (Jack) Jiang. Studying the use of java logging utilities in the wild. In *International Conference on Software Engineering (ICSE)*, 2020.

[26] Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

[27] Jinfu Chen and Weiyi Shang. An exploratory study of performance regression introducing code changes. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017.

[28] Tse-Hsun Chen, Stephen W. Thomas, and Ahmed E. Hassan. A survey on the use of topic models when mining software repositories. *Empirical Software Engineering (EMSE)*, 2016.

[29] Zimin Chen and Martin Monperrus. The remarkable role of similarity in redundancy-based program repair. *arXiv preprint arXiv:1811.05703*, 2018.

[30] Shaiful Alam Chowdhury, Silvia Di Nardo, Abram Hindle, and Zhen Ming (Jack) Jiang. An exploratory study on assessing the energy impact of logging on android applications. *Empirical Software Engineering (EMSE)*, 2018.

[31] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. Lightweight transformation and fact extraction with the srcml toolkit. In *IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2011.

[32] Hetong Dai, Heng Li, Che-Shao Chen, Weiyi Shang, and Tse-Hsun Chen. Logram: Efficient log parsing using nn-gram dictionaries. *IEEE Transactions on Software Engineering (TSE)*, 2022.

[33] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

[34] Leon Derczynski. *Determining the Types of Temporal Relations in Discourse*. PhD thesis, University of Sheffield, 2013.

[35] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *North American Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2019.

[36] Zishuo Ding, Jinfu Chen, and Weiyi Shang. Towards the use of the readily available tests from the release pipeline as performance tests: Are we there yet? In *International Conference on Software Engineering (ICSE)*, 2020.

[37] Zishuo Ding, Heng Li, and Weiyi Shang. Logentext: Automatically generating logging texts using neural machine translation. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022.

[38] Zishuo Ding, Heng Li, Weiyi Shang, and Tse-Hsun (Peter) Chen. Towards learning generalizable code embeddings using task-agnostic graph convolutional networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2022.

[39] Zishuo Ding, Heng Li, Weiyi Shang, and Tse-Hsun Peter Chen. Can pre-trained code embeddings improve model performance? revisiting the use of code embeddings in software engineering tasks. *Empirical Software Engineering (EMSE)*, 2022.

[40] Zishuo Ding, Yiming Tang, Yang Li, Heng Li, and Weiyi Shang. On the temporal relations between logging and code. In *International Conference on Software Engineering (ICSE)*, 2023.

[41] Li Dong and Mirella Lapata. Coarse-to-fine decoding for neural semantic parsing. In *Association for Computational Linguistics (ACL)*, 2018.

[42] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *European Conference on Software Maintenance and Reengineering (CSMR)*, 2013.

[43] Vasiliki Efstathiou and Diomidis Spinellis. Semantic source code models using identifier embeddings. In *International Conference on Mining Software Repositories (MSR)*, 2019.

[44] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2020.

[45] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *IEEE International Conference on Data Mining (ICDM)*, 2009.

[46] Qiang Fu, Jian-Guang Lou, Qingwei Lin, Rui Ding, Dongmei Zhang, and Tao Xie. Contextual analysis of program logs for understanding system behaviors. In *International Mining Software Repositories (MSR)*, 2013.

[47] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *International Conference on Software Engineering (ICSE): Companion*, 2014.

[48] Philip Gage. A new algorithm for data compression. *C Users Journal*, 1994.

[49] Christian Garbin, Xingquan Zhu, and Oge Marques. Dropout vs. batch normalization: an empirical study of their impact to deep learning. *Multimedia Tools and Applications (MTA)*, 2020.

[50] Shlok Gilda. Source code classification using neural networks. In *International Joint Conference on Computer Science and Software Engineering (JCSSE)*, 2017.

[51] Yoav Goldberg. *Neural Network Methods for Natural Language Processing.* 2017.

[52] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* 2016.

[53] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations (ICLR)*, 2021.

[54] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. In *Association for Computational Linguistics (ACL)*, 2022.

[55] Kelvin Guu, Tatsunori B. Hashimoto, Yonatan Oren, and Percy Liang. Generating sentences by editing prototypes. *Transactions of the Association for Computational Linguistics (TACL)*, 2018.

[56] Friedrich Hamm and Oliver Bott. Tense and Aspect. In *The Stanford Encyclopedia of Philosophy.* 2021.

[57] Jacob A. Harer, Louis Y. Kim, Rebecca L. Russell, Onur Ozdemir, Leonard R. Kosta, Akshay Rangamani, Lei H. Hamilton, Gabriel I. Centeno, Jonathan R. Key, Paul M. Ellingwood, Marc W. McConley, Jeffrey M. Opper, Peter Chin, and Tomo Lazovich. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

[58] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. Studying and detecting log-related issues. *Empirical Software Engineering (EMSE)*, 2018.

[59] Pinjia He, Zhuangbin Chen, Shilin He, and Michael R. Lyu. Characterizing the natural language descriptions in software logging statements. In *International Conference on Automated Software Engineering (ASE)*, 2018.

[60] Vincent J. Hellendoorn and Premkumar T. Devanbu. Are deep neural networks the best choice for modeling source code? In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2017.

[61] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. On the naturalness of software. In *International Conference on Software Engineering (ICSE)*, 2012.

[62] Matthew Honnibal and Ines Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. 2017.

[63] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *International Conference on Program Comprehension (ICPC)*, 2018.

[64] Aylin Caliskan Islam, Richard E. Harang, Andrew Liu, Arvind Narayanan, Clare R. Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *Security Symposium (USENIX Security)*, 2015.

[65] William F. Styler IV, Steven Bethard, Sean Finan, Martha Palmer, Sameer Pradhan, Piet C. de Groen, Bradley James Erickson, Timothy A. Miller, Chen Lin, Guergana K. Savova, and James Pustejovsky. Temporal annotation in the clinical domain. *Transactions of the Association for Computational Linguistics (TACL)*, 2014.

[66] Zhen Ming Jiang, Ahmed E Hassan, Gilbert Hamann, and Parminder Flora. Automatic identification of load testing problems. In *International Conference on Software Maintenance (ICSME)*, 2008.

[67] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. Logging library migrations: a case study for the apache software foundation projects. In *International Conference on Mining Software Repositories (MSR)*, 2016.

[68] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, Mark D. Syer, and Ahmed E. Hassan. Examining the stability of logging statements. *Empirical Software Engineering (EMSE)*, 2018.

[69] Rafael Kallis, Andrea Di Sorbo, Gerardo Canfora, and Sebastiano Panichella. Predicting issue types on github. *Science of Computer Programming (SCP)*, 2021.

[70] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering (TSE)*, 2002.

[71] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning (ICML)*, 2020.

[72] Hong Jin Kang, Tegawendé F. Bissyandé, and David Lo. Assessing the generalizability of code2vec token embeddings. In *International Conference on Automated Software Engineering (ASE)*, 2019.

[73] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, and Katsuro Inoue. Mudablue: An automatic categorization system for open source repositories. *Journal of Systems and Software (JSS)*, 2006.

[74] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[75] Yoon Kim. Convolutional neural networks for sentence classification. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[76] Yunsu Kim, Duc Thanh Tran, and Hermann Ney. When and why is document-level context useful in neural machine translation? In *Workshop on Discourse in Machine Translation (DiscoMT)*, 2019.

[77] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *nternational Conference on Learning Representations (ICLR)*, 2017.

[78] Alexandros Komninos and Suresh Manandhar. Dependency based embeddings for sentence classification tasks. In *North American Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2016.

[79] Roy Chanchal Kumar and Cordy James R. A survey on software clone detection research. *Queen's School of computing TR*, 2007.

[80] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International Conference on Machine Learning (ICML)*, 2014.

[81] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, 1966.

[82] Omer Levy and Yoav Goldberg. Dependency-based word embeddings. In *Association for Computational Linguistics (ACL)*, 2014.

[83] Bei Li, Hui Liu, Ziyang Wang, Yufan Jiang, Tong Xiao, Jingbo Zhu, Tongran Liu, and Changliang Li. Does multi-encoder help? A case study on context-aware neural machine translation. In *Association for Computational Linguistics (ACL)*, 2020.

[84] Chen Li, Jianxin Li, Yangqiu Song, and Ziwei Lin. Training and evaluating improved dependency-based word embeddings. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[85] Guohao Li, Matthias Müller, Ali K. Thabet, and Bernard Ghanem. Deepgcns: Can gcns go as deep as cnns? In *International Conference on Computer Vision (ICCV)*, 2019.

[86] Heng Li, Weiyi Shang, and Ahmed E Hassan. Which log level should developers choose for a new logging statement? *Empirical Software Engineering (EMSE)*, 2017.

[87] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E Hassan. Towards just-in-time suggestions for log changes. *Empirical Software Engineering (EMSE)*, 2017.

[88] Heng Li, Tse-Hsun (Peter) Chen, Weiyi Shang, and Ahmed E. Hassan. Studying software logging using topic models. *Empirical Software Engineering (EMSE)*, 2018.

[89] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E. Hassan. A qualitative study of the benefits and costs of logging from developers' perspectives. *IEEE Transactions on Software Engineering (TSE)*, 2021.

[90] Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[91] Yangguang Li, Zhen Ming (Jack) Jiang, Heng Li, Ahmed E. Hassan, Cheng He, Ruirui Huang, Zhengda Zeng, Mian Wang, and Pinan Chen. Predicting node failures in an ultra-large-scale cloud computing platform: An aiops solution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2020.

[92] Yichen Li, Yintong Huo, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su, and Michael R. Lyu. Exploring the effectiveness of llms in automated logging generation: An empirical study. *arXiv preprint arXiv:2307.05950*, 2023.

[93] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In *International Conference on Learning Representations (ICLR)*, 2016.

[94] Zhenhao Li, Tse-Hsun (Peter) Chen, Jinqiu Yang, and Weiyi Shang. Dlfinder: characterizing and detecting duplicate logging code smells. In *International Conference on Software Engineering (ICSE)*, 2019.

[95] Zhenhao Li, Tse-Hsun Chen, and Weiyi Shang. Where shall we log? studying and suggesting logging locations in code blocks. In *International Conference on Automated Software Engineering (ASE)*, 2020.

[96] Zhenhao Li, Heng Li, Tse-Hsun Chen, and Weiyi Shang. DeepLV: Suggesting log levels using ordinal based neural networks. In *International Conference on Software Engineering (ICSE)*, 2021.

[97] Zhenhao Li, Heng Li, Tse-Hsun Peter Chen, and Weiyi Shang. Deeplv: Suggesting log levels using ordinal based neural networks. In *International Conference on Software Engineering (ICSE)*, 2021.

[98] Zhenhao Li, Tse-Hsun (Peter) Chen, Jinqiu Yang, and Weiyi Shang. Studying duplicate logging statements and their relationships with code clones. *IEEE Transactions on Software Engineering (TSE)*, 2022.

[99] Zhenhao Li, An Ran Chen, Xing Hu, Xin Xia, Tse-Hsun Chen, and Weiyi Shang. Are they all good? studying practitioners' expectations on the readability of log messages. In *International Conference on Automated Software Engineering (ASE)*, 2023.

[100] Zhenhao Li, Chuan Luo, Tse-Hsun Chen, Weiyi Shang, Shilin He, Qingwei Lin, and Dongmei Zhang. Did we miss something important? studying and exploring variable-aware log abstraction. In *International Conference on Software Engineering (ICSE)*, 2023.

[101] Bin Lin, Fiorella Zampetti, Gabriele Bavota, Massimiliano Di Penta, and Michele Lanza. Pattern-based mining of opinions in q&a websites. In *International Conference on Software Engineering (ICSE)*, 2019.

[102] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, 2004.

[103] Xiao Ling and Daniel S. Weld. Temporal information extraction. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2010.

[104] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Ming Li, Zhiyi Fu, and Zhi Jin. Characterizing logging practices in open-source software. In *International Conference on Program Comprehension (ICPC)*, 2020.

[105] Shangqing Liu, Xiaofei Xie, Jing Kai Siow, Lei Ma, Guozhu Meng, and Yang Liu. Graphsearchnet: Enhancing gnns via capturing global dependencies for semantic code search. *IEEE Transactions on Software Engineering (TSE)*, 2023.

[106] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[107] Zhiyuan Liu, Yankai Lin, and Maosong Sun. *Representation Learning for Natural Language Processing*. 2020.

[108] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Which variables should I log? *IEEE Transactions on Software Engineering (TSE)*, 2019.

[109] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. Cloudraid: hunting concurrency bugs in the cloud via log-mining. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018.

[110] Mingming Lu, Yan Liu, Haifeng Li, Dingwu Tan, Xiaoxian He, Wenjie Bi, and Wendbo Li. Hyperbolic function embedding: Learning hierarchical representation for functions of source code in hyperbolic space. *Symmetry*, 2019.

[111] Thang Luong, Ilya Sutskever, Quoc Le, Oriol Vinyals, and Wojciech Zaremba. Addressing the rare word problem in neural machine translation. In *Association for Computational Linguistics (ACL)*, Beijing, China, 2015.

[112] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research (JMLR)*, 2008.

[113] Diego Marcheggiani and Ivan Titov. Encoding sentences with graph convolutional networks for semantic role labeling. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2017.

[114] Leonardo Mariani and Fabrizio Pastore. Automated identification of failure causes in system logs. In *International Symposium on Software Reliability Engineering (ISSRE)*, 2008.

[115] Sameen Maruf and Gholamreza Haffari. Document context neural machine translation with memory networks. In *Association for Computational Linguistics (ACL),*, 2018.

[116] Sameen Maruf, André F. T. Martins, and Gholamreza Haffari. Selective attention for context-aware neural machine translation. In *North American Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2019.

[117] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*, 2018.

[118] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader-Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *International Conference on Software Engineering (ICSE)*, 2021.

[119] Antonio Mastropaolo, Nathan Cooper, David Nader-Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Using transfer learning for code-related tasks. *IEEE Transactions on Software Engineering (TSE)*, 2022.

[120] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. Using deep learning to generate complete log statements. In *International Conference on Software Engineering (ICSE)*, 2022.

[121] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software Maintenance (ICSM)*, 1996.

[122] Paul W. McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *International Conference on Program Comprehension (ICPC)*, 2014.

[123] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *International Conference on Learning Representations (ICLR)*, 2013.

[124] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2013.

[125] George A. Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 1995.

[126] Andriy Mnih and Geoffrey E. Hinton. A scalable hierarchical distributed language model. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2008.

[127] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *International Conference on Program Comprehension (ICPC)*, 2013.

[128] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *International Workshop on Artificial Intelligence and Statistics (AISTATS)*, 2005.

[129] Nasrin Mostafazadeh, Nathanael Chambers, Xiaodong He, Devi Parikh, Dhruv Batra, Lucy Vanderwende, Pushmeet Kohli, and James F. Allen. A corpus and cloze evaluation for deeper understanding of commonsense stories. In *North American Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2016.

[130] Nasrin Mostafazadeh, Alyson Grealish, Nathanael Chambers, James Allen, and Lucy Vanderwende. CaTeRS: Causal and temporal relation scheme for semantic annotation of event structures. In *Workshop on Events*, 2016.

[131] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2016.

[132] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Symposium on Network System Design and Implementation (NSDI)*, 2012.

[133] Masato Neishi, Jin Sakuma, Satoshi Tohda, Shonosuke Ishiwatari, Naoki Yoshinaga, and Masashi Toyoda. A bag of useful tricks for practical neural machine translation: Embedding layer initialization and large batch size. In *Workshop on Asian Translation (WAT)*, 2017.

[134] Smith Nicholas, Bruggen Danny van, and Tomassetti Federico. javaparser, 2017.

[135] Emilio Soria Olivas, Jos David Mart Guerrero, Marcelino Martinez-Sober, Jose Rafael Magdalena-Benedito, L Serrano, et al. *Handbook of research on machine learning applications and trends: Algorithms, methods, and techniques: Algorithms, methods, and techniques*. 2009.

[136] Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *International Conference on Program Comprehension (ICPC)*, 2010.

[137] Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery: A ten-year retrospective. In *International Conference on Program Comprehension (ICPC)*, 2020.

[138] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *North American Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2019.

[139] John F. Pane, Chotirat (Ann) Ratanamahatana, and Brad A. Myers. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies (IJHCS)*, 2001.

[140] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado Aaron Visaggio, Gerardo Canfora, and Harald C. Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In *International Conference on Software Maintenance and Evolution (ICSME)*, 2015.

[141] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Association for Computational Linguistics (ACL)*, 2002.

[142] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[143] Matt Post. A call for clarity in reporting BLEU scores. In *Conference on Machine Translation (WMT): Research Papers*, 2018.

[144] Michael Pradel and Koushik Sen. Deepbugs: a learning approach to name-based bug detection. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2018.

[145] James Pustejovsky, Robert Ingria, Roser Saurí, José M. Castaño, Jessica Littman, Robert J. Gaizauskas, Andrea Setzer, Graham Katz, and Inderjeet Mani. The specification language timeml. In *The Language of Time - A Reader*. 2005.

[146] Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. Stanza: A python natural language processing toolkit for many human languages. In *Association for Computational Linguistics (ACL): System Demonstrations*, 2020.

[147] Likun Qiu, Yue Zhang, and Yanan Lu. Syntactic dependencies and distributed word representations for analogy detection and mining. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2015.

[148] Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting program properties from "big code". In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2015.

[149] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Workshop on New Challenges for NLP Frameworks*, 2010.

[150] Xin Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.

[151] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. Dropedge: Towards deep graph convolutional networks on node classification. In *International Conference on Learning Representations (ICLR)*, 2020.

[152] Aurko Roy, Rohan Anil, Guangda Lai, Benjamin Lee, Jeffrey Zhao, Shuyuan Zhang, Shibo Wang, Ye Zhang, Shen Wu, Rigel Swavely, Tao Yu, Phuong Dao, Christopher Fifty, Zhifeng Chen, and Yonghui Wu. N-grammer: Augmenting transformers with latent n-grams. *arXiv preprint arXiv:2207.06366*, 2022.

[153] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: scaling code clone detection to big-code. In *International Conference on Software Engineering (ICSE)*, 2016.

[154] Frank Schilder and Christopher Habel. Temporal information extraction for temporal question answering. In *New Directions in Question Answering*, 2003.

[155] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Association for Computational Linguistics (ACL)*, 2016.

[156] Andrea Setzer and Robert Gaizauskas. A pilot study on annotating temporal relations in text. In *Workshop on Temporal and spatial information processing (TASIP)*, 2001.

[157] Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process*, 2013.

[158] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Bram Adams, Ahmed E. Hassan, and Patrick Martin. Assisting developers of big data analytics applications when

deploying on hadoop clouds. In *International Conference on Software Engineering (ICSE)*, 2013.

[159] Weiyi Shang, Meiyappan Nagappan, and Ahmed E. Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering (EMSE)*, 2015.

[160] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *International Conference on Automated Software Engineering (ASE)*, 2010.

[161] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research (JMLR)*, 2014.

[162] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. Flow2vec: value-flow-based precise code embedding. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2020.

[163] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *International Conference on Software Maintenance and Evolution (ICSME)*, 2014.

[164] Mark D Syer, Zhen Ming Jiang, Meiyappan Nagappan, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. Leveraging performance counters and execution logs to diagnose memory-related performance issues. In *International Conference on Software Maintenance (ICSME)*, 2013.

[165] Yiming Tang, Allan Spektor, Raffi Khatchadourian, and Mehdi Bagherzadeh. Automated evolution of feature logging statement levels using git histories and degree of interest. *Science of Computer Programming (SCP)*, 2022.

[166] Bart Theeten, Frederik Vandeputte, and Tom Van Cutsem. Import2vec learning embeddings for software libraries. In *International Conference on Mining Software Repositories (MSR)*, 2019.

[167] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering (EMSE)*, 2010.

[168] Jörg Tiedemann and Yves Scherrer. Neural machine translation with extended context. In *Workshop on Discourse in Machine Translation (DiscoMT)*, 2017.

[169] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Deep learning similarities from different representations of source code. In *International Conference on Mining Software Repositories (MSR)*, 2018.

[170] Shikhar Vashishth, Manik Bhandari, Prateek Yadav, Piyush Rai, Chiranjib Bhattacharyya, and Partha P. Talukdar. Incorporating syntactic and semantic information in word embeddings using graph convolutional networks. In *Association for Computational Linguistics (ACL)*, 2019.

[171] Mario Linares Vásquez, Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. On using machine learning to automatically classify software applications into domain categories. *Empirical Software Engineering (EMSE)*, 2014.

[172] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[173] Marc Verhagen, Robert J. Gaizauskas, Frank Schilder, Mark Hepple, Graham Katz, and James Pustejovsky. Semeval-2007 task 15: Tempeval temporal relation identification. In *International Workshop on Semantic Evaluations (SemEval)*, 2007.

[174] Elena Voita, Pavel Serdyukov, Rico Sennrich, and Ivan Titov. Context-aware neural machine translation learns anaphora resolution. In *Association for Computational Linguistics, (ACL)*, 2018.

[175] Elena Voita, Rico Sennrich, and Ivan Titov. When a good translation is wrong in context: Context-aware machine translation improves on deixis, ellipsis, and lexical cohesion. In *Association for Computational Linguistics (ACL)*, 2019.

[176] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. Context-aware retrieval-based deep commit message generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2021.

[177] Kai Wang, Xiaojun Quan, and Rui Wang. Biset: Bi-directional selective encoding with template for abstractive summarization. In *Association for Computational Linguistics (ACL)*, 2019.

[178] Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embeddings for program repair. In *International Conference on Learning Representations (ICLR)*, 2018.

[179] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *International Conference on Software Engineering (ICSE)*, 2016.

[180] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

[181] Huihui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

[182] Laura Wendlandt, Jonathan K. Kummerfeld, and Rada Mihalcea. Factors influencing the surprising instability of word embeddings. In *North American Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2018.

[183] Lesly Miculicich Werlen, Dhananjay Ram, Nikolaos Pappas, and James Henderson. Document-level neural machine translation with hierarchical attention networks. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[184] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *International Conference on Automated Software Engineering (ASE)*, 2016.

[185] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019.

[186] Frank Wilcoxon. *Individual Comparisons by Ranking Methods.* 1992.

[187] Sam Wiseman, Stuart M. Shieber, and Alexander M. Rush. Learning neural templates for text generation. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[188] Shuangzhi Wu, Dongdong Zhang, Nan Yang, Mu Li, and Ming Zhou. Sequence-to-dependency neural machine translation. In *Association for Computational Linguistics (ACL)*, 2017.

[189] Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. Answerbot: Automated generation of answer summary to developersundefined technical questions. In *International Conference on Automated Software Engineering (ASE)*, 2017.

[190] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Online system problem detection by mining patterns of console logs. In *IEEE International Conference on Data Mining (ICDM)*, 2009.

[191] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[192] Jian Yang, Shuming Ma, Dongdong Zhang, Zhoujun Li, and Ming Zhou. Improving neural machine translation with soft template prediction. In *Association for Computational Linguistics (ACL)*, 2020.

[193] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. Log4perf: suggesting and updating logging locations for web-based systems' performance monitoring. *Empirical Software Engineering (EMSE)*, 2020.

[194] Yichun Yin, Furu Wei, Li Dong, Kaimeng Xu, Ming Zhang, and Ming Zhou. Unsupervised word and dependency path embeddings for aspect term extraction. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2016.

[195] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems (NeurIPS)*, 2014.

[196] Dongjin Yu, Quanxin Yang, Xin Chen, Jie Chen, and Yihang Xu. Graph-based code semantics learning for efficient semantic code clone detection. *Information and Software Technology (IST)*, 2023.

[197] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[198] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael Mihn-Jong Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[199] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *International Conference on Software Engineering (ICSE)*, 2012.

[200] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[201] Chen Zeng, Yue Yu, Shanshan Li, Xin Xia, Zhiming Wang, Mingyang Geng, Linxiao Bai, Wei Dong, and Xiangke Liao. degraphcs: Embedding variable-based flow graph for neural code search. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2023.

[202] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun (Peter) Chen. Studying the characteristics of logging practices in mobile apps: a case study on f-droid. *Empirical Software Engineering (EMSE)*, 2019.

[203] Jiacheng Zhang, Huanbo Luan, Maosong Sun, Feifei Zhai, Jingfang Xu, Min Zhang, and Yang Liu. Improving the transformer translation model with document-level context. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[204] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *International Conference on Software Engineering (ICSE)*, 2019.

[205] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[206] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Symposium on Operating Systems Principles (SOSP)*, 2017.

[207] Zaixiang Zheng, Xiang Yue, Shujian Huang, Jiajun Chen, and Alexandra Birch. Towards making the most of context in neural machine translation. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2020.

[208] Chen Zhi, Jianwei Yin, Shuiguang Deng, Maoxin Ye, Min Fu, and Tao Xie. An exploratory study of logging configuration practice in java. In *IEEE international conference on software maintenance and evolution (ICSME)*, 2019.

[209] Kuangqi Zhou, Yanfei Dong, Kaixin Wang, Wee Sun Lee, Bryan Hooi, Huan Xu, and Jiashi Feng. Understanding and resolving performance degradation in deep graph convolutional networks. In *ACM International Conference on Information & Knowledge Management (CIKM)*, 2021.

[210] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *International Conference on Software Engineering (ICSE)*, 2015.