Run Run Shaw Library

香港城市大學
City University of Hong Kong

Copyright Warning

Use of this thesis/dissertation/project is for the purpose of private study or scholarly research only. ***Users must comply with the Copyright Ordinance.***

Anyone who consults this thesis/dissertation/project is understood to recognise that its copyright rests with its author and that no part of it may be reproduced without the author's prior written consent.

CITY UNIVERSITY OF HONG KONG
香港城市大學

Code Representation Learning and Its Applications
代碼表示學習及其應用

Submitted to
Department of Computer Science
電腦科學系
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
哲學博士學位

by

YANG Zhen
楊振

August 2023
二零二三年八月

Abstract

Code representation learning has been an important technology to bridge various downstream tasks in the field of intelligent software engineering, such as software defect prediction, code summarization, code-comment synchronization, and code clone detection. The process of code representation learning can be partitioned into two stages, i.e., code representation and representation learning. Code representation can be constructed as token-based sequences, Abstract Syntax Trees (ASTs), control flow-based graphs, and expert-formulated features, while the learning algorithms of these representations are also diverse, including machine learning, deep learning, and reinforcement learning.

In this thesis, we focus on three specific downstream software engineering tasks, i.e., (1) smart contract vulnerability auditing, (2) code summarization, and (3) code-comment synchronization. Meanwhile, we propose corresponding code representation learning approaches to resolve their existing research challenges. To be specific, smart contract vulnerability auditing aims to detect vulnerability-prone modules before the execution of the smart contracts on the blockchain, thereby ensuring the quality and security of Decentralized Applications (DApps). Code summarization concentrates on automatically generating a piece of Natural Language Description (NLD) to present the functionality of a given code snippet, while code-comment synchronization targets to automatically

synchronize comments with code changes, which are both helpful for developers to comprehend code functionalities and maintain code repositories. Although numerous studies have been dedicated to the tasks above during the last several years, many pervasive and longstanding research challenges remain unsolved.

Towards smart contract vulnerability auditing, current data-driven approaches normally represent smart contract source code with a series of sequences according to only one tokenization standard, resulting in some of the semantic contexts could not be reflected within restricted sequence length. To address this limitation, we generate sequences from smart contracts in three tokenization standards (i.e., text only, structure only, and combining both above). Subsequently, we utilize the n-gram language model to capture semantic contexts respectively, and finally exploit our effective combination strategy of *Intersection* or *Union* to integrate the audited results from multiple semantic contexts. Consequently, code representation for smart contracts can be learned from multiple perspectives, and the accuracy of vulnerability diagnosis is also improved by a large margin.

In the area of code summarization, effectively representing source code and capturing its long-range dependencies have not been solved satisfactorily. Hence, we propose a Multi-Modal Transformer-based (MMTrans) code summarization approach. Specifically, MMTrans learns the representation of source code from the two heterogeneous modalities of the AST, i.e., Structure-Based Traversal (SBT) sequences and graphs. The SBT sequence provides the global semantic information of AST, while the graph convolution focuses on the local details. MMTrans uses two encoders to extract both global and local semantic information from the two modalities, respectively. Then it designs a joint decoder to generate code comments. Both encoders and the decoder employ the multi-head attention structure of the Transformer to enhance the ability of capturing the

long-range dependencies between code tokens. Therefore, MMTrans can learn the code representation well and take effect on the code summarization task.

For the Code-Comment Synchronization (CCS) task, solutions of a single type (i.e., either deep learning-based or heuristic-based) cannot successfully handle all kinds of samples in complex realistic situations. An intuitive rescue is allocating Code-Comment Inconsistent (CCI) samples to their suitable CCS models guided by an accurate classification. To this end, we propose a composite approach named CBS (i.e., Classifying Before Synchronizing) to further improve the code-comment synchronization performance, which combines the advantages of deep learning-based (i.e., CUP) and heuristic-based (i.e., HebCUP) models via the assistance of inferred categories of CCI samples. Specifically, we first define two categories (i.e., heuristic-prone and non-heuristic-prone) for CCI samples and propose five features to represent code changes and code-comment connections, thereby assisting category prediction. The samples whose comments can be correctly synchronized by HebCUP are heuristic-prone, while others are non-heuristic-prone. Then, CBS employs our proposed Multi-Subsets Ensemble Learning (MSEL) classification algorithm to alleviate the class imbalance problem and construct the category prediction model via learning from manual-crafted features. Next, CBS uses the trained MSEL to predict the category of the new sample. If the predicted category is heuristic-prone, CBS employs HebCUP to conduct the code-comment synchronization for the sample, otherwise, CBS allocates CUP to handle it. The key strength of CBS lies in its ability to accurately allocate CCI samples, which is attributed to the representation of code-comment changes based on manual-crafted features and the learning of this representation.

We have conducted extensive experiments that clearly demonstrate the superiority of our proposed approaches in each research area over their corresponding state-of-the-art

baselines by a significant margin. These results highlight the practicality of each approach for software development and maintenance. We hope that the promising outcomes of this thesis will attract interest and motivate further comprehensive research in these domains, ultimately leading to increased automation of software development and maintenance procedures and a substantial reduction in manual efforts.

Keywords: Code Representation Learning, Vulnerability Detection, Code Summarization, Code-Comment Synchronization, Natural Language Processing

CITY UNIVERSITY OF HONG KONG
Qualifying Panel and Examination Panel

Surname:	YANG
First Name:	Zhen
Degree:	PhD
College/Department:	Department of Computer Science

The Qualifying Panel of the above student is composed of:

Supervisor(s)

Prof. KEUNG Wai Jacky	Department of Computer Science City University of Hong Kong

Qualifying Panel Member(s)

Prof. WANG Shiqi	Department of Computer Science City University of Hong Kong
Prof. SONG Linqi	Department of Computer Science City University of Hong Kong

This thesis has been examined and approved by the following examiners:

Prof. LEUNG Wing Ho Howard	Department of Computer Science City University of Hong Kong
Prof. WONG Hau San Raymond	Department of Computer Science City University of Hong Kong
Prof. KEUNG Wai Jacky	Department of Computer Science City University of Hong Kong
Prof. LO David	School of Computing and Information Systems Singapore Management University

Declaration

I declare that the thesis entitled "*Code Representation Learning and Its Applications*" represents my own work, except where due acknowledgments are made, and that it has not been previously included in a thesis, dissertation, or report submitted to this university or any other institution for a degree, diploma, or other qualifications.

YANG Zhen

August 2023

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Prof. Wai Jacky KEUNG. Throughout my entire Ph.D. journey, he has provided me with immense support and guidance. I still remember when I was just a master's student, Prof. KEUNG gave me the opportunity to join his laboratory and guided me to start with the most basic research tasks. This commenced scientific research for me and allowed me to grow and improve continuously. Additionally, I would like to express my thanks to Prof. Zhi JIN and Prof. Ge LI for their inspiring guidance and for taking care of me during my visit to Peking University. With their help, I have gained a lot.

I would also like to thank my Ph.D. qualifying panel members, Prof. Shiqi WANG and Prof. Linqi SONG, for their valuable time and rigorous evaluation. Their constructive comments and probing questions have significantly enhanced the quality and breadth of my work. Furthermore, I am also very grateful to my dissertation examiners, Prof. David LO, Prof. Hau San Raymond WONG, and Prof. Wing Ho Howard LEUNG, for their invaluable feedback and suggestions on my thesis. Their constructive insights have inspired me and helped me improve the quality of this thesis.

My sincere thanks also go to my friends and lab mates, including Dr. Xiao YU, Dr. Yan XIAO, Dr. Miao ZHANG, Dr. Shuo FENG, Dr. Md Alamgir KABIR, Ms. Xiaoxue MA, Ms. Yishu LI, Mr. Pak Yuen Patrick CHAN, Ms. Jingyu ZHANG, Mr. Shuo LIU,

and Ms. Yihan LIAO, who provided me with comfort during my moments of despair and celebrated my achievements with me. Pursuing Ph.D. is a journey that is long yet fleeting, filled with passion but also tediousness, and I would not have been able to make it this far without their companionship. I hope our friendship will last and we can look forward to the future together.

Finally, I would like to express my heartfelt gratitude to my family. I am very grateful for their constant support and encouragement, and I will continue to work hard and not let them down.

It is true that the achievements already obtained are encouraging. However, compared with the future, they are insignificant.

- See you in the next decade. -

List of Publications

Published and Under Reviewed Journal Papers

1. **Yang Zhen**, Keung Wai Jacky, Yu Xiao, Xiao Yan, Jin Zhi, Zhang Jingyu. On the Significance of Category Prediction for Code-Comment Synchronization. ACM Transactions on Software Engineering and Methodology, 2023, 32(2): 1-41.

2. Zhang Fengji, Yu Xiao, Keung Wai Jacky, Li Fuyang, Xie Zhiwen, **Yang Zhen**, Ma Caoyuan & Zhang Zhimin. Improving Stack Overflow question title generation with copying enhanced CodeBERT model and bi-modal information. Information and Software Technology, 2022, 148: 1-13.

3. Ma Xiaoxue, Keung Wai Jacky, **Yang Zhen**, Yu Xiao, Li Yishu, Zhang Hao. CASMS: Combining clustering with attention semantic model for identifying security bug reports. Information and Software Technology, 2022, 147: 1-14.

4. **Yang Zhen**, Keung Wai Jacky, Kabir Md Alamgir, Yu Xiao, Tang Yutian, Zhang Miao, Feng Shuo. AComNN: Attention enhanced Compound Neural Network for financial time-series forecasting with cross-regional features. Applied Soft Computing, 2021, 111: 1-13.

5. **Yang Zhen**, Keung Wai Jacky, Sun Zeyu, Zhao Yunfei, Li Ge, Jin Zhi, Liu Shuo, and

Li Yishu. Few-shot Learning for Neural Code Generation. Information and Software Technology. (Under Review)

Published Conference Papers

1. Wei Zhengyuan, Wang Haipeng, **Yang Zhen**, Chan Wing-Kwong. SEbox4DL: A Modular Software Engineering Toolbox for Deep Learning Models. In the IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). IEEE, 2022: 193-196.

2. **Yang Zhen**, Keung Wai Jacky, Yu Xiao, Gu Xiaodong, Wei Zhengyuan, Ma Xiaoxue & Zhang Miao. A Multi-Modal Transformer-based Code Summarization Approach for Smart Contracts. In the IEEE/ACM 29th International Conference on Program Comprehension (ICPC). IEEE/ACM, 2021: 1-12.

3. **Yang Zhen**, Keung Wai Jacky, Zhang Miao, Xiao Yan, Huang Yangyang, Hui Tik. Smart contracts vulnerability auditing with multi-semantics. In the IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC). IEEE, 2020: 892-901.

Contents

Abstract **i**

Acknowledgements **vii**

List of Publications **ix**

1 Introduction **1**

1.1 Smart Contract Vulnerability Auditing 2

1.2 Code Summarization 4

1.3 Code-Comment Synchronization 7

1.4 Contributions of the Thesis 11

1.5 Organization of the Thesis 13

2 Literature Review **14**

2.1 Smart Contract Vulnerability Auditing 14

2.2 Code Summarization 15

2.3 Code-Comment Synchronization 16

2.4 Comment Classification 17

2.5 Inconsistent Comment Detection 18

3 Smart Contracts Vulnerability Auditing With Multi-Semantics 20

3.1 Background . . . 21

3.1.1 Ethereum Smart Contracts and Vulnerabilities . . . 21

3.1.2 N-gram Language Model . . . 22

3.2 Methodology . . . 23

3.2.1 Model Framework Overview . . . 23

3.2.2 Tokenization . . . 25

3.2.3 Model Parameters and Thresholds . . . 26

3.2.4 Combination Strategies . . . 29

3.3 Experiments . . . 33

3.3.1 Experimental Design and Evaluation Metrics . . . 33

3.3.2 Dataset Collection and Construction . . . 33

3.3.3 Experimental Devices . . . 34

3.3.4 Optimization for the NLM . . . 36

3.3.5 Experiments on Combination Strategies . . . 37

3.3.6 Comparison with the Baseline . . . 43

3.4 Threats to Validity . . . 44

3.5 Chapter Summary . . . 45

4 MMTrans: Multi-Modal Analysis with Transformer Structures 46

4.1 Preliminary . . . 47

4.1.1 Structure-based Traversal . . . 47

4.1.2 Graph Convolutional Neural Network . . . 48

4.1.3 Transformer-related Structures . . . 49

4.1.3.1 Positional Encoding . . . 49

4.1.3.2 Multi-head Attention . . . 50

4.1.3.3 Point-Wise Feed-Forward Networks 51
4.2 Approach 51
4.2.1 Graph Encoder 54
4.2.2 SBT Encoder 55
4.2.3 Joint Decoder 56
4.3 Data Preparation 57
4.3.1 Preprocessing 57
4.3.2 Data Transformation 58
4.3.3 Generating Vocabularies and Input Pipeline 59
4.4 Experimental Setup 60
4.4.1 Research Questions 60
4.4.2 Methodology 62
4.4.3 Baselines 63
4.4.4 Metrics 65
4.4.5 Experimental Device 66
4.5 Results 66
4.5.1 RQ1: Quantitative Evaluation 67
4.5.2 RQ2: Head Number Analysis 68
4.5.3 RQ3: Dimensionality Analysis 69
4.5.4 RQ4: Strength of the MMTrans 70
4.6 Theats to Validity 75
4.7 Chapter Summary 76

5 CBS: Classifying Before Synchronizing for Code-Comment Inconsistencies 77

5.1 Background 79
5.1.1 Problem Formalization 79

5.1.2 Usage Scenario 80
5.1.3 Current Code-Comment Synchronization Approaches 80
5.1.4 An Example of Code-Comment Synchronization 81
5.2 Exploring Existing Code-Comment Synchronization Methods 83
5.2.1 Dataset 83
5.2.2 Evaluation Metrics for Code-Comment Synchronization 88
5.2.3 Experimental Design 90
5.2.4 Results and Discussion 92
5.3 Methodology 98
5.3.1 Motivation 98
5.3.2 Approach Overview 99
5.3.3 Category Definition and Partition Criteria 103
5.3.4 The Proposed Features 103
5.3.5 Multi-Subsets Ensemble Learning 109
5.4 Experiments for Category Prediction 110
5.4.1 Experimental Setup 113
5.4.2 Evaluation Metrics for Category Prediction 115
5.4.3 Experimental Results 116
5.5 Experiments for Code-Comment Synchronization 118
5.5.1 Experimental Setup 118
5.5.2 Evaluation Metrics 119
5.5.3 Experimental Results 120
5.6 Further Exploration For CBS 123
5.6.1 RQ1: How do other classifiers influence the performance of CBS? . 125

5.6.2 RQ2: How does the Multi-Subset Ensemble Learning (MSEL) influence the performance of CBS? . . . 128
5.6.3 RQ3: How do the proposed features influence the performance of CBS? . . . 130
5.7 Discussion . . . 135
5.7.1 Where Does CBS Fail . . . 135
5.7.2 Implications . . . 137
5.8 Threats to Validity . . . 139
5.8.1 Internal validity . . . 139
5.8.2 External Validity . . . 140
5.8.3 Construct Validity . . . 141
5.9 Chapter Summary . . . 142

6 Conclusions and Future Work **143**
6.1 Conclusion . . . 143
6.2 Future Work . . . 145

Bibliography **147**

List of Tables

3.1 The parameter and threshold combination of each optimized NLM for three kinds of sequence types under different sequence lengths 35

3.2 The performance of each optimized NLM for three kinds of sequence types under different sequence lengths 35

3.3 The Precision before and after the *Intersection* strategy under each sequence length 37

3.4 The improvement on Precision after the *Intersection* strategy under each sequence length 38

3.5 The Precision, Recall, and F-Measure before and after the *Union* strategy under different sequence length 39

3.6 Improvement in Precision, Recall, and F-Measure after the *Union* strategy under different sequence length 39

3.7 The performance of baseline and MSgram under different sequence length . 42

3.8 The improvement of the MSgram under different sequence length 42

4.1 Statistics for data lengths 59

4.2 Automated metrics evaluation results for the baselines and MMTrans 66

4.3 Examples of generated comments by each approach 71

4.4 An example for illustrating attention matrices 74

5.1 The statistics of the datasets 85
5.2 The code-comment synchronization performance of HebCUP and CUP on the validation set 91
5.3 Category distribution of code-comment inconsistent samples in the training and validation sets 97
5.4 The feature values of the CCI sample shown in Listing 5.1 106
5.5 The description of our proposed five features 108
5.6 Hyper-parameter tuning for category prediction 115
5.7 The evaluation metrics for category prediction. 116
5.8 The performance of the MSEL algorithm with different base classifiers across the 10-fold cross-validation. 118
5.9 The performance of our proposed CBS approach on the testing set 124
5.10 The performance of our proposed CBS approach and baselines on the testing set 124
5.11 The category prediction results of the MSEL algorithm with different base classifiers on the testing set 126
5.12 The code-comment synchronization performance of CBS with different base classifiers on the testing set 127
5.13 The experimental results for exploring the effect of MSEL algorithm 131
5.14 Top performing feature set of each level evaluated via 10-fold cross-validation 131
5.15 The examples where CBS fails 135

List of Figures

3.1 The framework overview of MSgram 24
3.2 The function “sample” and its AST 24
3.3 Sequence representations of function “sample” 27
3.4 The performance of each optimized NLM for three kinds of sequence types under different sequence lengths 34
3.5 The improvements on Precision of Step 1, Step 2 under *Intersection* strategy and this strategy’s final improvement 38
3.6 The improvement on Precision, Recall, and F-Measure of Step 1 and Step 2 under *Union* strategy and this strategy’s final improvement 40
3.7 The performance of baseline and MSgram 43

4.1 An example of smart contract method (_tokensToSell) with its AST (graph) and SBT sequence 49
4.2 The overall framework of MMTrans 52
4.3 (Self) Multi-head attention module 53
4.4 The performance variation of MMTrans on each evaluation metric with the tuning of dimensionality 69
4.5 An attention matrix example 74

5.1 The code change sequence of the code-comment co-change instance on sub-sequence-level 86
5.2 The code change sequence of the code-comment co-change instance on element-level 86
5.3 (A) The Venn graph for the distributions of samples with correctly synchronized comments after being handled by CUP and HebCUP within the first attempt, (B) The Venn graph for the distributions of samples with correctly synchronized comments after being handled by CUP and HebCUP within five attempts, (C) The Venn graph for distributions of samples with $ED_diff < 0$ after being handled by CUP and HebCUP . . . 92
5.4 The density plot for the sample distributions of $ED_diff \neq 0$ after being handled by CUP and HebCUP 93
5.5 The preferences of HebCUP and CUP on samples with different percentages of “replace”, “insert”, and “delete” 96
5.6 The overview of the CBS approach 101
5.7 A running example to show how CBS works 102
5.8 The distributions of heuristic-prone and non-heuristic-prone samples among each proposed feature 108
5.9 The overview of the Multi-Subsets Ensemble Learning (MSEL) algorithm . 111
5.10 The category prediction results on each evaluation metric among different feature levels 130
5.11 The code-comment synchronization performance on each evaluation metric among different feature levels 132

Chapter 1

Introduction

In the intelligent software engineering field, the effective application of probabilistic models on various code-related tasks relies heavily on the selection of appropriate code representations and the designing of powerful learning algorithms, which is also the essence of code representation learning [9, 101, 6, 50, 41, 24, 32, 140]. Code representations can be constructed in a wide variety of formats, such as token-based sequences [2, 47], Abstract Syntax Trees (ASTs) [43, 44], control-flow graphs [84, 78], and expert features [86, 67]. On the other hand, learning algorithms towards aforementioned representations are also diverse, including machine learning [82, 137], deep learning [62, 77], and reinforcement learning [111, 127]. Different code-related tasks require different (combinations of) code representation formats and learning algorithms. For example, Phan et al. [121] adopted control-flow graphs to represent source code and designed deep neural networks to identify program defects. Bui et al. [14] represented source code in ASTs and incorporated tree-based convolutional neural networks [83] to implement the code classification task.

In this thesis, we concentrate on three code-related downstream tasks, i.e., smart

contract vulnerability auditing, code summarization, and code-comment synchronization. Meanwhile, we propose corresponding code representation learning approaches to resolve their implied research challenges.

1.1 Smart Contract Vulnerability Auditing

Smart contracts, as the backend logics of Decentralized Applications (DApp), are a series of automatically executed programs written by a Turing-complete programming language (e.g., solidity), first applied on Ethereum in around 2013 [131]. They operate on top of the consensus protocol of the blockchain network for the purpose of ensuring the trading to complete between different parties without trust, by which people can remove the traditional central entities that manage the transaction network and operate their businesses directly peer-to-peer, thereby simplifying the transaction procedure. As one of the most popular blockchain platforms, Ethereum has been growing to a huge volume: The total Ether supply and market capitalization on Ethereum have been up to 100 Million Ether and 18 Billion USD respectively. Nevertheless, due to the unfamiliarity with smart contract development and its peculiar operation mechanism (e.g., immutability, trusted transactions, anonymity), a great number of potential risks appeared in those DApps operating on Ethereum, such as the infamous DAO attack in 2016, leading to a 60 million USD loss [64, 7, 16]. As such, detecting vulnerabilities of smart contracts before deploying them on the blockchain has been becoming a vitally critical research area.

Current data-driven approaches (e.g., S-gram [70]) adopt the tree-based format (e.g., AST) and tokenize the tree structure into sequences to represent smart contract source code. Afterward, they utilize the N-gram Language Model (NLM) to learn representations for vulnerability auditing. Nevertheless, most previous studies only adopt one kind of

tokenization standard in sequence generation, leading to the ignorance of some semantic contexts within a restricted sequence length.

Consequently, we propose an improved model, namely Multi-Semantic gram (MSgram), based on the S-gram to further boost its performance by analyzing sequences of multiple semantics. Basically, we generate sequences in three ways for the multi-semantic representation of smart contracts: 1) *Text sequence*s, covering only text information, are composed of leaf nodes in ASTs. 2) *Structure sequence*s, covering only structure information, focus on non-leaf nodes in ASTs when tokenization. 3) *Combined sequence*s, covering both structure and text information of a smart contract, are complete AST representations. To construct the sequences of the above three versions, we modify the Structure-Based Traversal (SBT) proposed by Hu et al. [43].

Subsequently, we feed each kind of sequence into their corresponding trained NLMs respectively to capture their respective semantics, thereby obtaining separate audited results. For each smart contract, we can receive three audited results from the above analysis. After that, we adopt a combination strategy of *Intersection* or *Union* on these three separate audited results to integrate them together, by which the final audited result can be obtained. In this way, developers can repair bugs according to the auditing results of MSgram.

We divide the experiment into two phases, i.e., the model construction phase and the security auditing phase. In the model construction phase, we optimize each NLM separately and assess combination strategies to obtain the best setting of MSgram based on the training set. Then in the auditing security phase, we build the MSgram with the pre-configured setting in the construction phase for vulnerability auditing and make a comparison with the baseline (i.e., Sgram) on the testing set.

We evaluate our proposed approach on 7,200 Ethereum smart contracts gathered from

Etherscan.io [1]. The experimental results demonstrate MSgram achieves a high Precision score up to 51.79% (48.02% on average) on the testing set when we make an *Intersection* combination. However, with the *Union* combination for capturing more true vulnerabilities, we obtain another group of final audited results with a Recall score up to 66.02% (58.57% on average) and an F-Measure score up to 42.11% (34.06% on average) on the testing set. Compared with the baseline model, i.e., Sgram, the final audited results with the *Intersection* strategy can improve the Precision score by up to 106.59% (44.24% on average). For the auditing result with the *Union* strategy, it can improve the Recall score by 360.55% at most (306.50% on average) and the F-Measure score by 271.02% at most (224.22% on average).

1.2 Code Summarization

With the rapid development of scale and complexity of software systems, program comprehension has been consuming tremendous efforts of developers during their software construction and maintenance [105]. Xia et al.'s study [133] reveals that software developers devote an average of 58% of their time to program comprehension, where low-quality code comments are crucial factors. Nonetheless, maintaining code comments is a labor-intensive task. Consequently, the need for automating this process has become more urgent, leading to the emergence of a critical research area known as code summarization [80, 34]. Currently, the challenges of automatic code summarization usually include:

(1) How to extract the semantic information of source code. Early researchers, such as Iyer et al. [47] and Loyola et al. [74], used the plain source code as the input of

[1] https://etherscan.io/

the code summarization model, which ignored the structural information of source code. Therefore, most recent works firstly parsed source code to the AST, and then extracted the semantic information of code. For example, Hu et al. [44] firstly traversed ASTs to SBT sequences, then used the plain source code and the SBT sequences as inputs to represent the semantic information of source code. LeClair et al. [60] regarded ASTs as graphs, and used the Graph Neural Network (GNN)-based encoder to model ASTs, combined with the RNN-based encoder to model the plain source code. However, ASTs can be represented as multiple modalities, such as SBT sequences and graphs, each focusing on a distinct aspect of the semantic information. Therefore, it is not comprehensive to use a single AST modality to represent the semantic information of code.

(2) How to capture the long-range dependencies between code tokens. Our experiments are conducted on smart contracts. Based on our preliminary statistics, the source code of their methods can be very long. To be specific, smart contract methods contain 58.72 code tokens on average, and the longest one contains 3,272 code tokens. Previous works often applied the Recurrent Neural Network (RNN), Long Short Term Memory (LSTM), or Gated Recurrent Unit (GRU) models to extract features from their inputs, which have shown to be difficult to capture the long-range dependencies between code tokens [120, 2].

In order to address the above challenges, we propose a Multi-Modal Transformer-based (MMTrans) code summarization approach. Firstly, MMTrans represent the semantic information of source code from the two modalities of the AST, i.e., the SBT sequence and the graph. Specifically, the SBT sequence is globally parsed from the AST using the SBT method, which involves the global semantic information of the source code. Meanwhile, MMTrans regards the AST as a graph and employs the Graph Convolutional Neural Network (GCN) to learn representations of nodes based on

their neighboring nodes, thus obtaining the local semantic information of source code. Then, the MMTrans uses a dual-encoder architecture: an SBT encoder for encoding SBT sequences to learn the **global** semantic information, and a graph encoder for encoding graphs to learn the **local** semantic information. Finally, the MMTrans uses a joint decoder to decode the outputs of the two encoders and previously generated words to produce each time step's prediction. In addition, both the encoders and the decoder employ the multi-head attention structure of the Transformer to reinforce the capability of capturing the long-range dependencies between code tokens.

To evaluate the MMTrans, we carefully collect 347,410 <method, comment> pairs from 40,932 smart contracts on Etherscan.io [2], one of the most popular and active smart contract communities. We extensively assess the performance of the MMTrans against the three recently proposed approaches (i.e., Hybrid-DeepCom [44], code+gnn+GRU [60], and Vanilla-Transformer [2]) on the dataset in terms of sentence-level BLEU (S-BLEU), corpus-level BLEU (C-BLEU), ROUGE-LCS F1 and METEOR. The experimental results show that the MMTrans performs better than the baselines by 17.23%-58.45% in terms of S-BLEU, by 20.74%-62.49% in terms of C-BLEU, by 6.78%-55.60% in terms of ROUGE-LCS F1, and by 10.17%-45.50% in terms of METEOR. We further conduct two groups of ablation experiments to explore the strength of MMTrans. Both the quantitative and instance analysis demonstrate that the MMTrans can generate higher-quality comments for smart contracts.

[2]https://etherscan.io/

1.3 Code-Comment Synchronization

With the increasing frequency of software product iteration, developers often ignore or forget to synchronize comments when the corresponding code goes through changes, leading to inconsistent or obsolete comments (i.e., bad comments) [99, 72, 112, 129, 113]. Bad comments consume the effort of software developers in double-checking the code implementation and lead to the unintended injections of software bugs [46, 93, 115], which undoubtedly delay the project development and hinder software maintenance. If comments can be correctly synchronized when software developers change the source code, bad comments can be reduced or even avoided. To this end, code-comment synchronization become an emerging research field, which is intended to synchronize comments with the changes of corresponding code snippets.

In recent years, several approaches have been proposed, which can be divided into two categories: deep learning-based approach (e.g., CUP [73]) and heuristic-based approach (e.g., HebCUP [66]). The former constructs a Neural Machine Translation (NMT)-structured semantic model, while the latter designs a series of heuristic rules for token-level replacements on old comments. In addition, they both act on the code changes with bad comments left which are also referred to as Code-Comment Inconsistent (CCI) samples. However, CCI samples are complicated in practice. For example, some tokens that need to be updated in comments do not appear in the corresponding code change contents, which makes it impossible for the heuristic-based approach to design hand-crafted heuristic rules to synchronize comments correctly [66]. Certain code snippets contain some high-frequency tokens that are not related to the code changes, which causes the deep learning-based approach to be distracted and ignore the key information of code changes during code-comment synchronization [139, 124, 28]. On the other hand, approaches of different kinds carry heterogeneous characteristics.

Typically, deep learning-based approaches have a more generalized capability of synchronizing comments with software evolution and growth. And heuristic-based approaches can generate completely correct comments for CCI samples fully covered by their fine-designed heuristic rules.

Therefore, towards the state-of-the-art deep learning-based and heuristic-based approaches (i.e., CUP and HebCUP), we first experimentally verify their different capabilities on an open-source code-comment synchronization dataset, which is extracted from 1,496 Java repositories [73]. The dataset has been split into three parts, i.e., 80,591 samples in the training set, 8,827 samples in the validation set, and 9,204 samples in the testing set. Based on the empirical study in Section 5.2, we find that CUP and HebCUP indeed have their own unique advantages, and neither can dominate all evaluation metrics because these metrics assess code-comment synchronization approaches from different aspects. HebCUP can correctly synchronize more CCI samples than CUP does, therefore it defeats CUP in terms of Accuracy. On the contrary, CUP has a more generalized capability due to its deep learning-based structure, causing it to defeat HebCUP in terms of Average Edit Distance (AED), Relative Edit Distance (RED), and BLEU-4. In addition, owing to its beam search technique, CUP also performs better in terms of Recall@5. We also find that both HebCUP and CUP have some distinctive CCI samples that the other side cannot handle but can be better solved by themselves. In particular, HebCUP prefers to handle CCI samples that their code changes contain more token replacements and fewer token insertions or deletions. However, CUP is not overly dependent on the edit actions of CCI samples. It is noteworthy that CUP outperforms HebCUP in terms of AED and RED, but has fewer CCI samples whose edit distances are reduced after the synchronization. This finding motivates us to propose a new evaluation metric, the Effective Synchronized

Sample (ESS) ratio, to complement AED and RED, which measures the ratio of CCI samples whose edit distances are reduced after being handled by synchronizers.

Upon the analysis above, we argue that combining the advantages of HebCUP and CUP can be a rational attempt to further boost the code-comment synchronization performance, which is still blank in this field and worth trying. An intuitive solution is to design a classifier to predict the proneness of each CCI sample on synchronizers (i.e., either CUP or HebCUP) and assign each of them to one of the synchronizers according to the predicted proneness. Therefore, each of the approaches is only responsible for the samples that they can handle better, and the overall performance of the code-comment synchronization can be further improved.

Nevertheless, there are some research challenges that need to be resolved: (1) How to accurately predict the proneness of a CCI sample in advance between the approaches of HebCUP and CUP? There is no previous study on category prediction for CCI samples. Besides, it is also hard to define or represent the internal relation patterns between CCI samples and their model proneness. (2) How to deal with the class imbalance problem in the proneness prediction? Based on our preliminary statistics, comments of only 23.83% of CCI samples in the training set can be correctly synchronized by the HebCUP, which means the proneness prediction is a class imbalance problem. Simply adopting normal classifiers may bias the overall prediction performance, which is also a challenge that needs to be resolved.

We take the first step towards constructing a composite approach named CBS (i.e., Classifying Before Synchronizing), which combines the advantages of CUP and HebCUP with the assistance of inferred categories of CCI samples. Firstly, we define two categories for CCI samples, i.e., the samples that can be correctly synchronized by HebCUP are heuristic-prone, while others are non-heuristic-prone. To improve the classification

performance, we propose five manually-crafted features to represent code changes and their interaction with old comments, thereby modeling the internal relation patterns between CCI samples and their model proneness. Then, since there are more non-heuristic-prone samples than heuristic-prone ones in the dataset, we propose a Multi-Subsets Ensemble Learning (MSEL) classification algorithm to alleviate the class imbalance problem and construct the category prediction model, i.e., the representation learning. Next, CBS employs the trained classification model to predict the category of the new CCI sample. Finally, if the predicted category is heuristic-prone, CBS adopts HebCUP to synchronize the comment of the sample, otherwise, CBS assigns CUP to handle the sample.

We evaluate CBS and the two baselines (i.e., CUP and HebCUP) on the testing set in terms of Accuracy, Recall@5, AED, RED, BLEU-4, and ESS ratio. The experimental results demonstrate that CBS, on average, achieves an Accuracy score of 27.90%, a Recall@5 score of 35.67%, an AED score of 3.562, a RED score of 0.937, a BLEU-4 score of 73.16, and a ESS ratio of 35.85%. In addition, CBS outperforms HebCUP by 8.53% in terms of Accuracy, by 35.16% in terms of Recall@5, by 4.85% in terms of AED and RED, by 1.67% in terms of BLEU-4, and by 12.09% in terms of ESS ratio. CBS also performs better than CUP by 38.41%, 10.52%, 1.24%, 1.24%, 1.62%, and 26.69% in terms of the six evaluation metrics, respectively. Subsequently, we further discuss the performance of CBS with different base classifiers, the effect of the MSEL algorithm, and investigate the impact of the proposed five features. In summary, according to our extensive experiments and discussion, the category prediction for CCI samples can indeed boost the performance of code-comment synchronization via combining it with current approaches.

1.4 Contributions of the Thesis

The main contributions of this thesis are as follows:

- In the area of smart contract vulnerability auditing, we propose a novel concept of integrating multiple semantics in smart contract vulnerability auditing with the N-gram Language Model.
- We design three tokenization standards by modifying the SBT Traversal based on structure only, text only, and combined version for code representation learning in multi-semantics.
- We explore the combination strategies (i.e., *Intersection* and *Union* strategies) of multi-semantics and empirically study the performance and their pros and cons.
- In the area of code summarization, we propose a novel Multi-Modal Transformer-based (MMTrans) code summarization approach for smart contracts, which can represent both the global and local semantic information of code, and learn the long-range dependencies between code tokens to generate higher-quality comments.
- We build a dataset with totally 347,410 <method, comment> pairs for the field of smart contract code summarization. To the best of our knowledge, it is the first large-scale dataset for this task. And we open source our replication package, including the dataset [3] and the source code [4] of MMTrans for follow-up works.
- We extensively evaluate the MMTrans on the dataset using four universally applied metrics and further explore its strength through the ablation experiment. Based on

[3] https://zenodo.org/record/4587089#.YEMmWugzYuU
[4] https://github.com/yz1019117968/ICPC-21-MMTrans

both the statistic and instance analysis, MMTrans is shown to outperform the state-of-the-art models by a significant margin.

- In the area of code-comment synchronization, we systematically investigate the current state-of-the-art code-comment synchronization approaches of heuristic-based and deep learning-based, which verifies the heterogeneous advantages of each approach and the existence of model proneness among CCI samples.

- We define two categories (i.e., heuristic-prone and non-heuristic-prone) for CCI samples and propose a CBS approach to further improve the performance of code-comment synchronization models, which demonstrates how category prediction of CCI samples can benefit the code-comment synchronization.

- We propose five manually-crafted features to represent code changes and their interaction with old comments, which facilitates the modeling of internal relation patterns between CCI samples and their model proneness. Besides, we design a Multi-Subsets Ensemble Learning (MSEL) algorithm to alleviate the class imbalance problem in category prediction. We open source the code of our CBS approach [5] to facilitate future research and application.

- Extensive experiments are conducted to evaluate CBS, and a new evaluation metric named Effective Synchronized Sample (ESS) ratio is proposed to make the assessment of this work more comprehensive. This work highlights that category prediction for CCI samples can make code-comment synchronization more practical in the real development scenario and sheds light on future directions for both industry and academia in this field.

[5] https://github.com/yz1019117968/TOSEM-22-CBS

1.5 Organization of the Thesis

The rest of this thesis is organized as follows.

In Chapter 2, we review related work towards this thesis, including smart contract vulnerability auditing, code summarization, comment classification, inconsistent comment detection, and code-comment synchronization.

In Chapter 3, we introduce MSgram, an n-gram language model-based multi-semantic representation and learning framework for smart contract vulnerability auditing. Extensive experiments have been conducted on MSgram and demonstrated its superior vulnerability auditing capability compared with baselines.

In Chapter 4, we present MMTrans, a transformer-based code summarization approach, which integrates multi-modalities of ASTs. MMTrans has been experimented on smart contracts code summarization and presented a superior performance against baselines.

Chapter 5 describes CBS in detail. CBS carries a methodology of classifying before synchronizing. We experiment with CBS on a publicly available Java dataset, demonstrating the effectiveness of our proposal.

We conclude the thesis in Chapter 6, which includes the results that have been drawn from the research, main contributions, and suggestions for future work.

Chapter 2

Literature Review

In this chapter, we discuss a series of code-related downstream tasks that are relevant to this thesis, such as smart contract vulnerability auditing, code summarization, code-comment synchronization, comment classification, and inconsistent comment detection.

2.1 Smart Contract Vulnerability Auditing

Since the vulnerable nature of smart contracts, a lot of researchers have been devoting themselves to this field and proposed various attempts [92]. Barboni et al. [8] adopted mutation testing to detect smart contract vulnerabilities. Luu et al. [76] proposed a symbolic execution technique named Oyente to detect contracts' vulnerabilities based on several predefined patterns. Jiang et al. [48] put forward a tool, namely ContractFuzzer, with Application Binary Interface (ABI) specification-based fuzzing input relying on predefined test oracles. Similarly, Liu et al. [69] also adopted fuzzing techniques and introduced ReGuard but only targeted the reentrancy risk of smart contracts. The method based on predefined detecting patterns can indeed audit vulnerabilities with high precision but is also restricted by those patterns. Therefore, other researchers raised data-driven

approaches that mainly applied various language models. For example, Liu et al. [70] leveraged the n-gram language model to analyze smart contracts' vulnerabilities at the program statement level. However, they only adopted one kind of tokenization criterion for the language model to analyze, leading to the semantic representation not being comprehensive enough.

2.2 Code Summarization

Automatic code summarization approaches can be divided into two main categories, i.e., heuristic/template-driven approaches and AI/data-driven approaches [61]. Haiduc et al. [35, 34] applied text retrieval techniques to select some important keywords as the generated code summary. Eddy et al. [23] partially replicated and expanded Haiduc et al.'s approaches by exploiting a new hierarchical topic model. Sridhara et al. [106, 107] proposed a source code summarization approach for Java methods by using the software word usage model [40], which can automatically extract natural language phrases from source code identifiers.

Owing to the rapid development of deep learning technology, the recently proposed code summarization approaches are almost AI/data-driven approaches. Most of them [74, 75, 65, 43, 122, 61, 5, 25, 125, 44, 60, 139, 4] adopted the Sequence-to-Sequence (Seq2Seq) model with attention mechanism. The major difference between the approaches is the input to the Seq2Seq model. For example, Loyola et al. [74] input the plain source code into the Seq2Seq model, Lu et al. [75] used the API sequences as the input, and Fernandes et al. [25] used the graph representations of source code as the input. In addition, in order to extract more information from the source code, Hu et al. [45, 44] and LeClair et al. [61, 60] proposed some multi-input Seq2Seq models. For example, Hu et al. [44] input the

plain source code and the SBT sequences to the Seq2Seq model, in order to learn both the lexical and structural information from the source code and the AST. LeClair et al. [60] used the graph representation of the AST and plain source code as the inputs. However, it is not comprehensive that these works [43, 44, 60] used a single AST modality to represent the semantic information of code. In addition, the proposed Seq2Seq models [47, 74, 75, 65, 43, 122, 61, 5, 25, 125, 44, 60] mainly applied RNN, LSTM, or GRU to extract the code feature, which may fail to capture the long-range dependencies between code tokens. Therefore, Ahmad et al. [2] empirically investigated the advantage of using the Transformer model for the source code summarization task. However, they only adopted the plain source code as the single input, thus ignoring the structure information of the source code.

2.3 Code-Comment Synchronization

Code-comment synchronization is an emerging research field that started from the year of 2020. Until now, a total of three works [73, 90, 66] have been conducted to solve the research problem. Liu et al. [73] and Panthaplackel et al. [90] almost simultaneously for the first time proposed their own solution for code-comment synchronization with the code evolution. The former proposed an LSTM-based NMT model, which integrates the information from the old code, new code, and old comments to predict the corresponding new comments; while the latter proposed a similar GRU-based NMT model. Subsequently, Lin et al. [66] proposed HebCUP, for which they designed a series of heuristic rules to perform the token-level replacements based on old comments. Since Panthaplackel et al.'s [90] approach can only synchronize the comments of the functions with return statements, and their proposed heuristic features incorporated in their

approach cannot apply to other kinds of functions, which causes that their approach is not generalized enough, and applicable as the complementary of HebCUP and hard to compare with other approaches under the same criteria. Therefore, in our work, we only try to combine CUP and HebCUP to improve the code-comment synchronization performance. Both code summarization and code-comment synchronization focus on facilitating the program comprehension for developers, but the former generates code comments from scratch, targeting code snippets without any comments. However, the latter synchronizes pre-existing comments with corresponding code changes [73]. It is true that code summarization approaches can be applied for code-comment synchronization, but given more extra information, such as old comments and old code, code-comment synchronization approaches can generate much higher-quality comments than code summarization approaches, as verified by Panthaplackel et al. [90]. Therefore, in code-comment synchronization scenarios, code-comment synchronizes are more suitable. Thus their related studies are irreplaceable.

2.4 Comment Classification

Besides the source code, code comments have been considered the most essential software artifacts for program comprehension and maintenance [20, 103, 29, 30]. Nevertheless, comments with different characteristics have different goals and target audiences, and analyzing comments in a single manner hinders the empirical understanding of both comments and associated code [95, 94]. Researchers have proposed to classify code comments into different categories to help developers comprehend the source code better [94, 95, 138, 118, 98, 88]. Haouari et al. [37] proposed the taxonomy of comments and used it to conduct their analysis based on 39 programmer subjects and three open-source

projects. Steidel et al. [109] adopted machine learning methods to classify comments into seven categories, and further analyze and evaluate the quality of code comments. Zhai et al. [138] constructed a comprehensive comment taxonomy to classify comments into appropriate perspectives and granularity levels. In summary, despite the fact that comment classification has been proposed for many years and extensively applied in various fields of software engineering, classification for CCI samples has not been investigated yet. To the best of our knowledge, CBS is the first work to analyze the categories of CCI samples and their application in code-comment synchronization.

2.5 Inconsistent Comment Detection

Researchers have investigated the inconsistent comment detection to improve software maintainability and reduce bugs when code-comment evolves [141, 112, 114, 71, 99, 110, 115, 89]. Seminal works in this field are conducted by Tan et al. [112, 113, 114]. They first proposed an inconsistent comment detection approach named *iComment* [112], which combined Natural Language Processing (NLP), machine learning, and program analysis techniques. In the follow-up works, they investigated the detection of code-comment inconsistencies concerning the specific concepts of lock mechanism [113], function calls [113], and concurrency-related interrupts [114]. Stulova et al. [110] designed a technique and a tool, *upDoc*, which built a map between the code and its comment, ensuring that changes in the code match the changes in the respective comment. Liu et al. [71] proposed a machine learning-based method to check whether the comments should be changed with the code evolution. In addition, Cimasa et al. [18] adopted word embedding techniques to detect the incoherence between source code and their associated comment. Their extensive experiments demonstrate their method is more

efficient in terms of execution time while maintaining performance very close to the baseline. Moreover, Corazza et al. [19] created a publicly available Java dataset consisting of 3,636 methods in three open-source software applications to investigate coherence between the source code and associated comments. Besides, their adoption of Support Vector Machine (SVM) with tf-idf has also been proven to be effective in code-comment incoherence detection. The above-mentioned studies focus on code-comment inconsistency detection, while code-comment synchronization aims to automatically synchronize comments when the associated code changes, thereby avoiding the introduction of inconsistent comments.

Chapter 3

Smart Contracts Vulnerability Auditing With Multi-Semantics

Smart contracts vulnerability auditing is vitally critical to ensure transaction execution in normal on the blockchain. Current data-driven approaches normally tokenize smart contracts into a series of sequences according to only one tokenization standard for vulnerability detection purposes, resulting in some of the semantic contexts cannot be reflected within restricted sequence length.

To address this limitation, we propose three heterogenous tokenization methods for multi-semantic code representation and utilize the n-gram language model to learn their semantic contexts, respectively. Afterward, we exploit our effective combination strategy of *Intersection* or *Union* to integrate the audited results from multiple semantic contexts. In order to evaluate the proposed approach, we applied it to over 7,200 Ethereum smart contract samples. Experimental results show our proposed method is capable of detecting vulnerabilities and competitive with the baseline in test sets, with improved precision of over 44% when *Intersection* is applied in their results, as well as improved Recall by over

300% and F-measure by 220% at most when *Union* is applied. Our proposed method for smart contract vulnerability detection, an important tool for developing quality decentralized software applications, is able to analyze multiple semantic contexts and successfully detects more true vulnerabilities with high precision, outperforming that of the baseline approaches. This work was published at the 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC) [135].

The rest of this chapter is organized as follows. Section 3.1 presents the background and theoretical foundations. Section 3.2 proposes MSgram and introduces its methodology. Section 3.3 discusses the experimental design and results, respectively. In Section 3.4, we elaborate on the threats to the validity of this work. The summary of this chapter is provided in Section 3.5.

3.1 Background

3.1.1 Ethereum Smart Contracts and Vulnerabilities

Ethereum Smart contracts consist of a series of state variables and functions that can be invoked by Ethereum's externally own accounts for transaction operations. Most of those state variables store digital assets or currency possessed by users and those functions manage the trading processes or regulations predefined by the contract deployer. Due to the unfamiliarity with smart contract development and its peculiar operation mechanism, a lot of security risks in capital management and trading continuously come out [48]. The reentrancy risk leading to the DAO attack in 2016 is a typical example. It is caused by malicious attackers trying in a recursive manner to invoke a profitable function of the targeting contract repeatedly via his attack contract. Other instances like timestamp dependency risk, i.e., some of the critical operations depending on timestamp, could have

the risk of getting attacked by malicious attackers once they change the block's timestamp illegally to make profits. Vulnerabilities in smart contracts, as mentioned above, won't crash the code execution but can destroy the normal transaction logics predefined by contract deployers, thereby causing huge losses [7, 117, 48].

3.1.2 N-gram Language Model

In this chapter, we adopt the N-gram Language Model (NLM) to analyze those token sequences generated from smart contracts. The core methodology of the NLM is derived from the Markov chain [87], which is essentially a Bayesian conditional probability model. It uses the continuous former n words to estimate the probability of the current word t_i, i.e. $P(t_i|t_{i-(n-1)},...,t_{i-1})$ as shown in Equation 3.1. Thus, given a sentence, e.g. $S = t_1t_2t_3t_4,...,t_K$, its probability is shown in Equation 3.2.

$$P(t_i|t_{i-(n-1)},...,t_{i-1}) = \frac{count(t_{i-(n-1)},...,t_{i-1},t_i)}{count(t_{i-(n-1)},...,t_{i-1})} \tag{3.1}$$

$$P(S) = \prod_{i=1}^{K} P(t_i|t_{i-(n-1)},...,t_{i-1}) \tag{3.2}$$

After we train the model (i.e. we calculate each word's probability of n-gram in the whole corpus), similarly with S-gram, we also adopt the log-transformed perplexity metric to measure whether a token sequence is a potential vulnerability or not, because the form of addition can speed up the calculation while avoiding the floating-point number overflows downward caused by the too small probability after product. In the field of natural language process, a sentence with a higher perplexity score means its word sequence is scarcer. In other words, it tends to be an abnormal or incorrect sentence. Since vulnerable sequences occupy only a small part of the whole dataset, if a token sequence is scarce, it

has a great chance to be a vulnerable sequence [126]. Therefore, for those sequences with high perplexity scores, we put them into the candidate vulnerability list. The perplexity log-transformed score for a whole sentence S with the length of K is represented by $H(S)$ below:

$$H(S) = -\frac{1}{K}\sum_{i=1}^{K} P(t_i|t_{i-(n-1)},...,t_{i-1}) \tag{3.3}$$

3.2 Methodology

3.2.1 Model Framework Overview

The framework of the proposed MSgram includes two phases, i.e. the model construction phase and the security auditing phase, respectively. Figure 3.1 shows the overview of MSgram. In the model construction phase, **the first step is Tokenization**. We propose three tokenization standards and adopt our tokenizer to extract each smart contract source code in the training set into three sequence collections of *Combined Sequence*, *Text Sequence*, and *Structure Sequence*, respectively. **The next step is model training and optimization.** We feed each collection into an NLM tool powered by KenLM [39] to train their models and optimize them by different parameters and thresholds. Then we select those optimized NLMs based on the precision of their separate audited result, we dub the optimized NLM as *Detector*. **The final step is designing and assessing combination strategies.** We provide two kinds of combination strategies, i.e. *Intersection* and *Union*, to improve the performance of those single *Detectors* and thereby generating final audited results. The verified best parameter and threshold combination for each single *Detector*, along with the effective combination strategy, will be applied in the security auditing phase.

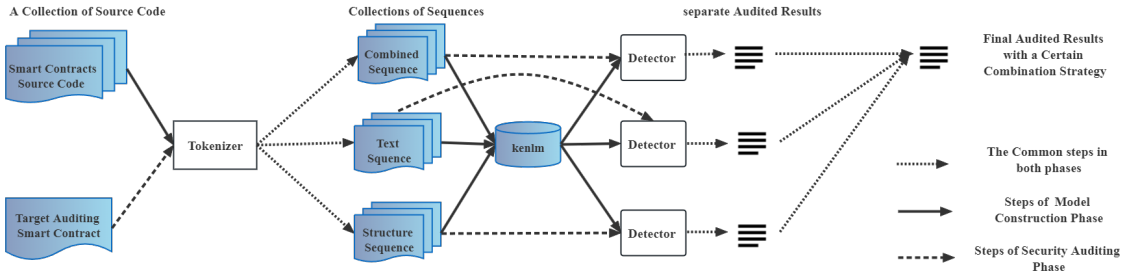


Figure 3.1: The framework overview of MSgram

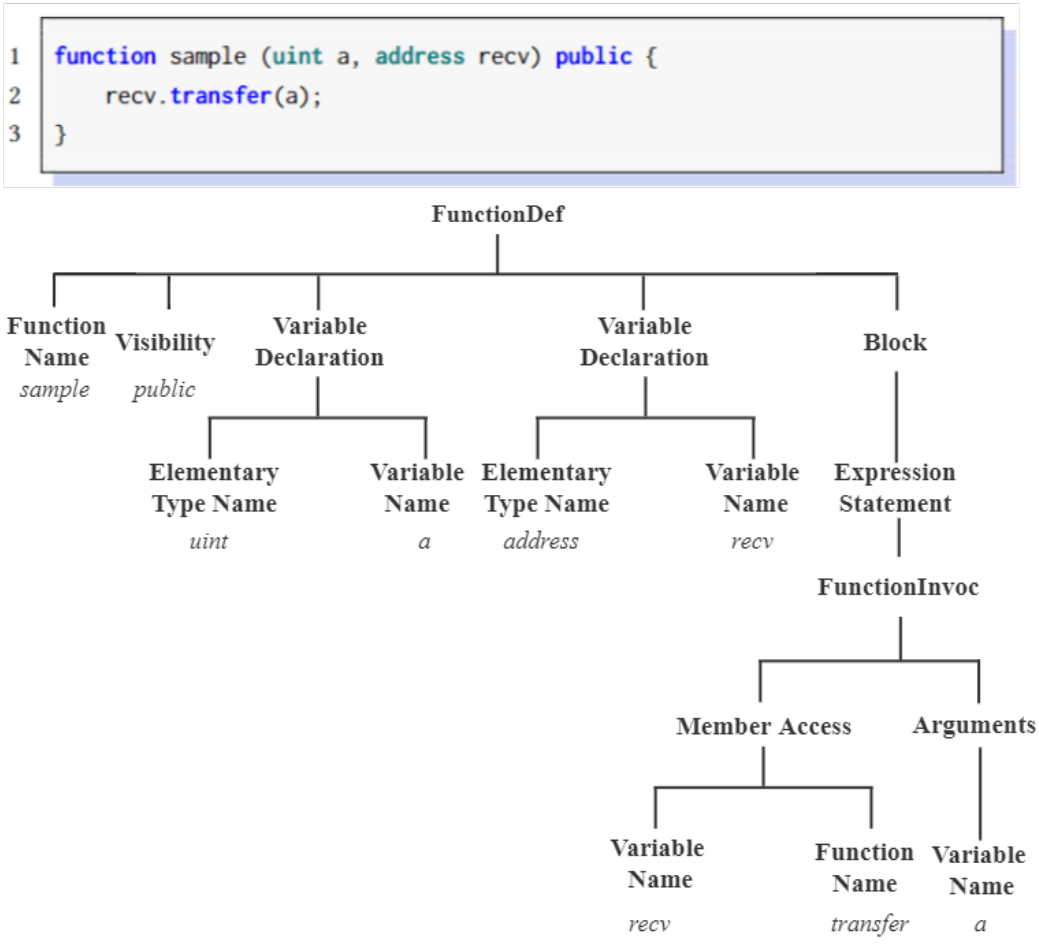


Figure 3.2: The function “sample” and its AST

3.2.2 Tokenization

In this chapter, we modify SBT Traversal to generate three kinds of sequences, which is the first step of MSgram. SBT Traversal essentially transforms each node of AST into tokens for sequence construction and uses a series of brackets to reserve structural information of code in sequences, making them recoverable to AST. Figure 3.2 shows the AST of a contract function "sample" that is used for transferring ethers to the account address "recv". More specifically, leaf nodes in ASTs are composed of **type** and *value*, while non-leaf nodes are composed of **type** only, as we specified in bold and italic in Figure 3.2, respectively. The original SBT Traversal uses four tokens: "(", "**type**(**type**-*value*)", ")" and "**type**(**type**-*value*)" to represent a non-leaf(leaf) node in ASTs. Moreover, it arranges tokens in a nested manner to represent code structure [43]. Nonetheless, in NLMs, we cannot distinguish different brackets of different nodes in this way for every bracket with the same probability in NLM, causing us to lose the structure range information of different nodes. On the other hand, too many insignificant tokens increase the perplexity of normal sequences with different degrees, which increases the model's false positive rate. Besides, former literature has studied adequately and testified that researchers should tokenize the source code at a relatively high level when detecting vulnerabilities in semantics. Otherwise, the NLM could only detect syntactic errors, which is the work of compilers [126]. Therefore, we modify the SBT Traversal basically in the following way: 1) For non-leaf nodes denoting structures such as "FunctionDefinition" and "ExpressionStatement", we represent them by two tokens: "(**type**" and "**type**)" so that the language model can differentiate structures' range from themselves with their corresponding probabilities concisely. 2) For those leaf nodes without range information, we represent them in the format of "**type**-*value*", such as **ElementaryTypeName**-*uint* and **VariableName**-*a*, thereby dropping insignificant tokens

but guarantees their integrity of necessary information. 3) We also add characteristics of the tokenization standard S-gram used, such as adding a "call_min_gas" token when using the transfer function, to make our tokenizer more specific to Ethereum smart contracts. After the above operation, the code snippet in Figure 3.2 can be extracted into a *Combined sequence* like below Figure 3.3, where we adopt different indentations to show its structure intuitively. From the above sequence, we can recover it back to the AST. Because the structural information has been reserved, and each leaf node holding **type** and *value* can reserve its text information. For the *Structure sequence*, we delete all the leaf nodes, which means we extract it at a much higher semantic level and only keep the code structural information. For the *Text sequence*, we keep those leaf nodes as well as specifier tokens to denote a certain intention's start, such as **FunctionSpecifier**-*function*, which means a start of a function. Both examples extracted from the above code snippet are listed in Figure 3.3. Based on the basic rules above, we create a tokenizer by a solidity parser [1] to generate the three kinds of sequences for the whole data set.

3.2.3 Model Parameters and Thresholds

This section introduces the parameters (i.e. gram number, sequence length, and minimum token occurrence) and thresholds (i.e. minimum perplexity and top flagged vulnerability amount) used in model training and optimization. Different from other literature adopting NLM [70, 126, 100, 36], we add another new threshold which is the minimum perplexity score to label the potential vulnerabilities together with the top flagged vulnerability amount, thereby making the candidate vulnerability list in audited result dynamic with different NLMs.

- Gram Number: An n-gram language model with different *n* will assign different

[1] https://github.com/federicobond/solidity-parser-antlr

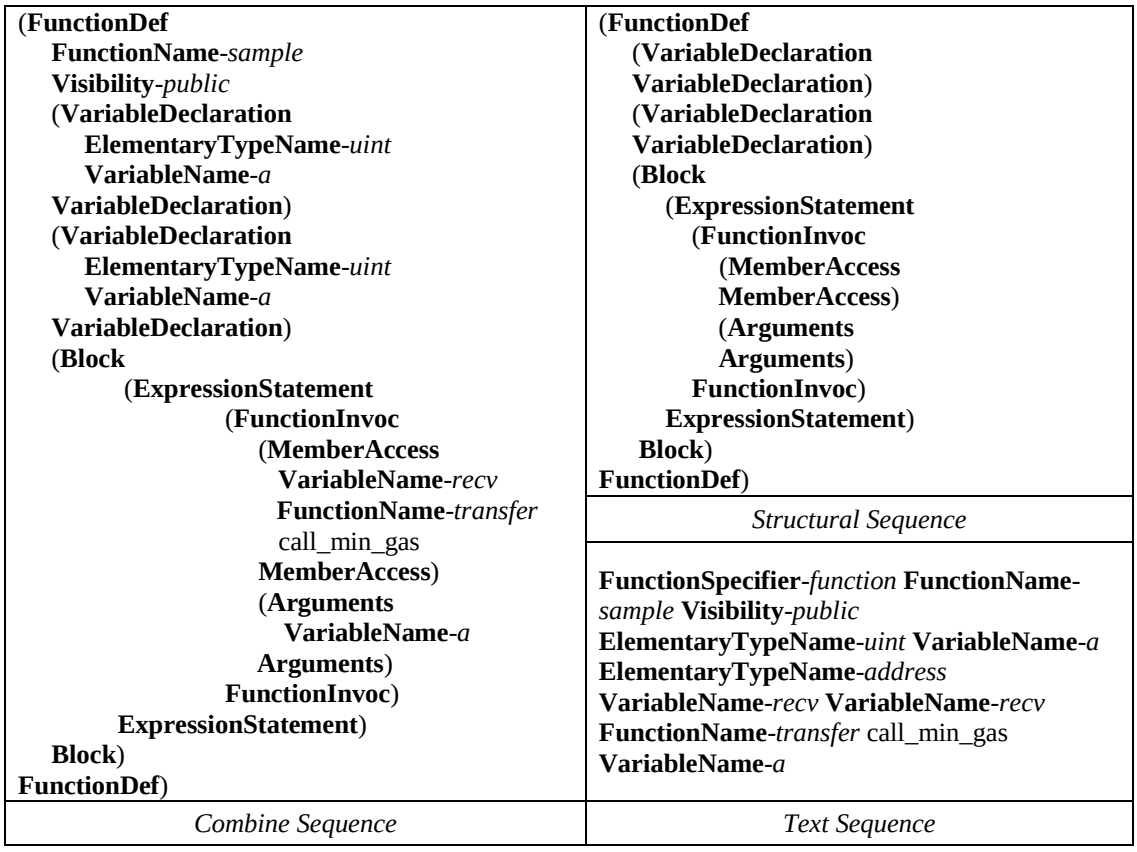


Figure 3.3: Sequence representations of function “sample”

perplexity scores for the same sequence. Due to the former study [70] has verified that when the n is larger than 5, the performance of NLM decreased in smart contracts vulnerability auditing. Thus we evaluate n with 2-6 to find the most appropriate one in our experiment.

- Sequence Length: Sequence length represents the number of tokens in a sequence. If we only detect the whole sequence of a contract, we cannot accurately localize the vulnerable row or position. Thus researchers who study vulnerability auditing with NLM normally break the whole sequence into snippets. Nevertheless, different snippets may reflect different scenarios, either vulnerable or not. Moreover, different lengths of sequences also affect the performance in vulnerability auditing. Consequently, in our experiment, we set the sequence length as $\{10, 20, 30, 40, 50\}$ respectively, to observe their impacts. Since the experiment empirically testifies that the precision of the separate audited result increases steadily with the increment of sequence length, thus we cannot optimize the NLM by this parameter. Instead, we evaluate the MSgram under each fixed sequence length.

- Minimum Perplexity: The minimum perplexity is one of the thresholds used in constructing candidate vulnerability lists and those sequences with higher perplexity scores than the predefined minimum perplexity score would be labeled as potentially vulnerable sequences. Since the same contract with different NLMs has different perplexity ranges and distributions for their different sequences snippets (e.g. some NLMs may mark the perplexity score of a collection of sequences in a range of 1-20, while others may mark them in a range of 100-10000), setting a same minimum perplexity for them when implementing the vulnerability auditing would lead to an unfair comparison among NLMs. Moreover, even if we normalize them into a fixed range such as 0-1, their distribution is still different and without any

change. In this case, after each collection of sequences is fed into their corresponding trained NLMs and obtain their collections of perplexity scores, we all evenly set three minimum perplexities within the range of the minimum and the median of their entire perplexity scores respectively, to ensure we can capture more than half of those true vulnerabilities every time under this threshold and try to keep fair comparison in different NLMs.

- Top Flagged Vulnerability Amount: The top flagged vulnerability amount is another threshold used in constructing candidate vulnerability lists. After we select a series of sequences with higher perplexity than the predefined minimum perplexity score, we rank them in descending order and only capture those sequences within the range of the top flagged vulnerability amount to further filter false positives in the candidate vulnerability list. For each collection of perplexity scores, we select six numeric values below or around the mean of perplexity scores to ensure we have the chance of capturing all vulnerable sequences of each smart contract under this threshold and try to keep a fair comparison as well.

- Minimum Token Occurrence: Setting the minimum token occurrence when building NLMs can generalize the model to avoid incorrectly auditing for some non-vulnerable sequences. In our experiment, we convert all the tokens that only appear three times to "<UNK>" as another parameter of NLM, which is a testified practice in NLP study [79].

3.2.4 Combination Strategies

After recording the parameter and threshold combinations of optimized NLMs for each sequence type under each sequence length, we start to combine their separate audited

results, which is the last step of MSgram. In this section, we try two general kinds of strategies for combination. Each strategy includes two steps for exploration. The 1^{st} step is combining by sequence type, i.e. integrating audited results based on different semantics. The 2^{nd} step is further pruning by sequence length to try to reduce false positives. In the following part, we first introduce step 1 of each strategy respectively, then introduce step 2 of them in the last paragraph of this section.

The first strategy is *Intersection*. An obvious principle is if a sequence or its sub-sequence can be detected as vulnerable by more than one *Detector* of different sequence types, it has a much more opportunity to be a true vulnerability. Since the sequence length represents the number of tokens in a sequence, thus even with the same sequence length, the range of sequences in a contract of different sequence types are various for different semantic levels. In this case, we cannot simply combine the separate audited results. Instead, we adopt those sub-sequences (i.e. sequences within the range of its long sequences in smart contracts) to prune their long sequences in corresponding audited results. Furthermore, the range of *Structure sequence*s is larger than that of *Text sequence*s and *Combined sequence*s under the same sequence length in the same contract, which means *Text sequence*s and *Combined sequence*s are always sub-sequences of *Structure sequence*s. Thus, as the 1^{st} step of *Intersection*, we consider that if a sequence is confirmed as vulnerable in the Structure Sequence [2] and its sub-sequences in Combined Sequence or Text Sequence are also detected as vulnerable, then this sequence is labeled as vulnerable. More formally, we get a new candidate vulnerable set of audited results for

[2]We adopt the "Combined Sequence" with initial capital letters to represent the collection of combined sequences, which is also applicable to "Text Sequence" and "Structure Sequence".

the smart contract i by the following equation $S_z(i)$:

$$\begin{aligned} S_z(i) = \bigcup_{\forall z,j,k \in i} AR_{Struct}(z, \theta_{Struct}) \wedge \\ ((AR_{Text}(j, \theta_{Text}) = TRUE) \wedge Range(j) \subseteq Range(z)) \\ \vee (AR_{Comb}(k, \theta_{Comb}) = TRUE \wedge Range(k) \subseteq Range(z)) \end{aligned} \tag{3.4}$$

where $AR_{Struct}(z, \theta_{Struct})$, $AR_{Text}(j, \theta_{Text})$, and $AR_{Comb}(k, \theta_{Comb})$ represents the audited result for *Structure sequence* z, *Text sequence* j, and *Combined sequence* k in ith contract by the optimized NLM with best parameter and threshold combination θ_{Struct}, θ_{Text}, and θ_{Comb}, respectively. $Range(z)$, $Range(j)$, and $Range(k)$ represent the ranges of sequences z, j, k, respectively. $\bigcup$ represents the union of each sequence that satisfies the conditions we defined.

The second strategy is *Union*. Since those sequences are generated from different tokenization standards, leading them to capture different semantics by different token arrangements and semantic levels. Therefore, if we take a union way to combine their audited results, it should capture more true vulnerabilities than models that only apply one tokenization. In this case, we use the audited results of the other two kinds of sequences to trim the false negatives for the audited results of the *Structure sequence* in a union way, which is the 1^{st} step of the *Union* strategy. Specifically, in addition to keeping sequences labeled as vulnerable in audited results for Structure Sequence, once a sub-sequence of a *Structure sequence* is identified as vulnerable in Text Sequence or Combined Sequence, we label this *Structure sequence* as vulnerable. More formally, we obtain another new candidate vulnerable set of audited results for the ith smart contract by the following

equation $S_z(i)$:

$$\begin{aligned} S_z(i) = \bigcup_{\forall z,j,k \in i} AR_{Struct}(z, \theta_{Struct}) \vee \\ ((AR_{Text}(j, \theta_{Text}) = TRUE) \wedge Range(j) \subseteq Range(z)) \\ \vee (AR_{Comb}(k, \theta_{Comb}) = TRUE \wedge Range(k) \subseteq Range(z)) \end{aligned} \tag{3.5}$$

Here we introduce the 2^{nd} step of each strategy. When comes to this step, all audited results have been combined based on the structure type. Thus, in this step, we further prune false positives by different sequence lengths, which means the 2^{nd} step is the same in both strategies. Basically, we use the audited result of the previous smaller sequence length to prune the false positives in that of the current sequence length, e.g., using the audited result of sequence length 10 to prune the false positives in that of sequence length 20. The rationale of the 2^{nd} step is if a vulnerability is confirmed in multiple sequences of different lengths, it has a great chance to be a true vulnerability. For instance, in audited results of the same sequence type but different sequence lengths, if a sequence *BC* is labeled with vulnerable in an audited result of sequence length 2 and in another audited result of sequence length 3, a sequence *ABC* is also labeled with vulnerable. Since *BC* is a sub-sequence of *ABC*, there is a great chance that a vulnerability exists in ABC. Thus we labeled *ABC* with vulnerable. Otherwise, it should be labeled as non-vulnerable to reduce false positives. More formally, we obtain the final candidate vulnerable set of the audited result for the *i*th smart contract by the equation below.

$$S_z(i) = \bigcup_{\forall z,j,k \in i} AR_{SL+10}(z) = TRUE \wedge (AR_{SL}(j) = TRUE \wedge Range(j) \subseteq Range(z)) \tag{3.6}$$

where $AR_{SL+10}(z)$ represents the audited result of *Structure sequence z* in *i*th contract under

sequence length $sl + 10$ (as our interval is 10 among each sequence length) processed by the 2^{nd} step. But the performance of the above strategies' application and combination need further empirical verification, which will be studied in Section 3.3.

3.3 Experiments

3.3.1 Experimental Design and Evaluation Metrics

The whole experiments mainly follow the steps discussed in Section 3.2, including the data collection, NLM optimization, combination strategy exploration, and the comparison with baseline. The baseline method, i.e., S-gram, utilizes accuracy as the evaluation metric, which is actually unfair and meaningless. Because vulnerable sequences only occupy an extremely small part of the whole sequence collection, which means even a vulnerability detector can only detect very few true vulnerabilities, its performance in accuracy still can be very high. Therefore, the accuracy metric cannot reliably reflect the true performance of the model. As such, in our experiment, we adopt precision, recall, and F-measure to assess the model's performance on the whole testing set. Precision measures a model's performance of detecting true vulnerabilities among labeled vulnerabilities, while recall measures a model's performance of detecting true vulnerabilities among the whole true vulnerabilities. And F-measure is the harmonic mean of the above metrics. Only when the model's performance in precision and recall are both high, its F-measure will be high.

3.3.2 Dataset Collection and Construction

To make a fair comparison with the S-gram, we curate all smart contracts used in its original paper, which are extracted from Etherscan.io [3], and adopt the Oyente [76] to

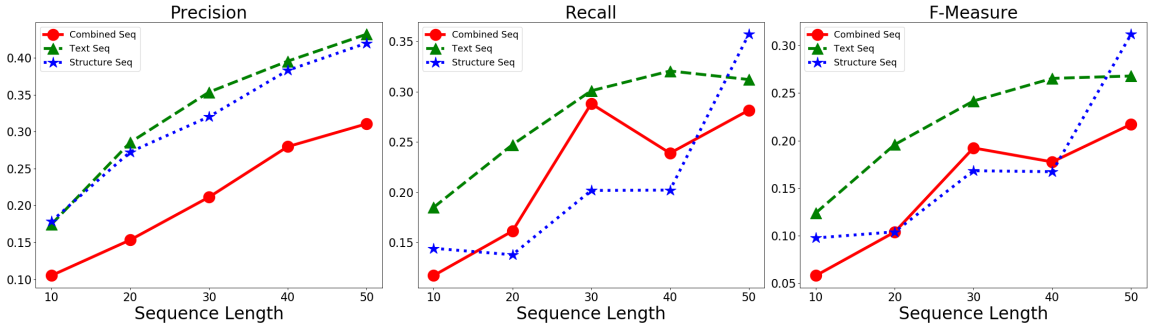
[3] https://etherscan.io/

confirm vulnerabilities for those contracts and generate test reports, which follows the same way applied in the literature of S-gram. However, we find there are a lot of identical smart contracts but with different contract addresses, owing to deploying the identical contracts more than once on Ethereum. This circumstance might decrease the perplexity of both normal and defective sequences of repeatedly deployed contracts, causing the experimental results to be imprecise. Therefore, we delete all the duplicated contracts. Finally, we obtain a total of over 7,200 smart contracts as our dataset. We divide the whole dataset into two parts of the training set (90%, around 6,500 smart contracts) and the testing set (10%, around 700 smart contracts). The whole dataset has been published [4], including source code and test reports that record the location and type of vulnerabilities.

3.3.3 Experimental Devices

Our experiments are conducted on a Windows 10 laptop with 6 cores and 12 threads of Intel Core 2.20GHz i7-8750H CPU, 16GB RAM, and 512GB SSD.

Figure 3.4: The performance of each optimized NLM for three kinds of sequence types under different sequence lengths



[4] https://github.com/yz1019117968/MSgramDataset

Table 3.1: The parameter and threshold combination of each optimized NLM for three kinds of sequence types under different sequence lengths

	Optimized NLM For Combined Sequence			**Optimized NLM For Text Sequence**			**Optimized NLM For Structure Sequence**		
Sequence Length	***Gram Number***	***Minimum Perplexity***	***Top Flagged Vulnerability Amount***	***Gram Number***	***Minimum Perplexity***	***Top Flagged Vulnerability Amount***	***Gram Number***	***Minimum Perplexity***	***Top Flagged Vulnerability Amount***
10	3	100	30	2	70	40	6	30	20
20	2	20	60	2	30	60	3	5	20
30	2	7	130	3	7	80	3	3	40
40	5	7	180	2	30	130	6	4	50
50	5	5	200	3	7	150	4	4	150

Table 3.2: The performance of each optimized NLM for three kinds of sequence types under different sequence lengths

	Optimized NLM For Combined Sequence			**Optimized NLM For Text Sequence**			**Optimized NLM For Structure Sequence**		
Sequence Length	***Precision***	***Recall***	***F-Measure***	***Precision***	***Recall***	***F-Measure***	***Precision***	***Recall***	***F-Measure***
10	0.1050	0.1172	0.0580	0.1743	0.1850	0.1240	0.1784	0.1443	0.0976
20	0.1532	0.1614	0.1038	0.2853	0.2473	0.1958	0.2724	0.1379	0.1040
30	0.2112	0.2881	0.1923	0.3534	0.3009	0.2417	0.3201	0.2018	0.1683
40	0.2796	0.2389	0.1776	0.3954	0.3206	0.2655	0.3829	0.2022	0.1673
50	0.3105	0.2816	0.2172	0.4318	0.3121	0.2680	0.4198	0.3571	0.3119

3.3.4 Optimization for the NLM

In this part, we adjust those parameter and threshold combinations discussed in section 3.2 and investigate their influences on each single NLM. In previous studies [7], researchers tend to fix the sequence length and top flagged vulnerability firstly to find a gram number with the highest precision or maximum average true vulnerabilities. In the way of adjusting one parameter while fixing others, all the parameters can be explored. However, this kind of greedy optimization method cannot find the best parameter and threshold combination, since it has no chance of testifying every combination of those parameters and thresholds. Thus, in our experiments, we try to exploit all the parameters and thresholds together to explore different combinations as many as we can for each single NLM's construction, thereby determining the best combination.

During the whole experiments, we try different combinations of those parameters and thresholds and constructed 5*5*3*6*3=1350 models in total for comparison on the precision of the whole training set. Then the model with the highest precision under each sequence length in each sequence type is selected as the optimized NLM. Table 3.1 presents the parameter and threshold combination of each optimized NLM for three kinds of sequence types under different sequence lengths. Table 3.2 presents their performances in precision, recall, and F-measure. Obviously, regardless of sequence types, the highest value of each metric almost appears when auditing the longest sequences. On the other hand, by horizontal comparison, the performance of NLMs for Text Sequence is always the best under each sequence length, which means this kind of tokenization standard captures more useful information in vulnerability detection.

We also present Figure 3.4 to show their performance variations among five sequence lengths clearly. According to Figure 3.4, we find that, for each sequence type, with the increment of sequence length, each of the evaluation metrics is generally in ascending

order, especially the precision, which is also demonstrated in the literature of S-gram [70]. It is because the increment of sequence length leads to the overlap of vulnerable snippets, which means a sequence may consist of more vulnerabilities, causing those vulnerabilities easier to be detected to some extent. Nevertheless, a very long sequence is useless in localizing vulnerability in a small range, thus developers should refer to the actual situation to select a proper sequence length and we cannot optimize NLM on it. As such, in later comparison with the baseline, we compare the performance under each sequence length, which follows the same way used in the S-gram paper.

Table 3.3: The Precision before and after the *Intersection* strategy under each sequence length

Sequence Length	**Original**	**After the Intersection Strategy**	
		1st Step: Combined by Sequence Type	**2nd Step: Pruning by Sequence Length**
10	0.1784	0.2061	0.2061
20	0.2724	0.3038	0.4606
30	0.3201	0.3416	0.3731
40	0.3829	0.3948	0.4638
50	0.4198	0.4376	0.4538
Average	0.3147	0.3368	0.3915

3.3.5 Experiments on Combination Strategies

In this section, we conduct a series of experiments on the two combination strategies proposed in section 3.2 to explore the way of exploiting the audited results from multi-semantics.

For the *Intersection* strategy, we focus on its improvement of precision. After implementing the combination by sequence type and pruning by sequence length, we

Figure 3.5: The improvements on Precision of Step 1, Step 2 under *Intersection* strategy and this strategy's final improvement

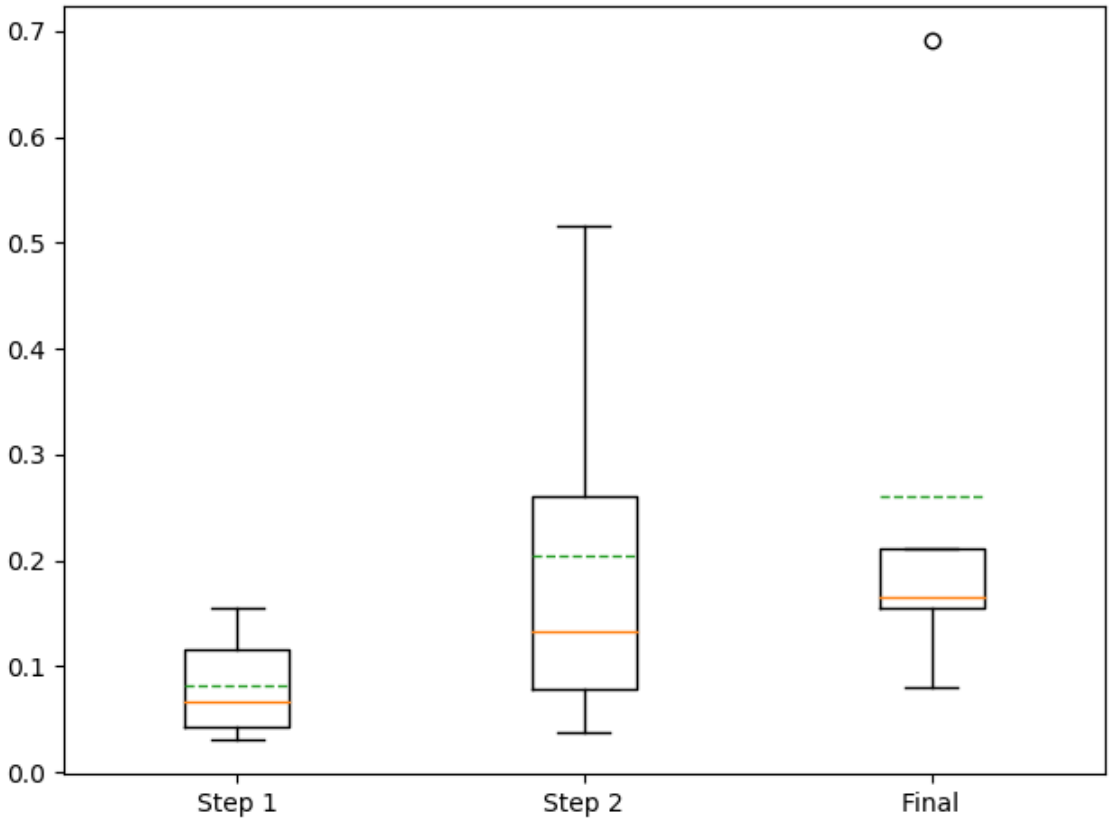


Table 3.4: The improvement on Precision after the *Intersection* strategy under each sequence length

	Improvement of Intersection Strategy		
Sequence Length	**1st Step: Combined by Sequence Type**	**2nd Step: Pruning by Sequence Length**	**Final Improvement**
10	15.53%	/	15.53%
20	11.53%	**51.60%**	**69.08%**
30	6.74%	9.20%	16.56%
40	3.11%	17.48%	21.14%
50	4.23%	3.71%	8.10%
Average	8.23%	20.50%	26.08%

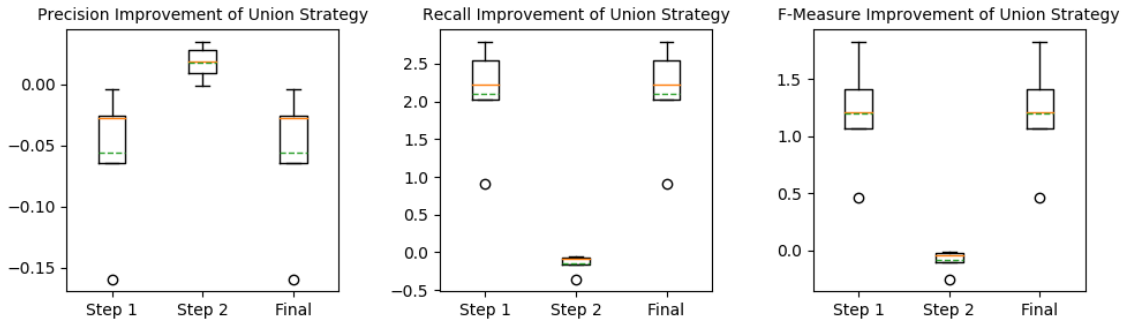
Table 3.5: The Precision, Recall, and F-Measure before and after the *Union* strategy under different sequence length

Sequence Length	Original			Performance After the Union Strategy					
				1st Step: Combined by Sequence Type			*2nd Step: Pruning by Sequence Length*		
	Precision	***Recall***	***F-Measure***	***Precision***	***Recall***	***F-Measure***	***Precision***	***Recall***	***F-Measure***
10	0.1784	0.1443	0.0976	0.1499	0.5124	0.2019	0.1499	0.512	0.2019
20	0.2724	0.1379	0.1040	0.2550	0.5232	0.2943	0.2547	0.496	0.2862
30	0.3201	0.2018	0.1683	0.3186	0.6498	0.3719	0.3297	0.596	0.3679
40	0.3829	0.2022	0.1673	0.3730	0.6120	0.4030	0.3775	0.554	0.3824
50	0.4198	0.3571	0.3119	0.4082	0.6838	0.4548	0.4187	0.438	0.3384
Average	0.3147	0.2087	0.1698	0.3009	0.5962	0.3452	0.3061	0.519	0.3154

Table 3.6: Improvement in Precision, Recall, and F-Measure after the *Union* strategy under different sequence length

Sequence Length	Improvement in Precision, Recall, and F-Measure After the Union Strategy						Final Improvement		
	1st Step: Combined by Sequence Type			*2nd Step: Pruning by Sequence Length*					
	Precision	***Recall***	***F-Measure***	***Precision***	***Recall***	***F-Measure***	***Precision***	***Recall***	***F-Measure***
10	-16.00%	254.97%	106.78%	/	/	/	-16.00%	254.97%	106.78%
20	-6.40%	**279.38%**	**182.97%**	-0.10%	-5.10%	-2.80%	-6.40%	**279.38%**	**182.97%**
30	-0.50%	221.94%	120.92%	3.49%	-8.30%	-1.10%	-0.50%	221.94%	120.92%
40	-2.60%	202.64%	140.93%	1.20%	-9.60%	-5.10%	-2.60%	202.64%	140.93%
50	-2.80%	91.46%	45.81%	2.56%	-36.00%	-25.60%	-2.80%	91.46%	45.81%
Average	-5.60%	210.08%	119.48%	1.79%	-14.70%	-8.60%	-5.60%	210.08%	119.48%

Figure 3.6: The improvement on Precision, Recall, and F-Measure of Step 1 and Step 2 under *Union* strategy and this strategy's final improvement



present Table 3.3 to demonstrate the precision of audited results under each sequence length before and after applying the *Intersection* strategy. Since the strategy is applied based on the audited result of Structure Sequence, we set its precision as an original reference. Figure 3.5 shows the improvements of two steps under the *Intersection* strategy towards the original precision and the precision after 1*st* step respectively, as well as this strategy's final improvement towards the original precision. Each box represents the improvements in precision under five sequence lengths respectively, the green dotted line represents the mean improvement among five sequence lengths. It is obvious that the improvement by the 1^{st} step is small compared with that of the 2^{nd} step. Table 3.4 showcases Figure 4's information by accurate numbers. we find that the improvement by the 1^{st} step up to 15.53%, on average 8.23% but the improvement by the 2^{nd} step can up to 51.60%, on average 20.50%. The final precision of audited results under each sequence length achieves improvement by up to 69.08%, on average 26.08% with respect to the original precision. Nevertheless, their recall and F-measure decline, which is caused by a small part of the incorrect removal of true positives during pruning. Due to the setting that this strategy is used for boosting the detecting precision, the little decline in recall and

F-measure can be tolerated. Therefore, we keep this strategy and adopt it in the security auditing stage.

For the *Union* strategy, we mainly focus on the final result of whether it can find more vulnerabilities. Similarly, after implementing the combination by sequence type and pruning by sequence length, we present Table 3.5 to demonstrate the performance on precision, recall, and F-measure before and after applying the *Union* strategy for audited results under each sequence length. Similarly, we adopt the performance of audited results for Structure Sequence as the original reference to show the improvements. Figure 3.6 adopts three sub-figures to demonstrate the improvements in step 1 and step 2 under the *Union* strategy, as well as this strategy's final improvement toward each metric. For every sub-figure, each box represents the five improvements on its metric under five sequence lengths respectively, and the green dotted line represents the mean improvement on this metric among five sequence lengths. It's obvious that the 1^{st} step's improvements on recall and F-measure are substantial but those of 2^{nd} step are weak and even negative. On the other hand, their improvements in precision are both weak. More specifically, Table 3.6 shows Figure 3.6 information by accurate numbers, from which we find that, after 1^{st} step, the recall achieves improvements by up to 279.38%, on average 210.08% and the F-measure by up to 182.97%, on average 119.48% after the combination by sequence type with comparison to the original performance, although the precision declines a little by 5.63% on average. Nevertheless, after the 2^{nd} step, the precision on average improves only by 1.79% while the recall and F-measure both decline on average by 14.73% and 8.63%, which is not worth for pruning. As such, in the later security auditing stage, for the strategy of *Union*, we only adopt the 1^{st} step of combination by sequence type and drop the 2^{nd} step.

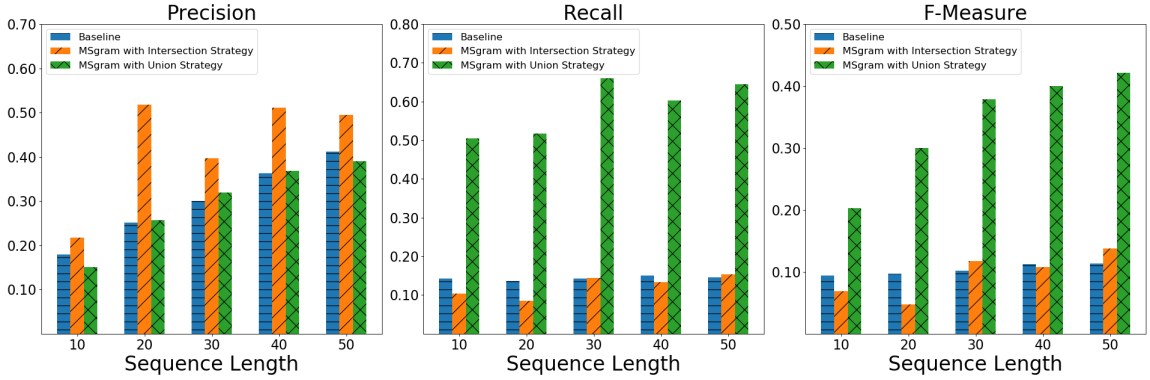
Table 3.7: The performance of baseline and MSgram under different sequence length

	Baseline					**MSgram in Intersection Strategy**					**MSgram in Union Strategy**				
Sequence Length	***10***	***20***	***30***	***40***	***50***	***10***	***20***	***30***	***40***	***50***	***10***	***20***	***30***	***40***	***50***
Precision	0.1791	0.2507	0.3006	0.3633	0.4112	0.2176	**0.5179**	0.3972	0.5116	0.4942	0.1501	0.2571	0.3196	0.3679	**0.3897**
Recall	0.1423	0.1366	0.1433	0.1510	0.1465	0.1036	0.0844	0.1441	0.1336	**0.1540**	0.5051	0.5175	**0.6602**	0.6020	0.6439
F-Measure	0.0937	0.0972	0.1020	0.1129	0.1138	0.0691	0.0469	0.1173	0.1075	**0.1374**	0.2030	0.3001	0.3785	0.4004	**0.4211**

Table 3.8: The improvement of the MSgram under different sequence length

	Improved by MSgram in Intersection Strategy						**Improved by MSgram in Union Strategy**					
Sequence Length	***10***	***20***	***30***	***40***	***50***	***Average***	***10***	***20***	***30***	***40***	***50***	***Average***
Precision	21.5%	106.59%	32.11%	40.81%	20.19%	44.24%	-16.20%	2.56%	6.31%	1.27%	-5.23%	-2.26%
Recall	-27.2%	-38.21%	0.53%	-11.52%	5.09%	-14.26%	255.05%	278.75%	360.55%	298.68%	339.48%	306.50%
F-Measure	-26.3%	-51.78%	14.94%	-4.80%	20.74%	-9.43%	116.54%	208.71%	271.02%	254.71%	270.14%	224.22%

Figure 3.7: The performance of baseline and MSgram



3.3.6 Comparison with the Baseline

After the models are trained and the combination strategies are configured, we start to adopt them on the testing set to obtain an unbiased estimation for the performance of the MSgram and make a comparison with the baseline. Since the paper of S-gram did not publish its source code, we try our best to re-implement one for the comparison. Table 3.7 shows the performance of MSgram with both combination strategies and that of the baseline we re-implemented. Table 3.8 presents the improvement of MSgram with respect to the baseline. Figure 3.7 reflects the content of Table 3.7 and Table 3.8 intuitively. It is obvious that, in the testing set, the MSgram's precision in vulnerability detecting can be up to 51.16% in sequence length 40 under the *Intersection* strategy while, under the *Union* strategy, the precision, recall, and F-measure can be up to 38.97% in sequence length 50, 66.02% in sequence length 30 and 42.11% in sequence length 50 respectively. Then compared with S-gram, the MSgram's audited result of *Intersection* strategy can improve the precision by up to 106.59%, on average 44.24%. Under sequence lengths 30 and 50, the improvement appears in all three metrics. While the MSgram's audited result of *Union* strategy, although on average declines 2.26% in precision, can improve the recall by 360.55% at most, on average by 306.50% and the F-measure by 271.02% at most, on average by 224.22%. More

specifically, under sequence lengths 20, 30, and 40, MSgram outperforms the baseline in all three metrics. Thus, both the *Intersection* and the *Union* strategy can obtain great improvements in their focusing metrics, and keep stability or even excel on other metrics. As such, MSgram outperforms the baseline to a significant extent.

3.4 Threats to Validity

This chapter includes the following threats to validity:

The Re-implementation for the baseline. Since the source code of S-gram was not published, we tried our best to follow every experimental detail and re-implemented one for comparison with our improved MSgram model. The re-implemented S-gram model presents a higher precision than the one in its paper but with lower accuracy. One possible reason is the smart contracts that the S-gram paper used include a lot of duplicates (we pointed out in Section refDataset Collection and Construction), where invulnerable sequences among those occupy a larger part, causing the relatively high accuracy of the S-gram, even if its precision is low.

Vulnerabilities labeled by a symbolic execution tool. We followed the work of S-gram to use a symbolic execution tool named Oyente to confirm vulnerabilities. According to the paper of Oyente, it has a low false positive rate of only 6.3%, which means the dataset still has a small part of sequences mislabeled. Even if it is labeled by humans manually, mislabeled data cannot be avoided either. But actually, on the other hand, using the same data improves comparability with the baseline.

3.5 Chapter Summary

In this chapter, we implement MSgram, where we design three tokenization methods to represent smart contract source code in multiple semantics, including aspects of structure, text, and combined one. To learn the semantics extracted above, we adopt N-gram Language Models (NLs) in this work. Besides, we also provide two optional combination strategies of *Intersection* and *Union* to exploit multi-semantics. Experimental results show that if developers tend to boost their auditing precision, adopting the *Intersection* strategy could be better. However, if they focus on detecting more true vulnerabilities, the *Union* strategy is a good choice. Afterward, we make a comprehensive comparison with the baseline, which demonstrates that the MSgram with multi-semantics outweighs the baseline in the performance of vulnerability detection. The MSgram with either the *Intersection* or the *Union* strategy not only obtains great improvements on their focusing metrics but also keeps stability or even excels on other metrics. In this case, we come to the conclusion that the novel concept of analyzing smart contracts based on multiple semantics indeed performs much better than that based on single semantics in vulnerability auditing.

Chapter 4

MMTrans: Multi-Modal Analysis with Transformer Structures

Code comment has been an important part of computer programs, greatly facilitating the understanding and maintenance of source code. However, high-quality code comments are often unavailable in smart contracts, the increasingly popular programs that run on the blockchain. In this chapter, we propose a Multi-Modal Transformer-based (MMTrans) code summarization approach for smart contracts. Specifically, the MMTrans learns the representation of source code from the two heterogeneous modalities of the Abstract Syntax Tree (AST), i.e., Structure-based Traversal (SBT) sequences and graphs. The SBT sequence provides the global semantic information of AST, while the graph convolution focuses on the local details. The MMTrans uses two encoders to extract both global and local semantic information from the two modalities respectively, and then uses a joint decoder to generate code comments. Both the encoders and the decoder employ the multi-head attention structure of the Transformer to enhance the ability to capture the long-range dependencies between code tokens. We build a dataset with over 300K

<method, comment> pairs of smart contracts, and evaluate the MMTrans on it. The experimental results demonstrate that the MMTrans outperforms the state-of-the-art baselines in terms of four evaluation metrics by a substantial margin, and can generate higher-quality comments. This work was published at the 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC) [134].

The remainder of this chapter is organized as follows. Section 4.1 presents the preliminary of MMTrans. Section 4.2 and Section 4.3 elaborate on the proposed approach and data preparation. Section 4.4 and Section 4.5 discuss the experiment design and results. Section 4.6 demonstrates the threats to validity. Finally, Section 4.7 concludes the work of this chapter.

4.1 Preliminary

4.1.1 Structure-based Traversal

Hu et al. [43] proposed the Structure-based Traversal (SBT) method, which converts the ASTs into specially formatted sequences by globally traversing the ASTs. Specifically, it applies the "type" and "value" of nodes to represent the structural and lexical information of code, respectively, and adopts a series of brackets to keep the AST structure to ensure the generated sequence is recoverable to the original AST. The SBT method was applied in some previous code summarization models, such as ast-attendgru [61], Dual Model [128], and Hybrid-DeepCom [44], and was proved its powerful ability in preserving code structural and lexical information. Therefore, we regard the SBT sequence as the one modality of AST, and adopt the SBT method to represent the global semantic (i.e., both structural and lexical) information as an input of the MMTrans. Taking the smart contract snippet in the Figure 4.1 as an example, the method named *_tokensToSell* is firstly

transformed to its AST format, and then the SBT sequence is further extracted from the AST. The non-leaf nodes are represented by their "type" (such as "FunctionDefinition" and "Block" in bold). For leaf nodes, they are represented by the format of "type_value" (such as "Visibility_private", etc.) in the original paper [43]. However, in our experiment, based on the original SBT sequences, we split the "type" and "value" for leaf nodes, and further split the camelCase and snake_case tokens of leaf nodes' "values" (such as from "_tokensToSell" to "_tokens", "To", and "Sell" in italic) to reduce the Out-of-Vocabulary (OOV) tokens, the detailed data processing methods are elaborated in Section 4.3.1.

4.1.2 Graph Convolutional Neural Network

The Graph Convolutional Neural Network (GCN) is designed for information propagation along the edges between nodes, where the hop, representing the layers of GCN, is a critical hyper-parameter. With the hop increasing, each node can aggregate a larger range of information from its neighbors, thereby focusing on a wider scope of local semantic information. Since the node embeddings in GCN includes both the "type" and "value" in ASTs, the GCN implies both the lexical and structural semantics of the AST integrated along the edges. Previous code summarization works, such as heterogeneous GCN [123] and code+gnn+GRU [60], also have proved the strength of GCN in locally distilling the semantic information from the AST and achieved promising results. Therefore, we regard the graph as another modality of AST and adopt it as a parallel input of the MMTrans relative to the SBT sequences. The AST (graph) in Figure 4.1 is an example generated from the method named *_tokensToSell*, and shows the hop of 1,2 and 3 of the root node (i.e., "FunctionDefinition"). Intuitively, it can aggregate the information from its neighbors of "SimpleName", "Visibility", "ReturnParameters" and "Block" by the convolution of the first hop; and it can also indirectly aggregate the more extensive

information from its children nodes' neighbors by increasing the hop number. More formally, the graph convolution process can be defined by the following layer-wise propagation rule:

$$H^{(l+1)} = \sigma(\tilde{A}H^{(l)}W^{(l)}), \tag{4.1}$$

where the H^l is the nodes embedding matrix at the layer l, the $\tilde{A} = A + I_N$ is the adjacency matrix A of a particular graph with added self-connection, I_N is the identity matrix, W^l is a layer-specific trainable weight matrix, and σ is the activation function [54].

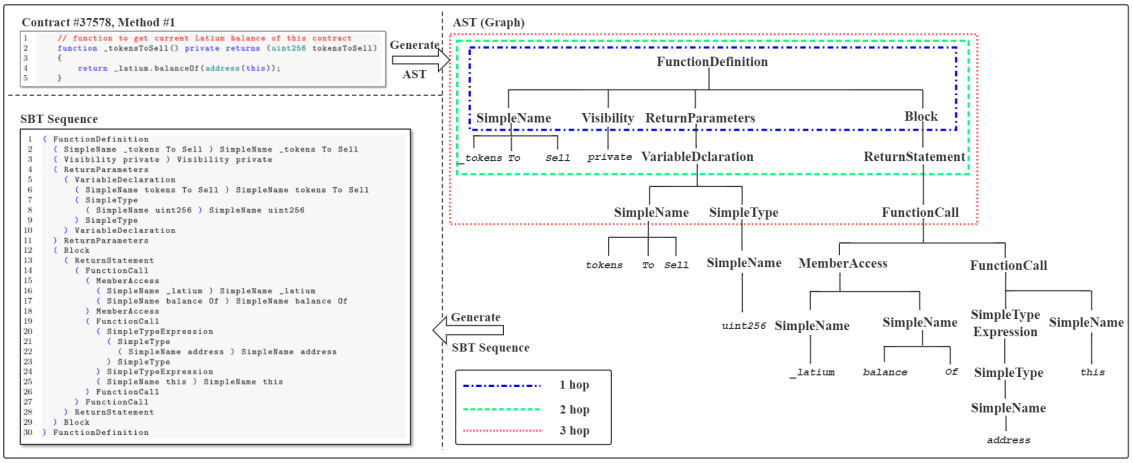


Figure 4.1: An example of smart contract method (_tokensToSell) with its AST (graph) and SBT sequence

4.1.3 Transformer-related Structures

4.1.3.1 Positional Encoding

Since the multi-head attention is not the recurrent structure, it needs the positional encoding to inject order information into the token embedding vectors. In this work, we follow one of the positional encoding approaches proposed by Vaswani et al. [120], which defines the specific pattern that model learns. This kind of positional encoding rule can be defined by

the equations 4.2 and 4.3, where *pos* is the token position in a sequence, i is the dimension index, and d is total dimension of the token embedding vector.

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d}) \tag{4.2}$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d}) \tag{4.3}$$

4.1.3.2 Multi-head Attention

In order to pay attention from different perspectives and capture the long-range dependencies in sequences, Vaswani et al. [120] also introduced the multi-head attention mechanism. The details are given by the following equations:

$$q_1, ..., q_J = split(QW^Q) \tag{4.4}$$

$$k_1, ..., k_J = split(KW^K) \tag{4.5}$$

$$v_1, ..., v_J = split(VW^V) \tag{4.6}$$

$$head_j = Softmax(\frac{q_j k_j^T}{\sqrt{d_k}})v_j,\ j = 1, ..., J \tag{4.7}$$

$$MultiHead(Q,K,V) = Concat(head_1, ..., head_J)W^o \tag{4.8}$$

Here, the $Q \in \mathbb{R}^{Q_l \times Q_d}$, $K \in \mathbb{R}^{K_l \times K_d}$ and $V \in \mathbb{R}^{K_l \times V_d}$ represent the matrices of query, key and value, respectively, while $q_j \in \mathbb{R}^{Q_l \times q_d}$, $k_j \in \mathbb{R}^{K_l \times k_d}$, $v_j \in \mathbb{R}^{K_l \times v_d}$ represent their splitted matrices for $head_j$. Specifically, $q_d = k_d = v_d = d_{model}/J$. The $W^Q \in \mathbb{R}^{Q_d \times d_{model}}$, $W^K \in \mathbb{R}^{K_d \times d_{model}}$, $W^V \in \mathbb{R}^{V_d \times d_{model}}$ are the three trainable weight matrices. The equation 4.7 describes the Scaled Dot-Product Attention output of $head_j$, where the d_k is the scaling factor equals to k_d. Finally, after the concatenating from all heads and the linear

transformation with $W^o \in \mathbb{R}^{Jv_d \times d_{model}}$, we obtain the output of the multi-head attention in equation 4.8[120].

4.1.3.3 Point-Wise Feed-Forward Networks

This is another module of Transformer that applied in [120]. It is composed of two dense layers with a ReLU activation function in between, which can be defined by the equation 4.9, where W_1 and W_2 are the weight matrices of each layer, b_1 and b_2 are their corresponding bias, and x is the input matrix. The dimensionality of the inner-layer d_{ff} is a hyper-parameter in [120].

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2 \tag{4.9}$$

4.2 Approach

The whole framework of the MMTrans illustrated in Figure 4.2 includes the three stages: the data processing, the MMTrans training, and the MMTrans testing. The source code we obtained from Etherscan.io is parsed and processed into a parallel corpus of smart contract methods and their corresponding comments. To comprehensively learn the semantic information of source code, we transform the smart contract methods to graphs (i.e., ASTs) and SBT sequences, respectively, as the inputs of the MMTrans. The MMTrans consists of two encoders (i.e., the graph encoder and SBT encoder) and a joint decoder. The node sequences and edges (i.e., their corresponding adjacency matrices) of graphs are fed into the graph encoder to learn the local semantic information, while the SBT sequences are fed into the SBT encoder to learn the global semantic information. Subsequently, in the training stage, the joint decoder integrates comment sequences, the

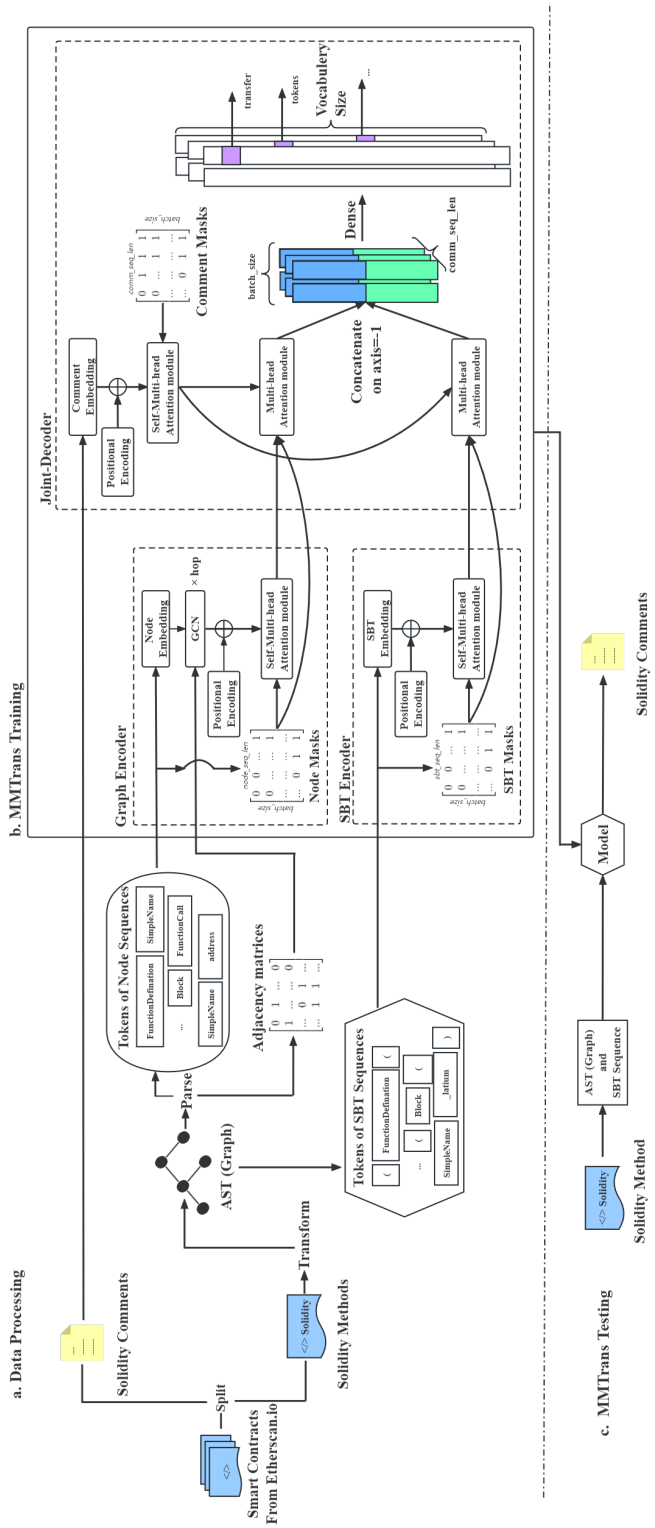


Figure 4.2: The overall framework of MMTrans

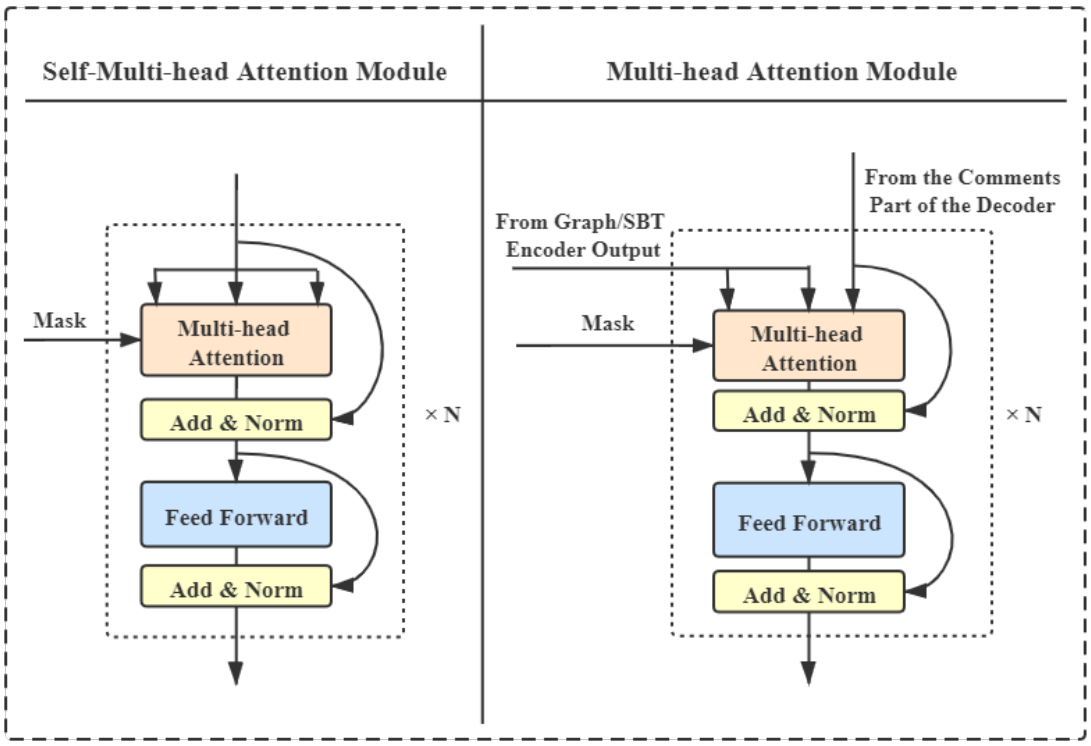


Figure 4.3: (Self) Multi-head attention module

graph encoder outputs, and the SBT encoder outputs to produce a batch of sentences under the teacher forcing method. Finally, the back-propagation is executed based on the predefined loss function to optimize the whole network. However, in the testing stage, the joint decoder integrates the previously generated comment words, and the above two encoder outputs to predict one word at each time step. It is noticeable that we do not include plain source code as one part of the inputs for the MMTrans to learn the lexical information of source code because each of the modalities of AST we adopt already contains both the lexical and structural information, as mentioned in 4.1.1 and 4.1.2.

4.2.1 Graph Encoder

Initially, for a batch of node sequences $X \in \mathbb{R}^{N_{batch} \times l}$, where l represents the maximum length of node sequences of this batch and N_{batch} represents the batch size, the graph encoder firstly embeds the node sequences X with the embedding size $d = 256$. Then, the GCN layer takes the embedding layer output and the edges $E \in \mathbb{R}^{N_{batch} \times l \times l}$ as the inputs to perform the graph convolution that we described in the Section 4.1.2. Previous work by LeClair et al. [60] has proved that the graph convolution layers $hop = 2$ is the best setting for the code summarization task, here we follow their setup and fix the $hop = 2$. Each node at the end of the GCN layer aggregates the neighboring information after the graph convolution. Since we adopt the pre-order traversal method to produce the node sequences from ASTs, the node sequences imply the original appearance order of the tokens in the source code, which is also the necessary information that needs to be considered. Therefore, we add the positional encoding matrices $PE \in \mathbb{R}^{N_{batch} \times l \times d}$ to the output of the GCN layer, so as to inject the position-wise information.

Subsequently, the output of the GCN layer is imported into the Self-Multi-head Attention Module (SMAM) to distill their semantic information further. The internal

structure of the SMAM is demonstrated in Figure 4.3. It includes the multi-head attention, layer-normalization, and point-wise feed-forward network (the d_{ff} of the network is set to 512), which are the principal parts of the Transformer introduced in Section 4.1.3. Initially, we set the head number $J = 4$, so that the multi-head attention can focus on each node sequence from four different representation subspaces [120]. Moreover, the d_{model} in the multi-head attention is set to 256, representing the width of this module; while the number of layers N, representing the depth of this module, is set to 1. Besides, we also adopt the node mask $M \in \mathbb{R}^{N_{batch} \times l}$ generated from the batch data to avoid distracting attention by <PAD> tokens. Finally, we obtain the output of SMAM $\hat{X} \in \mathbb{R}^{N_{batch} \times l \times d_{model}}$ at the end of the graph encoder, and the whole process can be described by the following equation, where the f is the abstract mapping function constructed by the graph encoder:

$$\hat{X} = f(X,\, E,\, PE,\, M) \tag{4.10}$$

4.2.2 SBT Encoder

For a batch of SBT sequences $X^{'} \in \mathbb{R}^{N_{batch} \times l^{'}}$, where the $l^{'}$ represents the maximum length of SBT sequences of this batch, the SBT encoder also firstly embeds the $X^{'}$ with the embedding size $d = 256$ and injects the position-wise information with $PE^{'} \in \mathbb{R}^{N_{batch} \times l^{'} \times d}$. Subsequently, the SBT encoder adopts the SMAM to extract the semantic information with the same hyper-parameters and uses the SBT mask $M^{'} \in \mathbb{R}^{N_{batch} \times l^{'}}$ to avoid distraction. Thereby, we obtain the final output of SBT encoder $\hat{X}^{'} \in \mathbb{R}^{N_{batch} \times l^{'} \times d_{model}}$. The equation below illustrates the abstract mapping function $f^{'}$ of the SBT encoder:

$$\hat{X}^{'} = f^{'}(X^{'},\, PE^{'},\, M^{'}) \tag{4.11}$$

4.2.3 Joint Decoder

Similarly, for a batch of comment sequences $Y \in \mathbb{R}^{N_{batch} \times l^Y}$, where the l^Y represents the maximum length of comment sequences of this batch, the joint decoder also firstly embeds the Y with the embedding size $d = 256$, and injects the positional information $PE^Y \in \mathbb{R}^{N_{batch} \times l^Y \times d}$. Then, the features of comment sequences are extracted by the SMAM with the same hyper-parameters. It should be noticed that the comment mask $M^Y \in \mathbb{R}^{N_{batch} \times l^Y \times l^Y}$ is an addition of a padding mask and a look-ahead mask (i.e., an upper triangular matrix), which is used for avoiding distraction and information leakage of the subsequent tokens in training [120].

Next, two Multi-head Attention Modules (MAMs) are introduced in the joint decoder with the same hyper-parameters as SMAM. The internal structure is shown in Figure 4.3. One for comment sequences and the output of graph encoder $\hat{X}$, another for the comment sequences and the output of SBT encoder $\hat{X}'$, thereby learning which tokens from the two encoders are important to the inference of comments, respectively. The outputs from the two MAMs are concatenated together on their last axis, representing a merge of their respective prediction for the comments. Finally, the merged output is fed into a linear transformation layer followed by a softmax function to produce the probability distribution over the vocabulary, thereby obtaining the final result $\hat{Y} \in \mathbb{R}^{N_{batch} \times l^Y \times S}$ of the MMTrans, where the S represents the comment vocabulary size. The outlined mapping function f^Y of the joint decoder can be summarized by the equation below:

$$\hat{Y} = f^Y(Y, \hat{X}, \hat{X}', PE^Y, M^Y) \tag{4.12}$$

Similar to most of the Seq2Seq models, we define the loss function for each batch as equation 4.13, where the l_i^y represents the length of the ith real comment (ground truth)

removed <PAD> tags, and $p(\hat{Y}_{ij}^{(z)})$ represents the probability that the jth token in the ith sample is the zth (z is the ground truth) word in the whole comment vocabulary.

$$Loss_{batch} = -\frac{1}{N_{batch}} \sum_{i=1}^{N_{batch}} \frac{1}{l_i^y} \sum_{j=1}^{l_i^y} log\ p(\hat{Y}_{ij}^{(z)}) \tag{4.13}$$

4.3 Data Preparation

4.3.1 Preprocessing

The raw dataset is provided by Zhuang et al. [144], which was collected from Etherscan.io. The dataset contains totally 40,932 Ethereum smart contracts written in solidity with 933,146 *normal methods*, 73,533 *modifiers*, and 12,482 *fallback methods*. *Normal methods* includes *constructors* and other *functional methods*; *Modifiers* are used to change the behavior of functions in a declarative way, such as checking a condition prior to executing the function; *Fallback methods* will be executed on a call to the contract, if none of the other methods match the given method signature; another kind of method in solidity is the *receive method*, which is also a kind of *fallback method*, and is first introduced in the version of solidity 0.6.0 to receive ether [1]. According to our preliminary study on the whole dataset, we find a 1-to-*n* matching problem between methods and comments, i.e., the same code may correspond to different comments among most of the *fallback methods*, which will confuse the MMTrans in training. The main reason is that most of the developers use *fallback methods* for reverting. But different *fallback methods* in the different smart contracts will revert different objectives. In addition, generating comments for *constructors* and *receive methods* is trivial because their functionalities are fixed and easy for machines to learn. Furthermore, there is also no

[1] https://docs.soliditylang.org/en/v0.6.0/index.html

receive method in the whole dataset. Therefore, in our experiment, we only consider normal *functional methods* and *modifiers*, and we remove the methods without comments. The remaining data are formulated as <method, comment> pairs for further processing.

According to the introduction of annotation in the solidity document [2] and our observation, we find that smart contract developers tend to place their comments under the NatSpec tags by the priority order of @*notice*, @*dev*, @*return* or just "//" and "/**/". Therefore, we extract the texts behind the @*notice* tag first, if there is no @*notice* tag, we extract the texts behind the @*dev* tag, and so on. In addition, following the prior similar work [44, 128, 43, 60], we also use the first sentence of texts as the ground truth comment, which typically describes the functionality of the particular method. Moreover, we remove those <method, comment> pairs whose comments contain less than 4 words for better computation of the BLEU-4 score [44]. Finally, we collect the 347,410 <method, comment> pairs from the 40,932 smart contracts.

4.3.2 Data Transformation

We transform the source code to SBT sequences and graphs (represented by XML format) respectively by utilizing a solidity parser [3]. To reduce the Out-Of-Vocabulary (OOV) tokens and facilitate the model to capture token representation, we split the camelCase and snake_case tokens for the "value" of leaf nodes in both SBT sequences and graphs. Further, we parse the graphs to their corresponding node sequences and edges (i.e., adjacency matrices), and formulate the <method, comment> pairs to (SBT sequence, node sequence, adjacency matrix, comment) tuples. The statistics of the lengths of the SBT sequences, graph node sequences, and comments are shown in Table 4.1. The

[2] https://docs.soliditylang.org/en/v0.6.0/index.html
[3] https://github.com/federicobond/solidity-parser-antlr

average length of the SBT sequences, graph node sequences, and comments are 241.44, 75.92, and 10.04, respectively. Here, we set the maximum length of comments to 20, and remove those tuples with comment lengths larger than 20 (i.e., retain 96.53% of the whole data). We also set the maximum length of SBT sequences and graph nodes to 600 and 200, and truncate those excessively long sequences. Afterward, we replace the *numeral* and *string* in source code with <NUM> and <STR> respectively. Since the *address* of smart contracts is a special object in smart contracts and is composed of fixed 40 hexadecimal digits, we generalize *address* constants by <ADDR>. Finally, for the SBT sequences and comments, we add <START> and <END> tokens to represent the start and end of sequences. However, for graph node sequences, we keep them intact, which is also a common operation in GCN [60].

Table 4.1: Statistics for data lengths

The Lengths of the SBT Sequences					
Avg.	Mode	Median	≤ 200	≤ 400	≤ 600
241.44	69	163	59.49%	85.69%	94.92%
The Lengths of the Graph Node Sequences					
Avg.	Mode	Median	≤ 100	≤ 150	≤ 200
75.92	21	52	76.77%	89.98%	95.27%
The Lengths of the Comments					
Avg.	Mode	Median	≤ 10	≤ 20	≤ 30
10.04	8	10	57.18%	96.53%	99.44%

4.3.3 Generating Vocabularies and Input Pipeline

The above procedures yield 317,680 tuples. We randomly select 90% (285,912 samples) of them for training, 5% (15,884 samples) of them for validation, and 5% (15,884 samples)

of them for testing. We remove duplicated samples in the validation set and the testing set that are already included in the training set to avoid data leakage, and finally retain 1,185 and 1,159 samples, respectively. Then, we generate vocabularies for the SBT sequences, the graph node sequences, and the comments, respectively on the training set. The size of the above vocabularies are 10441, 10431, and 13174, respectively. Furthermore, we set a <UNK> token in each of the above vocabularies to substitute the OOV tokens in the validation set and the testing set.

To prepare the data input pipeline, we randomly select 100 samples to form each batch without replacement. For each input tunnel (i.e., SBT sequences, graph node sequences, adjacency matrices, and comment sequences) of each batch, we append the special tag <PAD> to pad them to the maximum length of their batch.

4.4 Experimental Setup

In this section, we propose our research questions and the corresponding methodologies. Meanwhile, we elaborate on the baselines from the recent similar works, the evaluation metrics, and the experimental devices.

4.4.1 Research Questions

Our evaluation aim to verify if MMTrans adopting the multi-modalities of AST and the multi-head attention structure outperforms the state-of-the-art baselines, and why the above two points cause the outperformance of MMTrans. Based on these concerns, we propose the following Research Questions (RQs):

- **RQ1:** How effective is the MMTrans compared with the state-of-the-art baselines introduced in Section 4.4.3?

- **RQ2:** How does the head number of the multi-head attention structure affect the performance of the MMTrans?
- **RQ3:** How do dimensionalities of queries, keys, and values inside the attention mechanism influence the performance of MMTrans?
- **RQ4:** What are the advantages of using the multi-modalities of AST and the multi-head attention structure?

Inspired by the previous works [43, 44, 60], we make the first attempt of utilizing multi-modalities of AST (i.e., both SBT sequences and graphs) and the multi-head attention structure to construct the framework of dual encoders along with a joint decoder, thereby proposing the MMTrans. To this end, the RQ1 is put forward to evaluate the MMTrans against other state-of-the-art models in terms of the metrics in Section 4.4.4.

On the other hand, the head number in the multi-head attention structure has been shown to be important in attention allocation from different representation subspaces. Besides, dimensionalities of queries, keys, and values also play important roles in the effectiveness of the attention mechanism. To this end, we put forward the RQ2 and RQ3 in this work to explore how these highly associated hyper-parameters affect the MMTrans' learning in smart contracts code summarization.

Finally, the aim of the RQ4 is to investigate why the MMTrans is more effective in generating code comments and how its heterogeneous multi-modalities of AST and the multi-head attention structure contribute to the generation of higher-quality comments for smart contracts, respectively.

4.4.2 Methodology

To answer RQ1, we reproduced three baselines that are highly related to our work. Following LeClair et al.'s [60] experiment setup, we try to set the hyper-parameters of the baselines consistent with the MMTrans for a fair comparison. The detailed hyper-parameters setting for the MMTrans has been elaborated in Section 4.2. Besides, we adopt the same training strategy for each model during the experiment. Specifically, we set the maximum training epoch to 50. Each model is validated every 500 mini-batches on the validation set by sentence-level BLEU. We save the model with the best validation performance and adopt early stopping with the patience of 5 to avoid the model overfitting and save computation costs. Furthermore, we adopt Adam [53] as the training optimizer, and follow the learning rate decay schedule in [120]. After the above training operations for each model, we choose their respective best-performing model to evaluate on the testing set in terms of the metrics in Section 4.4.4 and report the comparison result in this chapter.

For the RQ2, we fix the other hyper-parameters of the MMTrans and set the head number $J = \{2,4,8,16,32\}$ respectively. For each adjustment of J, we retrain the MMTrans following the training strategy and report the experimental result in terms of the automated metrics in this chapter. The reason for the choice of the above head numbers is that they should be divisible by d_{model}, which is a necessary prerequisite for the multi-head attention structure.

For the RQ3, similar to the previous RQ, we still fix other hyper-parameters and tune the dimensionalities of queries, keys, and values. Considering the computation overhead, we consistently tune their dimensionalities in the range of $\{64,128,192,256,320,384,448,512\}$ for eight trials. It should be noticed that their

dimensionalities should be the same for each experiment owing to the purpose of dot product computation.

Finally, for the RQ4, we evaluate the strength of the heterogeneous multi-modalities of AST and the multi-head attention structure by ablation experiments separately. (1) Firstly, we construct an incomplete-MMTrans (i-MMTrans) utilizing the graph and plain source code as double inputs. Then, we adopt i-MMTrans to compare with the third baseline (i.e., vanilla-Transformer with only the plain source code input) and the MMTrans with both SBT sequences and graphs as inputs, thereby exploring the contribution of SBT sequences and graphs, respectively. (2) However, for the evaluation of the strength of the multi-head attention structure, we adopt the comparison between i-MMTrans and the second baseline, i.e., code+gnn+GRU, because they both adopt the graph and plain source code as double inputs. But i-MMTrans adopts the multi-head attention structure, while code+gnn+GRU adopts the GRU-based structure.

4.4.3 Baselines

We compare the MMTrans with the three baselines that are published last year and directly related to our work. We list their detailed information below:

(1) Hybrid-DeepCom: Hu et al. [44] exploited the plain source code and SBT sequence with only nodes' "type" as the double inputs to extract structural and lexical information of source code, respectively. They also construct a GRU-based Seq2Seq model to process the double inputs and generate comments of Java methods. This approach was published in the EMSE volume 25, 2020, and is the latest representative approach that adopts SBT sequences as input separately from the plain source code.

(2) code+gnn+GRU: LeClair et al. [61] proposed to utilize the ASTs (graphs) and the plain source code as the double inputs to improve the code summarization performance

based on their previous work [61] published in the ICSE 2019. This approach was presented in the ICPC'2020 and is the latest work adopting GCN in the code summarization task.

(3) Vanilla-Transformer: Ahmad et al. [2] proposed to adopt the Transformer to solve the Java source code comments generation task, which is the first attempt of utilizing the Transformer in this field. This work was published in the ACL 2020. We adopt their base model (i.e., Vanilla-Transformer) to involve in the comparison, rather than their full model with the relative positional encoding and copy attention, because these tricks make it difficult to distinguish the improvement of multi-modalities of AST on transformer-based approach.

Since those baselines are all applied to Java methods, we change their inputs as our smart contracts data and follow their data processing steps to prepare their inputs. Besides, we adopt the greedy search algorithm in inference for all approaches to save the computation cost and ensure a fair comparison. Other noteworthy recent works, such as Wang et al. [127], Zhang et al.[139], adopted the Reinforcement Learning (RL) and Information Retrieval (IR) techniques based on recurrent models to improve the model performance, respectively. Since the second improvement of the MMTrans focuses on the structural upgrade by the Transformer towards the recurrent models, the contribution of the Transformer-based structure towards the recurrent models cannot be distinguished from the RL and IR techniques based on recurrent models. Therefore, the above two works are not suitable for comparison in this experiment. However, a potential future direction is to study the effect of the RL and IR techniques combined with the Transformer-based structure in another separate experiment.

4.4.4 Metrics

We adopt BiLingual Evaluation Understudy (BLEU) [4] [91], Recall-Oriented Understudy for Gisting Evaluation (ROUGE) [5] [68], and Metric for Evaluation of Translation with Explicit ORdering (METEOR) [4] [21] to evaluate the performance of the code summarization approaches.

- We report a composite BLEU score, which is the average of BLEU-1, BLEU-2, BLEU-3, and BLEU-4 (BLEU-n is the n-gram precision of a candidate sequence to the reference). Following Hu et al.'s work [44], we adopt both sentence-level BLEU (S-BLEU) and corpus-level BLEU (C-BLEU) to evaluate the generated comments, respectively. The S-BLEU calculates the composite BLEU score according to the sentence, and the C-BLEU calculates the composite BLEU score according to the whole corpus. In addition, to avoid non-overlapping n-grams in sentences, we simply use the smoothing-1 method [15] to assist the computation of S-BLEU.

- ROUGE evaluates how much reference text appears in the generated text, which can be thought of as a recall score. Following LeClair et al.'s work [60], we adopt the ROUGE-LCS (Longest Common Sub-sequence) F1 to evaluate the generated comments on the LCS matching degree between generated comments and references.

- METEOR is also a widely used recall-oriented metric in machine translation and code summarization tasks. It evaluates how well the generated comments capture content from the references via recall, which is computed by stemming and synonymy matching.

[4] NLTK: (`https://www.nltk.org/`) is used to calculate BLEU and METEOR
[5] rouge: (`https://github.com/pltrdy/rouge`) is used to calculate ROUGE

4.4.5 Experimental Device

The experiments are conducted on a Ubuntu GPU server with four RTX2080ti GPUs of 11 GB memory for each. Our proposed MMTrans is constructed by Tensorflow 2.3 based on CUDA 10.1 and cuDNN 7.4.

Table 4.2: Automated metrics evaluation results for the baselines and MMTrans

Baseline	**head number**	**S-BLEU (%)**	**C-BLEU (%)**	**ROUGE-LCS F1 (%)**	**METEOR (%)**
Hybrid-DeepCom	/	19.23	21.0	38.55	29.81
code+gnn+GRU	/	21.93	24.52	32.50	35.67
Vanilla-Transformer	4	25.99	28.79	47.36	39.25
Ours					
i-MMTrans	4	28.67	30.77	48.76	41.51
MMTrans	2	23.57	26.63	43.45	35.37
MMTrans	4	**30.47**	**34.14**	**50.57**	**43.24**
MMTrans	8	29.68	33.19	50.50	43.26
MMTrans	16	22.27	24.91	42.71	34.38
MMTrans	32	26.97	30.41	47.38	39.67

$^{\Phi}$ S-BLEU represents the Sentence-level BLEU score; C-BLEU represents the Corpus-level BLEU score.

4.5 Results

This section reports the experimental results for the research questions proposed in Section 4.4.1.

4.5.1 RQ1: Quantitative Evaluation

Table 4.2 shows the performance of the MMTrans ($J = 4$) and the compared baselines. As shown in the table, the MMTrans performs the best and achieves an S-BLEU score of 30.47, a C-BLEU score of 34.14, a ROUGE-LCS F1 score of 50.57, and a METEOR score of 43.24. Specifically, the MMTrans outperforms Vanilla-Transformer by 17.23% in terms of S-BLEU, by 20.74% in terms of C-BLEU, by 6.78% in terms of ROUGE-LCS F1, and by 10.17% in terms of METEOR; the MMTrans also outperforms code+gnn+GRU by 38.94%, 39.23%, 55.60%, and 21.22% in terms of the four metrics, respectively; and the MMTrans outperforms Hybrid-DeepCom by 58.45%, 62.49%, 31.18%, and 45.05% in terms of the four metrics, respectively. Towards the great improvement of the MMTrans against the three baselines, we generally attribute it to the multi-head attention structure and the utilization of both GCN and SBT to capture the local and global semantic information of code. We will discuss the strength of the MMTrans in more depth in Section 4.5.4. We also find that code+gnn+GRU outperforms Hybrid-DeepCom in terms of S-BLEU, C-BLEU, and METEOR, while the latter performs better in terms of ROUGE-LCS F1. This may owe to the higher attention of GCN on specific tokens and structures [60], leading to the higher score in terms of the gram-statistic-oriented metrics (i.e., S-BLEU, C-BLEU, and METEOR). However, SBT sequences represent the global semantic information of source code, therefore, Hybrid-DeepCom can capture the global semantic information and performs better in terms of the longest common sub-sequence matching (measured by ROUGE-LCS F1). Moreover, even though Vanilla-Transformer only adopts the plain source code as the single input, its performance is still better than that of Hybrid-DeepCom and code+gnn+GRU by a relatively large margin, which indicates the powerful capability of the Transformer model.

Answer to RQ1

The MMTrans outperforms Hybrid-DeepCom, code+gnn+GRU, and Vanilla-Transformer by a significant margin in terms of all the automated evaluation metrics.

4.5.2 RQ2: Head Number Analysis

As mentioned in Section 4.4.1, the head number J is the prominent part of the multi-head attention module compared with other attention structures and affects the model performance to a great extent. In order to explore the influence of the J in the MMTrans, we fix the other hyper-parameters and tune the $J = \{2, 4, 8, 16, 32\}$, respectively. The automated metric evaluation results are presented in Table 4.2. The whole experiment result presents a general trend of increasing first and then decreasing in terms of each of the evaluation metrics. And the performance reaches the peak when the $J = 4$. With the J increasing, more representation subspaces of tokens appear, but the embedding dimension of tokens decreases. As such, a potential explanation is that increasing the head number indeed can focus on more different perspectives; but when the subspaces proliferate excessively, the low representation dimension makes it impossible to accurately describe tokens, thus leading the attention distraction. Also, we notice that using head number $J = 32$ outperforms the $J = 16$. This outlier may be due to random initialization or other minor factors.

Answer to RQ2

When tuning the head number $J = \{2, 4, 8, 16, 32\}$, there is a general performance trend of increasing first then decreasing, and the MMTrans performs the best when the head number is 4.

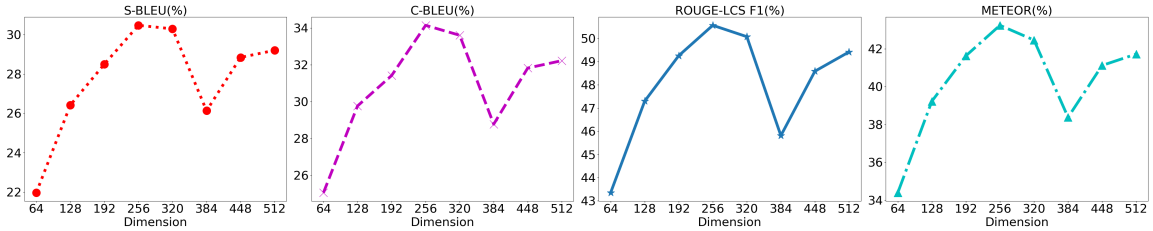


Figure 4.4: The performance variation of MMTrans on each evaluation metric with the tuning of dimensionality

4.5.3 RQ3: Dimensionality Analysis

The dimensionalities of queries, keys, and values of the attention mechanism have also been important hyper-parameters for their significant influence on attention-based neural models' performance. Hence, we fix other hyper-parameters and tune dimensionalities in the range of $\{64, 128, 192, 256, 320, 384, 448, 512\}$ as we mentioned in Section 4.4.2. Table 4.4 demonstrates the performance variation of MMTrans on each evaluation metric with the tuning of dimensionality among the range of 64 to 512. In general, with increasing dimensionalities of queries, keys, and values, the performance of MMTrans on each evaluation metric presents a trend of rising first and then declining consistently. When the dimensionality reaches 256, MMTrans achieves its best-performing status. The explanation is also straightforward: with the increasing dimensionalities of queries, keys, and values, the semantics of tokens become more meaningful and diverse. Therefore, MMTrans can allocate attention among tokens more accurately. However, when dimensionalities of queries, keys, and values exceed a peak point, MMtrans starts to overfit the training set. Thus, its performance declines.

Answer to RQ3

When tuning the dimensionalities of queries, keys, and values among the range of $\{64, 128, 192, 256, 320, 384, 448, 512\}$, there is a general performance trend of rising first then declining, and the MMTrans performs the best when the dimensionality is set to 256.

4.5.4 RQ4: Strength of the MMTrans

This section elaborates on the strength of MMTrans from two perspectives, i.e., heterogeneous multi-modalities of AST and the multi-head attention structure.

(1) Vanilla-Transformer uses the plain source code as a single input; i-MMTrans has double inputs of graphs and plain source code, while the MMTrans has double inputs of graphs and SBT sequences. Moreover, these approaches all employ the multi-head attention structure. As shown in Table 4.2, we find that i-MMTrans outperforms Vanilla-Transformer by 10.31% in terms of S-BLEU, by 6.88% in terms of C-BLEU, by 2.96% in terms of ROUGE-LCS F1, and by 5.76% in terms of METEOR, which indicates that the additional input of graphs indeed boosts the model performance. Subsequently, we make a comparison between i-MMTrans and MMTrans. Table 4.2 demonstrates that the latter outperforms the former by 6.28%, 10.95%, 3.71%, and 4.17% in terms of the four metrics, respectively, which proves that the additional input by SBT sequences can further improve the model performance.

Meanwhile, we present the first two instances to intuitively show the strength of the heterogeneous multi-modalities of AST in Table 4.3. Following LeClair et al.'s work [60], we intuitively illustrate the power of multi-modalities of AST from the copy mechanism perspective. For the first instance, i-MMTrans can directly copy the correct words, such as

Table 4.3: Examples of generated comments by each approach

ID	Smart Contract Methods	Comments

1	// Contract #32527, Method #7
	modifier whenCrowdsaleNotEnded {
	require(deadline >= now);
	_;
	}
		Vanilla-Transformer: modifier to make a function callable only when the contract is not finalized.
		i-MMTrans: modifier to make a function callable only when the crowdsale has not ended.
		MMTrans: modifier to allow actions only when the crowdsale is not ended.
		Reference: modifier to allow actions only when the crowdsale has not ended.

2	// Contract #37578, Method #1
	function _tokensToSell() private returns (uint256 tokensToSell) {
	return latium.balanceOf(address(this));
	}
		Vanilla-Transformer: function to sell tokens.
		i-MMTrans: function to sell tokens (with decimals) that we are selling tokens.
		MMTrans: function to get amount of Latium tokens (with decimals) of this contract.
		Reference: function to get current Latium balance of this contract.

3	// Contract #6743, Method #27
	function finalize() public inState(State.Success)
	onlyOwner stopInEmergency {
	if(finalized) throw;
	if(address(finalizeAgent) != 0) finalizeAgent.finalizeCrowdsale();
	finalized = true;
	}
		code+gnn+GRU: finalize a succcesful crowdsale.
		i-MMTrans: finalize a succcesful crowdsale.
		Reference: finalize a succcesful crowdsale.

4	// Contract #16118, Method #5
	function playerMakeBet(uint minRollLimit, uint maxRollLimit,
	bytes32 diceRollHash, uint8 v, bytes32 r, bytes32 s) public payable
	gameIsActive betIsValid(msg.value, minRollLimit, maxRollLimit) {
	if (playerBetDiceRollHash[diceRollHash] != 0x0
	|| diceRollHash == 0x0) throw;
	tempBetHash = sha256(diceRollHash, byte(minRollLimit),
	byte(maxRollLimit), msg.sender);
	if (casino != ecrecover(tempBetHash, v, r, s)) throw;
	. . .
	playerProfit[diceRollHash] = getProfit(msg.value, tempFullprofit);
	if (playerProfit[diceRollHash] > maxProfit) throw;
	. . .
	LogBet(diceRollHash, playerAddress[diceRollHash],
	playerProfit[diceRollHash], playerToJackpot[diceRollHash],
	playerBetValue[diceRollHash], playerMinRollLimit[diceRollHash],
	playerMaxRollLimit[diceRollHash]);
	}
		code+gnn+GRU: appends the bid's.
		i-MMTrans: public function player submit bet only if game is active bet is valid can be called.
		Reference: public function player submit bet only if game is active bet is valid.

"modifier", "when", "crowdsale", and "ended", from the source code to the comment. Compared with Vanilla-Transformer, its copy capability is indeed more powerful. However, i-MMTrans seems relatively weaker in summarizing the main idea of source code, when the incorrect words account for a relatively large ratio. Noticing the second instance, i-MMTrans wrongly copies the word "sell" and "tokens", which appear with the highest frequency in the source code. Therefore, it does not capture the main idea of the code snippet. A potential explanation is i-MMTrans equipped with GCN puts more focus on local features of the AST, causing it easy to be attracted by the specific words and code structure. However, MMTrans equipped with both the GCN and SBT can weigh between local and global semantic information, therefore it modifies the fault caused by i-MMTrans, and catches the correct main idea of the source code in the second instance.

(2) As mentioned in Section 4.4.2, i-MMTrans and code+gnn+GRU both exploit the graph and plain source code as the double inputs, but the former adopts the multi-head attention structure while the latter adopts the GRU-based structure. Statistically, i-MMTrans outperforms code+gnn+GRU by 30.73% in terms of S-BLEU, by 25.49% in terms of C-BLEU, by 50.03% in terms of ROUGE-LCS F1, and by 16.37% in terms of METEOR, which indicates that applying the multi-head attention structure indeed improves the performance a lot in the source code summarization task. And it is noticeable that the contribution to the performance improvement of the multi-head attention structure is greater than that of the multi-modalities of AST. We also list two examples of the generated comments in Table 4.3. The instance #3 is a short method that can be parsed to the plain source code with 49 tokens and the graph node sequence with 59 tokens. The i-MMTrans and code+gnn+GRU can both generate the exact correct comments on this short sample. Nevertheless, when it comes to a relatively long method, such as the instance #4 with 152 tokens in the plain source code and 200 tokens in the

graph node sequence (both have been truncated to the maximum length of the above two sequences), code+gnn+GRU with the GRU-based structure cannot summarize the correct comment, while i-MMTrans can still generate readable and meaningful comment. The reason is that the multi-head attention structure can properly allocate different attention weights on the tokens at each time step, and summarize their key information for inference; however, the GRU-based network has a limited capability in capturing the long-range dependencies between code tokens, thus generating the low-quality comments for relatively long methods.

To better demonstrate the multi-head attention's capability in capturing long dependencies between tokens, we additionally selected one example (shown in Table 4.4) and list its attention matrix between its comment tokens (rows) and corresponding SBT sequence tokens (columns) in Figure 4.5. This attention matrix locates in the last layer of the joint decoder. The SBT sequence of this function contains 416 tokens which can be seen as a long sequence, where, for clarity, we only present tokens in the range of 60 – 180. It is obvious that the token "Decrease" in the comment obtains more attention from the token "substracted" at the 65th, 69th, 163rd, and 168th positions, where these tokens carry similar meanings. For other tokens in the comment, they also can obtain different attention from tokens in the SBT sequence allocated by the multi-head attention mechanism. Although not all of their attention allocation is explainable, as the attention mechanism is not always consistent with human cognition [57] but guided by the training objectives, tokens in the comments can always fetch attention from the whole SBT sequence via the multi-head attention mechanism, showing that the long-range dependencies among code tokens can be captured in MMTrans.

Table 4.4: An example for illustrating attention matrices

Contract #17818, Method #3

```
//Decrease the amount of tokens that an owner allowed to a spender.
function decreaseApproval(address spender, uint256 subtractedValue)
  public returns (bool success) {
    uint256 oldValue = allowed[msg.sender][spender];
    if (subtractedValue >oldValue) {
        allowed[msg.sender][spender] = 0;
    } else {
        allowed[msg.sender][spender] = safeSub(oldValue, subtractedValue);
    }
    Approval(msg.sender, spender, allowed[msg.sender][spender]);
    return true;
    }
```

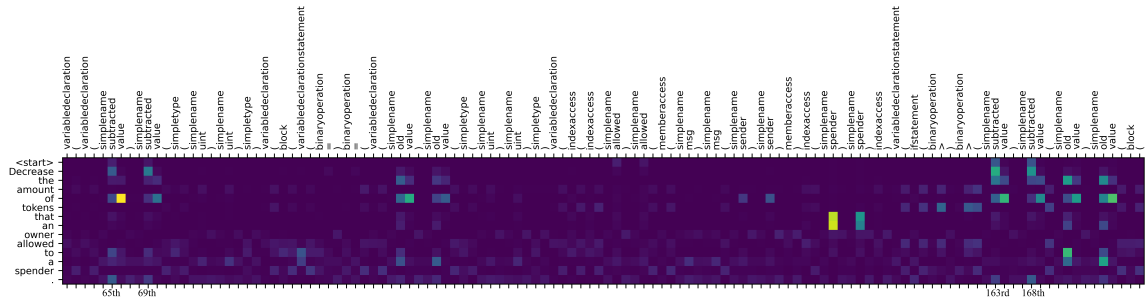


Figure 4.5: An attention matrix example

Answer to RQ4

Leveraging SBT sequences and graphs to extract the global and local semantic information of code, and employing the multi-head attention structure to capture the long-range dependencies between code tokens contribute to the generation of higher-quality comments.

4.6 Theats to Validity

We have identified the following threats to validity:

Dataset quality: As the first smart contract code summarization work, we used some heuristic rules to extract the <method, comment> pairs according to the characteristic of smart contract data. Although we did rigorous data processing, there may be still some noise. We will continuously refine and update the version of the open-source dataset.

Comparison on smart contract dataset: Since our work focuses on the smart contract code summarization, we did not compare the MMTrans with Hybrid-DeepCom, Vanilla-Transformer, and code+gnn+GRU on their datasets. But the results on smart contracts also have proved the effectiveness of the MMTrans. In the future, we will extend our experiment to other programming languages (e.g., Java and Python).

Fair comparison threat: Due to the hardware limitations, we were unable to conduct fully extensive hyper-parameters optimization for all baselines. Following LeClair et al.'s work [60], we try to mitigate the impact of this issue by making each baseline's hyper-parameters consistent with our model.

Automated evaluation: We adopt four automated evaluation metrics that have been widely used in previous code summarization studies [43, 60, 44, 2]. Although the metrics are not representative of human judgment [108], they can evaluate the performance of code

summarization models quickly and quantitatively. In the future, we will conduct a human evaluation for models.

4.7 Chapter Summary

This work aims to help programmers comprehend the meaning of smart contract code by automatically generating high-quality comments. To tackle this task, we, for the first time, collect a smart contract code summarization dataset with 347,410 <method, comment> pairs. Meanwhile, we propose a code summarization approach named MMTrans, which leverages the two modalities of the AST (i.e., SBT sequences and graphs) to represent both global and local semantic information of source code, then employs the two encoders and a joint decoder with the multi-head attention structure to capture the long-range dependencies between code tokens. The comprehensive experiments on the collected dataset show that the MMTrans performs better than the state-of-the-art baselines by a significant margin and can generate higher-quality comments in the practical tests.

Chapter 5

CBS: Classifying Before Synchronizing for Code-Comment Inconsistencies

Software comments sometimes are not promptly updated in sync when the associated code is changed. The inconsistency between code and comments may mislead the developers and result in future bugs. Thus, studies concerning code-comment synchronization have become highly important, which aim to automatically synchronize comments with code changes. Existing code-comment synchronization approaches mainly contain two types, i.e., (1) deep learning-based (e.g., CUP) and (2) heuristic-based (e.g., HebCUP). The former constructs a neural machine translation-structured semantic model, which has a more generalized capability of synchronizing comments with software evolution and growth. However, the latter designs a series of rules for performing token-level replacements on old comments, which can generate the completely correct comments for the samples fully covered by their fine-designed heuristic rules. In this chapter, we propose a composite approach named CBS (i.e., Classifying Before Synchronizing) to further improve the code-comment synchronization performance,

which combines the advantages of CUP and HebCUP with the assistance of inferred categories of Code-Comment Inconsistent (CCI) samples. Specifically, we first define two categories (i.e., heuristic-prone and non-heuristic-prone) for CCI samples and propose five features to assist category prediction. The samples whose comments can be correctly synchronized by HebCUP are heuristic-prone, while others are non-heuristic-prone. Then, CBS employs our proposed Multi-Subsets Ensemble Learning (MSEL) classification algorithm to alleviate the class imbalance problem and construct the category prediction model. Next, CBS uses the trained MSEL to predict the category of the new sample. If the predicted category is heuristic-prone, CBS employs HebCUP to conduct the code-comment synchronization for the sample. Otherwise, CBS allocates CUP to handle it. Our extensive experiments demonstrate that CBS statistically significantly outperforms CUP and HebCUP, and obtains an average improvement of 23.47%, 22.84%, 3.04%, 3.04%, 1.64%, and 19.39% in terms of Accuracy, Recall@5, Average Edit Distance (AED), Relative Edit Distance (RED), BLEU-4, and Effective Synchronized Sample (ESS) ratio, respectively, which highlights that category prediction for CCI samples can boost the code-comment synchronization performance. This work was published at ACM Transactions on Software Engineering and Methodology [136].

The remainder of this chapter is organized as follows: Section 5.1 clarifies the background of code-comment synchronization, including the problem formalisation, the usage scenario, the state-of-art approaches, and a code-comment synchronization example. Section 5.2 empirically explores the existing code-comment synchronization approaches. Section 5.3 describes the methodology of our proposed CBS approach. Section 5.4 and 5.5 present the experimental results for category prediction and code-comment synchronization, respectively. Section 5.6 explores the effect of the inside structures of our proposed approach. Section 5.7 discusses the failures of CBS and

summarizes the implications of this work. Section 5.8 presents threats to validity. Finally, Section 5.9 concludes this chapter.

5.1 Background

In this section, we introduce the background of code-comment synchronization, including its problem formalization, usage scenario, state-of-the-art methods, and an example.

5.1.1 Problem Formalization

Code-comment synchronization aims to synchronize comments when the associated code changes. More formally, given a piece of code change, including a pre- and a post-change version of the code snippet, namely c and $c^{'}$, as well as a pre- and a post-change version of the comment associated with it, namely x and y, the objective of this research is to find a function f (i.e., a kind of code-comment synchronization approach), such that the post-change comment y can be correctly produced from f on condition of given c, $c^{'}$, and x as below.

$$y = f(c, c^{'}, x) \tag{5.1}$$

Hereon, the c, $c^{'}$, x, and y are referred to as old code, new code, old comment, and new comment (it is also named as reference comment in other studies), respectively. For each triple composed of c, $c^{'}$, and x, we refer to it as a Code-Comment Inconsistent (CCI) sample in this chapter. The associated y is our target and is not available during the code-comment synchronization in practice. We cope with this problem by proposing a CBS approach to approximate f and refer to each comment outputted from CBS as a synchronized comment $\hat{y}$.

5.1.2 Usage Scenario

We illustrate the usage scenario of CBS as follows:

Firstly, CBS can promptly and automatically help developers synchronize the associated comment when making a code change. If the synchronized comment is correct or partially correct, it can greatly reduce the workloads of developers on manually editing comments and make them concentrate more on the code implementation of transaction logic, thereby improving productivity. Besides, this process also effectively avoids the obsolete comments that are ignored to update by developers during their programming.

Secondly, CBS is also useful in repairing the existing bad comments. For example, when developers are programming based on others' code, they may encounter code snippets with bad comments, which will mislead their understanding of the code's intention. In this case, developers can adopt a bad comment detector, e.g., the tool proposed by Liu et al. [71], to identify comments that need to be synchronized, then CBS can be applied to them to automatically repair those comments and make them synchronous with their associated code snippets.

The usage of CBS can also be quite frequent. According to the study of Kolassa et al. [56] on 11,143 open-source projects with a total of 8,705,118 commits by 47,548 committers, over 50% of developers commit twelve times daily. Besides, we also conducted a preliminary analysis on the 1,496 Java projects in this chapter, and we found that each commit is related to, on average, 2.2 method-level code changes. Hence, CBS, as a method-level code-comment synchronizer, can be utilized by developers very often.

5.1.3 Current Code-Comment Synchronization Approaches

CUP and HebCUP are two representative state-of-the-art code-comment synchronization approaches. Here, we introduce them below:

- **CUP** is a deep learning-based model proposed by Liu et al. [73]. They describe the code-comment synchronization task as a machine translation process and design an LSTM-based Sequence-To-Sequence (Seq2Seq) variant to deal with the mapping from old comments and code changes to new comments. Specifically, the semantics of code change sequences and old comment sequences are progressively extracted through the layers of embedding, co-attention, and modeling. Afterward, the high-level representation of code change sequences and old comment sequences are decoded with the previously generated tokens of new comments via the dot-production attention mechanism to generate a new comment token at each time step.

- **HebCUP** is a heuristic-based approach proposed by Lin et al. [66]. They put forward a series of heuristic rules by empirically studying CUP successful and failure cases. HebCUP can successfully handle most code-indicative changes, whose changed sub-tokens or tokens in comments can be found from the corresponding code change contents [66]. Specifically, HebCUP first aligns the sub-tokens in code change sequences according to its sub-token alignment algorithm and constructs a series of replacement pairs based on the alignment results. Each replacement pair can be represented as a format of *key-value*, where the *key* is the old token while the *value* is the potential new token for replacement. Finally, HebCUP adopts *keys* to match tokens in old comments and conduct the updating with *values*, thereby conducting the synchronization for comments.

5.1.4 An Example of Code-Comment Synchronization

We present a code-comment co-change instance in Listing 5.1, where the comment in red is the old comment describing the functionality of the old code, the comment in green is

the new comment describing the functionality of the new code, code in red is the deleted statement, code in green is the newly added statement, and other code is no-change statements. Old code is composed of no-change statements and deleted statements while new code is composed of no-change statements and newly added statements. Given a triple of old code, new code, and old comment, a code-comment synchronizer (e.g., CUP and HebCUP) is able to provide a comment suggestion describing the new code, and we refer to such suggestion as a synchronized comment. Taking Listing 5.1 as an example, CUP will generate a comment suggestion of "Return the PIM Interface" for it. Subsequently, this synchronized comment will be evaluated in terms of the metrics we illustrate in Section 5.2.2 via making a series of comparisons between itself and the reference comment (i.e., "Return the ONOS Interface"), thereby assessing the quality of each synchronized comment.

Listing 5.1: A code-comment co-change instance

```
1 - /*Get the PIM Interface.*/
2 + /*Return the ONOS Interface.*/
3    public  Interface   getInterface ()  {
4 -              return theInterface;
5 +              return this.onosInterface;
6           }
```

5.2 Exploring Existing Code-Comment Synchronization Methods

In this section, we experimentally investigate the performance of CUP and HebCUP to explore their respective advantages and their heterogeneous characteristics.

5.2.1 Dataset

The dataset we adopt in this work is extracted from 1,496 Java repositories hosted on GitHub, where most of them are famous projects, such as the nomulus [1] of Google, the Hive [2] of Apache, and Fresco [3] of Facebook. The dataset is built by Wen et al. [129] and first applied in the code-comment synchronization field by Liu et al. [73]. Subsequently, Lin et al. [66] further cleaned the dataset. According to Liu et al. [73], they deduplicated the dataset and arranged it in ascending order according to the commit creation time within each project. The first 80% of commits were put into the training set, and the remaining 20% of commits were shuffled and evenly split into the validation and test set, thereby avoiding the data leakage problem. Subsequently, Lin et al. [66] further cleaned the dataset and obtained 80,591, 8,827, and 9,204 code-comment co-change instances in the training, validation, and testing sets, respectively. For each instance, the old code, new code, and old comment constitute a CCI sample in practice, and the new comment is regarded as the reference comment for synchronization. Code and comments are tokenized to sequences, and their tokens are broken into sub-tokens based on camel casing and snake casing. Besides, "<*con*>" is also inserted to join the sub-tokens for future

[1] https://github.com/google/nomulus
[2] https://github.com/apache/hive
[3] https://github.com/facebook/fresco

recovery. For example, the token "*getInterface*" is tokenized to two sub-tokens "*get*" and "*Interface*", and "<*con*>" is inserted to join the two sub-tokens.

In particular, according to Liu et al. [73] who construct the original version of this dataset, if the word-level Levenshtein distance [85] of a comment pair (i.e., an old comment and its associated new comment) is larger than the old comment's length and 5, they regarded this pair as a rewrite instead of an update, where the word-level Levenshtein distance is minimum word edits required to change a sentence into the other. This makes sense because if the distance of a comment pair is too large, it has a great possibility that there is little relation between the pre- and post-change versions of code and comments, which should better use code summarization approaches to generate comments from scratch. On the other hand, if we still use code-comment synchronization approaches to deal with them, the synchronized comments may not be as good as we expected. Because the relation between old and new code (comment) has almost disappeared. Therefore, setting some restrictions on the distances of comment pairs is reasonable. In addition, according to the study of Liu et al. [73], before they conducted this filtering procedure, they collected a dataset with 242,649 samples. After the filtering, there still remain 147,844 samples which still account for over 60% of the original samples. Therefore, these samples with relatively small changes indeed occur very often in code changes, whereas the larger changes are the minorities. Thus, this dataset is practical.

Table 5.1 presents the statistics of the dataset we adopted, where Length is the number of sub-tokens in each type of data sequence, and Number of Changed Sub-tokens (NCS) records the number of sub-tokens that are modified between the old and new code (comments). According to the statistical results in Table 5.1, we find that the Lengths of code (comments) among each part of the dataset are similar on both the mean and standard deviation. Besides, the same phenomenon can also be observed in the NCS. It

indicates there is no significant difference in the data distributions among the training, validation, and testing sets.

Table 5.1: The statistics of the datasets

Dataset	**Type**	**Length**			**NCS**		
		Mean	Standard Deviation	Median	Mean	Standard Deviation	Median
Train	Old Code	91.22	84.42	60.0	34.73	57.49	11.0
	New Code	91.26	84.98	58.0			
	Old Comment	14.43	7.34	13.0	3.81	3.01	3.0
	New Comment	14.47	7.63	13.0			
Validation	Old Code	89.60	82.61	59.0	32.73	54.66	12.0
	New Code	88.77	82.66	57.0			
	Old Comment	15.16	7.57	14.0	3.83	3.03	3.0
	New Comment	15.21	7.83	14.0			
Test	Old Code	97.61	86.92	64.0	36.85	63.73	12.0
	New Code	97.44	87.94	63.0			
	Old Comment	15.37	7.52	14.0	3.97	3.11	3.0
	New Comment	15.29	7.77	14.0			

To better represent the code changes during the synchronization, Liu et al. [73, 72] adopted a python *diff* tool, namely SequenceMatcher [4] to align each old code sequence and its associated new code sequence. The SequenceMatcher is able to identify four edit actions, including "equal", "replace", "insert", and "delete" at the sub-sequence-level between two sequences. For example, Listing 5.1 presents a code-comment co-change instance. After breaking the old code, and new code of this sample into sub-tokens, SequenceMatcher can align old and new code sequences of Listing 5.1, and present results as shown in Figure 5.1. Subsequently, To convert the alignment from sub-sequence-level to element-level, Liu et al. conducted a series of operations: (1) for sub-sequence matches

[4]https://docs.python.org/3/library/difflib.html

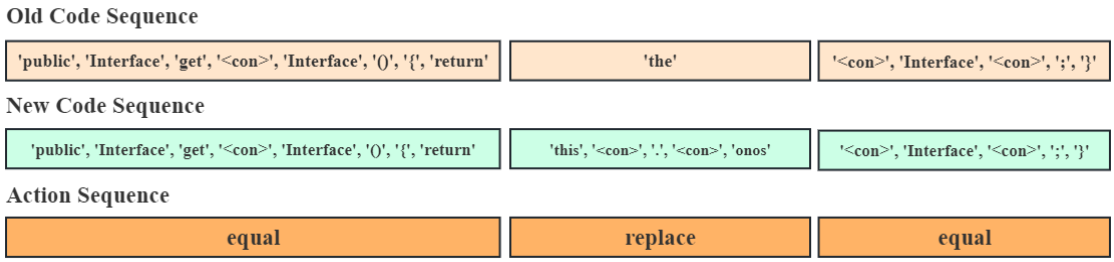


Figure 5.1: The code change sequence of the code-comment co-change instance on sub-sequence-level

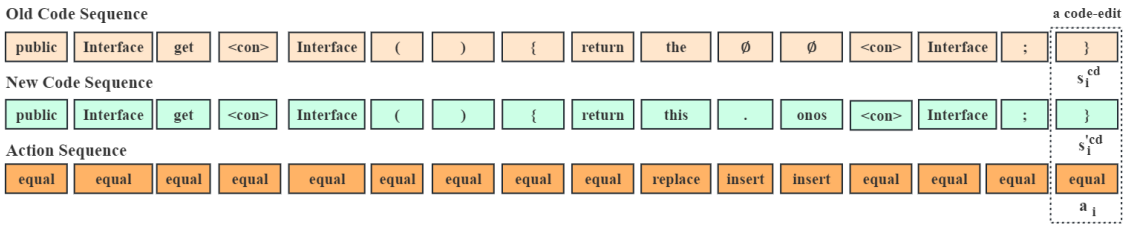


Figure 5.2: The code change sequence of the code-comment co-change instance on element-level

marked as “equal”, they keep them intact, (2) for those marked as “insert” and “delete”, they directly fill up empty tokens “∅” for the shorter ones to align them, (3) for those marked as “replace”, since the lengths of two sub-sequences in each matched pair may not be equal, they align them from the head or tail according to the similarity of their first and last tokens. In detail, if the similarity between the first tokens of the two sub-sequences is larger than that between the last tokens, they align the two sub-sequences from their head and add empty tokens “∅” to fill up the short one, otherwise, they align the two sub-sequences from the tail and add empty tokens “∅” accordingly. The similarity is computed by *quick_ratio* which is a built-in function of SequenceMatcher. For example, towards the sole “replace” edit action between the sub-sequence (i.e., [“the”]) of the old code and the sub-sequence (i.e., [“this”, “<con>”, “.”, “<con>”, “onos”]) of the new code shown in the Figure 5.1, sub-token “the” is closer to sub-token “this” than to sub-token “onos” according to the similarity score computed by *quick_ratio*. Thus sub-sequence [“the”] aligns sub-sequence [“this”, “<con>”, “.”, “<con>”, “onos”] from head, and empty tokens “∅” fills up the shorter sub-sequence [“the”] in the old code. Figure 5.2 is the element-level aligned code change sequence processed by the above procedure based on Listing 5.1. Each element, i.e., code-edit, in the code change sequences, is a triple <s_i^{cd}, $s_i^{'cd}$, a_i>, where s_i^{cd} is the ith sub-token in the old code, $s_i^{'cd}$ is the ith sub-token in the new code, and a_i denotes the edit action that converts s_i^{cd} to $s_i^{'cd}$, which can be “equal”, “replace”, “insert”, and “delete” as we mentioned above. If a_i is “insert” (“delete”), s_i^{cd} ($s_i^{'cd}$) will be the empty token ∅.

5.2.2 Evaluation Metrics for Code-Comment Synchronization

Following Liu and Lin et al.' works [66, 73], we adopt **Accuracy**, **Recall@5**, **AED**, **RED**, and **BLEU-4** to evaluate the performance of the two code-comment synchronization approaches.

- **Accuracy** measures the capability of code-comment synchronization approaches on *correct synchronization*s. The term *correct synchronization*s means the synchronized comments are identical with the reference comments, which also means CCI samples are correctly synchronized. Specifically, Accuracy is the percentage of CCI samples where *correct synchronization*s can be generated on the first attempt. More formally, it can be defined as below:

$$Accuracy = \frac{N_{correct}}{N} \tag{5.2}$$

  where $N_{correct}$ is the number of *correct synchronization*s, and N is the number of testing samples. Thus, the higher the Accuracy, the better the performance of code-comment synchronizers. When judging the identicalness between synchronized comments and reference comments, Liu et al. [73] ignore the punctuations at the end of the comments, while Lin et al. [66] ignore the case and all punctuations in comments. Since both punctuation and case will not affect developers' understanding of comments too much, we uniformly adopt the code implementation of Lin et al. to calculate the Accuracy of synchronizers in our study.

- **Recall@5** is similar to Accuracy but allows an approach to conduct five attempts for each CCI sample. If any one of the synchronized comments is a *correct synchronization*, it considers the approach can successfully handle this sample [73]. Likewise, the higher, the better. This metric is also useful and practical because

providing correctly synchronized comments in limited (e.g., five) candidates also saves the efforts of developers and offers alternative options.

- **AED** is the average edits that developers need to perform in order to change the synchronized comments to the reference comments after being handled by code-comment synchronizers, which indicates the distance between the synchronized comments to the reference comments. Therefore, the smaller the AED, the better the performance of code-comment synchronizers. More formally, AED can be defined as below:

$$AED = \frac{1}{N}\sum_{k=1}^{N} ED(\hat{y}^{(k)}, y^{(k)}) \tag{5.3}$$

where ED is the edit distance (i.e., word-level Levenshtein distance [85]) between $\hat{y}^{(k)}$and y^k, $\hat{y}^{(k)}$ represents the kth synchronized comment, y^k denotes the kth reference comment, and N is the number of the testing samples. Since Lin et al. [66] did not publish the source code of calculating AED, we adopt the code implementation of Liu et al. [73] except that we ignore the case.

- **RED** is similar to AED but presents the average of relative edit distance. In detail, it measures the ratio of AED between synchronized comments ($\hat{y}$s) and reference comments (ys) to the AED between old comments (xs) and reference comments (ys), which indicates to what extent an approach can release the burden of developers from a manual update: the smaller, the better.

$$RED = \frac{1/N\sum_{k=1}^{N} ED(\hat{y}^{(k)}, y^{(k)})}{1/N\sum_{k=1}^{N} ED(x^{(k)}, y^{(k)})} \tag{5.4}$$

- **BLEU-4** is a universally applied precision-oriented evaluation metric in code

summarization [17, 44, 134] and code-comment synchronization [73] fields, which measures the quality of automatically generated sentences by computing the n-gram precision of a generated sentence to the reference sentence [91]: the higher, the better. More formally, it can be defined as follows:

$$BLEU-N = BP * exp(\sum_{n=1}^{N} w_n log\ p_n) \tag{5.5}$$

$$BP = \begin{cases} 1, & if \quad candi > ref \\ e^{1-\frac{ref}{candi}}, & if \quad candi \leq ref \end{cases} \tag{5.6}$$

where p_n represents the ratio of sub-sequences of the length of n in candidate synchronized comments that also appear in the reference comments, BP represents the brevity penalty, $candi$ and ref represent the length of the candidate synchronized comments and reference comments, $w_n = \frac{1}{N}$, and $N = 4$ in this work is to reflect the weighted sum from 1 to 4.

To sum up, since Accuracy and Recall@5 only count the proportion of samples that are absolutely correctly synchronized, their evaluation results reflect the correct synchronization capability of approaches. However, AED, RED, and BLEU-4 compute the average similarity of the overall samples between synchronized comments and their corresponding reference comments. Thus, AED, RED, and BLEU-4 evaluate the general synchronization capability of approaches.

5.2.3 Experimental Design

In this section, first of all, we assess the performance of CUP and HebCUP on the same validation set and unified evaluation criteria. In detail, both CUP and HebCUP are

constructed based on the source code published in their respective papers [73, 66]. CUP is trained from scratch on the training set and then evaluated on the validation set, while HebCUP is directly evaluated on the validation set due to its training-free characteristic. With respect to the initial training parameters, hyper-parameters, and training strategies of CUP, we fixed them according to the setting in its original paper [73], such that the re-trained CUP can reproduce its original performance. Afterward, the outputs of CUP and HebCUP on validation set are uniformly evaluated via the metrics we illustrated in Section 5.2.2.

After obtaining and evaluating the outputs of CUP and HebCUP, we further deeply analyze the distributions of samples that are correctly synchronized within the first attempt (i.e., Accuracy) and five attempts (i.e., Recall@5) to explore the different advantages of CUP and HebCUP on *correct synchronization*. In addition, we also dig deeper into the distribution of samples in terms of edit distance (i.e., concerning the AED and RED) to explore the heterogeneous characteristics of HebCUP and CUP on general synchronization capability. Finally, for samples whose edit distances are reduced after the synchronization, we study them further to explore whether there are potential regularities in the sample preferences of approaches.

Table 5.2: The code-comment synchronization performance of HebCUP and CUP on the validation set

Approach	**Accuracy** (%)	**Recall@5** (%)	**AED**	**RED**	**BLEU-4** (%)
HebCUP	**25.60 [2260/8827]**	25.99	3.68	1.009	70.94
CUP	21.75 [1920/8827]	**30.50**	**3.45**	**0.944**	**72.09**

$^{\Phi}$ The number of *correct synchronization*s are presented in square brackets, where the number of total validation samples is 8827.

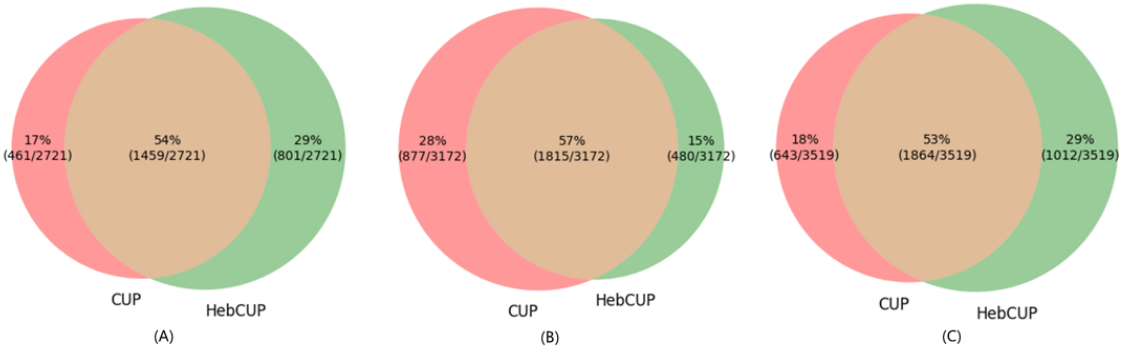


Figure 5.3: (A) The Venn graph for the distributions of samples with correctly synchronized comments after being handled by CUP and HebCUP within the first attempt, (B) The Venn graph for the distributions of samples with correctly synchronized comments after being handled by CUP and HebCUP within five attempts, (C) The Venn graph for distributions of samples with $ED_diff < 0$ after being handled by CUP and HebCUP

5.2.4 Results and Discussion

Table 5.2 presents the performance of CUP and HebCUP on the validation set under the unified evaluation metrics.

(1) As can be seen, HebCUP achieves a score of 25.60% in terms of Accuracy, 25.99% in terms of Recall@5, 3.68 in terms of AED, 1.009 in terms of RED, and 70.94 in terms of BLEU-4. CUP achieves a score of 21.75%, 30.50%, 3.45, 0.944, and 72.09 in terms of the five evaluation metrics, respectively. More specifically, CUP outperforms HebCUP by 17.35% in terms of Recall@5, 6.25% in terms of AED and RED, and 1.62% in terms of BLEU-4. On the contrary, HebCUP performs better than CUP in terms of Accuracy by 17.70%.

(2) It is noticeable that the Accuracy and Recall@5 on the validation set of CUP in our experiment are higher than those recorded in the paper of Lin et al. [66]. This is because Lin et al. did not uniformly apply the implementations of their evaluation metrics to CUP, and their implementations of the evaluation metrics for HebCUP are looser than those in

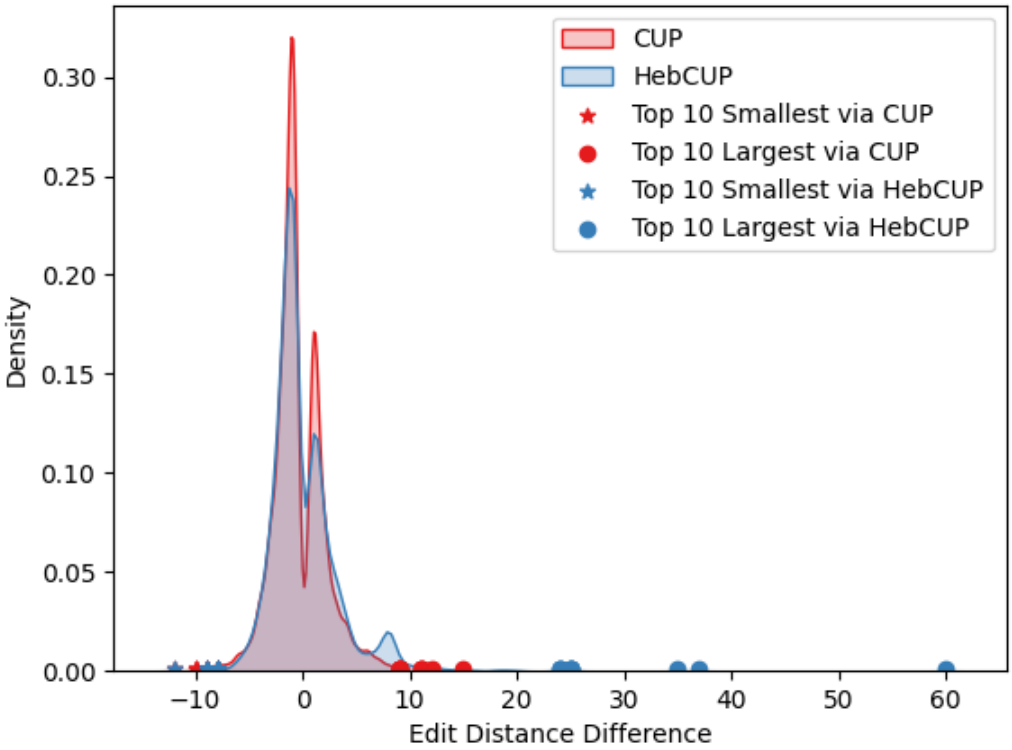


Figure 5.4: The density plot for the sample distributions of $ED_diff \neq 0$ after being handled by CUP and HebCUP

the CUP paper (details refer to Section 5.2.2). Therefore, owing to such kind of unfair comparison, the performance of CUP in terms of Accuracy, Recall@5, AED, and RED are lower than those of HebCUP in the paper of Lin et al.[66]. However, as shown in Table 5.2, neither of the two approaches can dominate on all evaluation metrics.

To dig deeper into the performance of CUP and HebCUP, we plot three Venn graphs to describe the sample distributions related to the metrics of Accuracy, Recall@5, AED, and RED in Figure 5.3 and list our findings as follows:

(1) According to Figure 5.3(A), in a total of 2721 samples that can be correctly synchronized by CUP or HebCUP, CUP can correctly synchronize comments for 461 (17%) samples that HebCUP cannot handle. On the contrary, the latter can correctly synchronize comments for 801 (29%) samples that the former cannot handle. Besides, 1459 (54%) samples can be correctly handled by both CUP and HebCUP.

(2) Nevertheless, if we expand the evaluation to five candidate synchronized comments, Figure 5.3(B) illustrates that CUP can correctly synchronize more distinctive samples (28%) than HebCUP (15%). This is due to the usage of beam search in the neural semantic model of CUP, which helps it infer a limited number of best predictions by approximately maximizing the conditional probability [27]. Whereas, the candidates of HebCUP are generated by permutation and combination within the replacement pairs, leading to a lack of diversity in most candidates. (3) Subsequently, we define an equation 5.7 to describe edit distance difference (ED_diff) before and after the code-comment synchronization.

$$ED_diff = ED(\hat{y}, y) - ED(x, y) \tag{5.7}$$

where $ED(\hat{y}, y)$ denotes the edit distance from the synchronized comment ($\hat{y}$) to the reference comment (y) while $ED(x, y)$ denotes the edit distance from the old comment (x) to the reference comment (y). For example, if $ED_diff < 0$, it represents the edit distance

of a sample is reduced, that is to say, comparing with the old comment, the synchronized comment approaches closer to the reference comment, thus the code-comment synchronizer shows a positive effect; if $ED_diff > 0$, it illustrates the edit distance increases and the code-comment synchronizer has a negative effect on the sample; and if $ED_diff = 0$, it tells us the synchronizer has no effect. with the aid of ED_diff, Figure 5.3(C) demonstrates that the edit distances of 29% distinctive samples are reduced after being handled by HebCUP, while the edit distances of 18% distinctive samples are reduced by CUP.

It is noteworthy that CUP outperforms HebCUP in terms of AED and RED but contributes fewer edit distance-reduced samples. In order to clearly explain this phenomenon, we plot samples of $ED_diff \neq 0$ in Figure 5.4 to more clearly present the distributions of samples whose edit distances are changed after being handled by CUP and HebCUP, where the red (blue) points represent the samples with top 10 largest ED_diff values processed by CUP (HebCUP), and the red (blue) stars represent the samples with top 10 smallest ED_diff values processed by CUP (HebCUP). [5] We find that although HebCUP contributes more edit distance-reduced samples, it produces more edit distance-increased samples, and the edit distance of these samples increases even more. [6] Therefore, CUP outperforms HebCUP in terms of AED and RED. The above analysis inspires us that AED and RED indeed can describe the model performance concerning the edit distance over the entire samples on the sub-token-level, but they cannot measure the edit distance from the perspective of individual quantity (i.e., sample-level), such as counting the proportion of samples whose edit distances are reduced after being handled

[5]The proportions of $ED_diff \neq 0$ for CUP and HebCUP are 43.94% and 52.71%, respectively, indicating the fact that both CUP and HebCUP indeed have a number of samples that cannot implement any synchronization for them. Nevertheless, this phenomenon does not affect the improvement of our proposal based on CUP and HebCUP.

[6]HebCUP: 1776 samples with $ED_diff > 0$, where the edit distance increases 3.106 on average; CUP: 1372 samples with $ED_diff > 0$, where the edit distance increases 2.103 on average.

by synchronizers. In particular, approaches with better AED and RED may also have fewer samples whose edit distances are reduced after the synchronization as shown in the above comparison between CUP and HebCUP, which is also the issue that the previous studies ignore. Therefore, in Section 5.5, we propose a new evaluation metric as complementation for AED and RED.

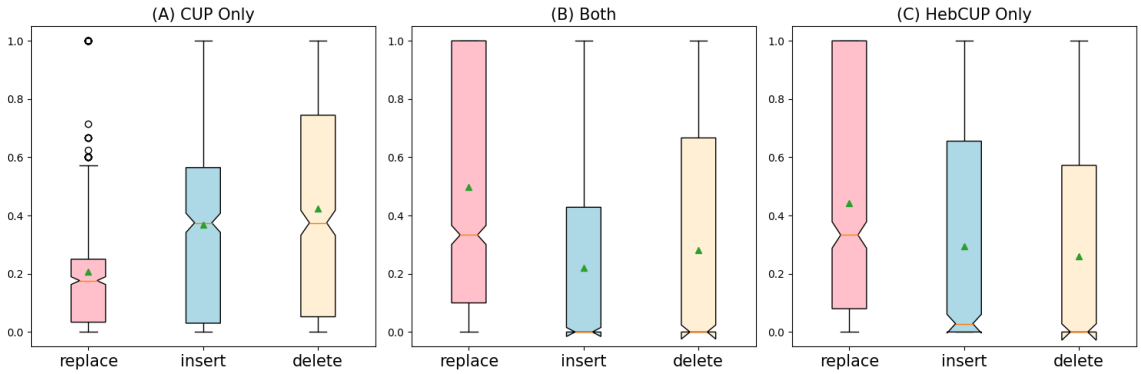


Figure 5.5: The preferences of HebCUP and CUP on samples with different percentages of "replace", "insert", and "delete"

Afterward, we further conduct a statistical analysis on the samples whose edit distances are reduced after the synchronization in Figure 5.5, where all these samples are partitioned into three kinds, i.e., (A) edit distances can be reduced by HebCUP only, (B) edit distances can be reduced by both approaches, and (C) edit distances can be reduced by CUP only. For each part of the samples, we show the percentages of their edit actions (i.e., "replace", "insert", and "delete") in code changes via box plots, where the green triangles represent the mean values and the orange lines represent the median values. As can be seen, HebCUP extremely prefers to handle samples with a high percentage of "replace" action, and around half of those samples do not have any edit actions of "insert" and "delete" according to Figure 5.5(B) and (C). Because the mechanism of HebCUP is exploiting replacement pairs to conduct token-level updates on old comments. On the other hand, according to Figures

5.5(A), we find that samples with edit actions of "insert" and "delete" account for higher percentage compared with those with "replace" action. However, in Figure 5.5(B), samples with "replace" action account for the most proportion. Edit distances of samples in both figures can be reduced by CUP, but present quite different distributions over edit actions, showing that CUP is not overly dependent on the edit actions, like HebCUP, because it is driven by deep learning.

Conclusion

(1) HebCUP outperforms CUP in terms of Accuracy, while CUP produces better results in Recall@5, AED, RED, and BLEU-4.

(2) Both HebCUP and CUP have some distinctive samples that the other one cannot handle but can be better solved by themselves.

(3) HebCUP prefers to handle samples having more edit actions of "replace" and fewer actions of "insert" and "delete" in their code changes whereas CUP is not overly dependent on the edit actions like HebCUP during the code-comment synchronization.

Table 5.3: Category distribution of code-comment inconsistent samples in the training and validation sets

Dataset	**Category**	**Count**	**Proportion**
Training Set	Heuristic-Prone	19202	23.83%
	Non-Heuristic-Prone	61389	76.17%
Validation Set	Heuristic-Prone	2260	25.60%
	Non-Heuristic-Prone	6567	74.39%

5.3 Methodology

This section interprets the methodologies of CBS, including the our motivation, approach overview, and a series of technical details.

5.3.1 Motivation

According to the statistics and analysis in Section 5.2.4, we have verified the heterogeneous characteristics of CUP and HebCUP, including their superiority on different metrics and preferences on different kinds of samples, which are originated from the different properties and mechanisms of the two approaches. Therefore, it has a great possibility that combining these two kinds of approaches can be beneficial and reach a better result in the code-comment synchronization task. Due to this, we propose a Classifying Before Synchronizing (CBS) approach, which predicts the proneness of CCI samples on approaches (i.e., HebCUP and CUP) first, then assigns samples to the most suitable approach to conduct the comment synchronization with code changes. Theoretically, the code-comment synchronization results of the CBS approach heavily depend on the performance of category prediction for the proneness of CCI samples. If the prediction for the proneness is wrong, CCI samples will be assigned to the wrong approach, and the final synchronization result will be awful. Hence, our methodology mainly focuses on the following problems: (1) How to represent the internal relation patterns between CCI samples and their proneness to facilitate the category prediction. (2) The class imbalance problem during category prediction is also an imperative issue that needs to be solved.

5.3.2 Approach Overview

Figure 5.6 shows the overall framework of our proposed CBS approach, which consists of two stages with multiple steps. For the offline training stage, we first extract the five predefined features from the training and validation sets, and label the samples as heuristic-prone or non-heuristic-prone in Step 1; In Step 2, exploiting the features we extract from the previous step, we train a Multi-Subsets Ensemble Learning (MSEL) classification model for category prediction of CCI samples. Besides, the MSEL classification model also alleviates the class imbalance problem during the category prediction due to its ensemble technique; Finally, we exploit the training and validation sets to train the CUP as the code-comment synchronizer for non-heuristic-prone samples in Step 3. Until now, the CBS has prepared for online testing as shown in the left-hand side of Figure 5.3.2. For the online testing stage, we firstly extract the same features from the testing sample in Step 4; Then, we employ the trained MSEL classification model (as mentioned in Step 2) to predict the category of the testing sample in Step 5; Finally, in Step 6, if the predicted category is heuristic-prone, we adopt HebCUP to synchronize the comment of the sample, otherwise, we assign CUP to handle the sample.

In order to illustrate our approach more clearly, we present a running example here in Figure 5.7 to show how CBS handles CCI samples in practice via combining CUP and HebCUP, and list two CCI samples for illustration, the one is from Apache Wicket [7] which is a sample can be correctly synchronized by CUP, another one is from the Google Nomulus [8], which can be correctly synchronized by HebCUP. For code changes, we highlight the statements that are deleted in red, and for those that are newly added, we highlight them in green. Besides, we also circle the specific changed regions in both code and comments

[7]https://github.com/apache/wicket/pull/283/commits/8dcf2e34927e0c164235f5bea79c7026d22192dc
[8]https://github.com/google/nomulus/commit/cf507dad6d7bfc9e30eb520da0c08a75d054b2bd

for the convenience of illustration. As can be seen, regardless of any CCI samples come, their five features will be extracted first, then the MSEL classification model will identify the categories of samples (i.e., Phase 1: Category Prediction), then allocate them to the corresponding code-comment synchronizers to generate new versions of comments and replace their old obsolete comments (i.e., Phase 2: Code-Comment Synchronization).

More specifically, Sample 1 is to fix the "createSessionFolder" value as "true" inside the method, thus the content of "and createSessionFolder is true" which is one of the conditions to create a new folder in the comment should be removed. However, Sample 1 has two parts of changes in the code, the one is the deletion of ",", "boolean", and "createSessionFolder" in Line 5 and the other one is a replacement of "createSessionFolder" with "true" in Line 8-9. The two kinds of operations for "createSessionFolder" make HebCUP hardly tell what should be done for "createSessionFolder" in the old comment. More importantly, the tokens that need to be deleted in the old comment include contents that are not shown in the code changes, such as "and" and "is", thus it is impossible to design heuristic rules. On the contrary, CUP can successfully handle this sample for its learning and comprehension of the code and comment semantics. In this case, the MSEL classification model correctly predicts the category (i.e., non-heuristic-prone) of Sample 1 and allocates it to CUP, thereby we obtain the *correct synchronization* for Sample 1, and list the result in the left-bottom corner of Figure 5.7. For Sample 2, it is going to change the path of the file in code, i.e., from "javatests/%s/testdata/%s" to "src/test/resources/%s/%s" in Line 8-9, and HebCUP is able to extract the replacement pairs between *key*: "javatests" and *value*: "src/test/resources", as well as *key*: "testdata" and *value*: "∅". As such, HebCUP exactly replaces the "javatests" with "src/test/resources" on the old comment, and achieves a *correct synchronization*. Nevertheless, CUP fails to conduct the *correct synchronization*, and its

generated result is “Returns the "real" location of the file loaded by the other commands, starting from test/test.”. Obviously, it repeatedly generates the word “test”, which is a very high-frequency word, ranks 156 out of 44577 in the whole vocabulary, and neglects the correct words, such as “src” and “resources”. In this case, the MSEL classification model correctly predicts the category (i.e., heuristic-prone) of Sample 2 and allocates it to HebCUP, thereby we obtain the *correct synchronization* for it, and list the result in the right-bottom corner of Figure 5.7. Via the procedure we mentioned above, CBS can effectively exploit the advantages of both CUP and HebCUP, thereby continue boosting the performance of code-comment synchronization. We illustrate each part of CBS in detail in the following sections.

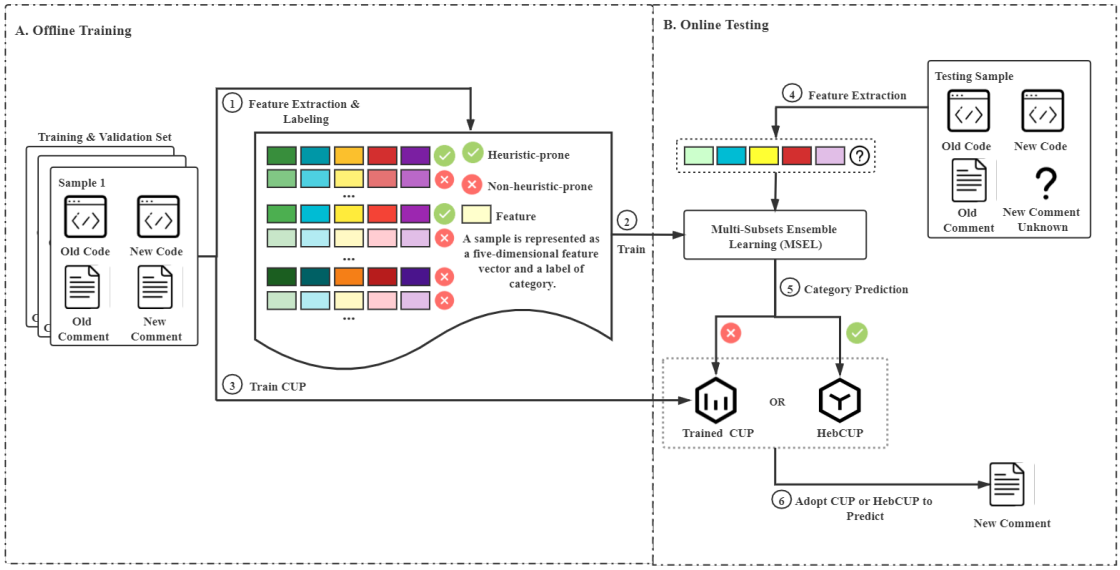


Figure 5.6: The overview of the CBS approach

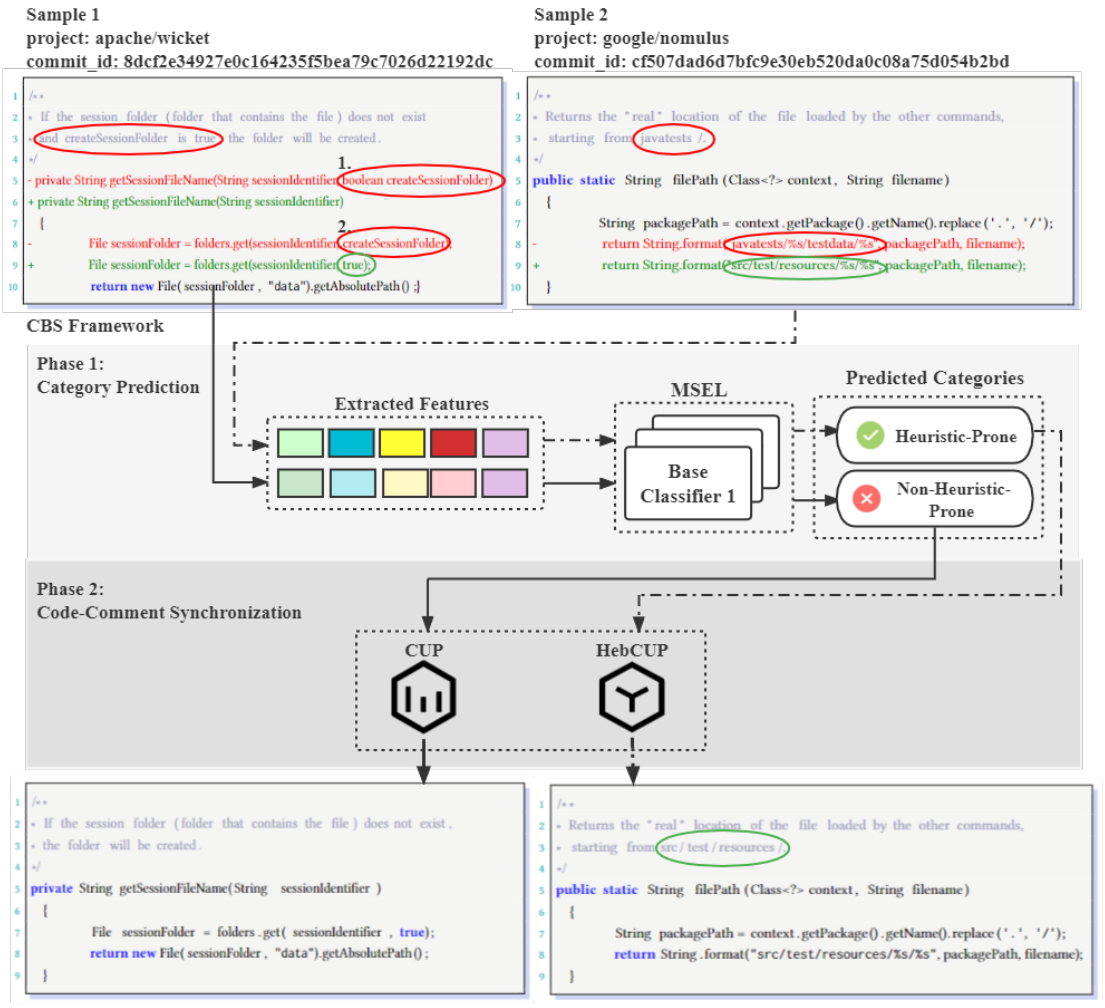


Figure 5.7: A running example to show how CBS works

5.3.3 Category Definition and Partition Criteria

In this chapter, we categorize CCI samples as heuristic or non-heuristic based on the synchronization results of HebCUP. In other words, for those samples that can be correctly synchronized by the HebCUP, we label them as heuristic-prone; otherwise, we label them as non-heuristic-prone. We adopt the synchronization results of HebCUP as the labeling criterion for a variety of reasons: (1) The labeling results of the deep learning-based approach, i.e., CUP, will change more or less for its randomness in training. The category prediction is hard to conduct with nonstationary labels. (2) Code-comment synchronization guided by hand-crafted rules is explainable, thus it is relatively easier to represent the internal relation patterns between CCI samples and their proneness. According to the above definition and partition criterion, we label the categories for CCI samples on the whole training and validation set. Table 5.3 demonstrates the category distribution of CCI samples on the training set and validation set, respectively, where 23.83% of them are heuristic-prone and 76.17% of them are non-heuristic-prone in the training set, while 25.57% of them are heuristic-prone and 74.43% of them are non-heuristic-prone in the validation set.

5.3.4 The Proposed Features

Based on our statistical analysis in Section 5.2.4, we further analyze the training samples in each category and propose five features to facilitate the representation of internal relation patterns between CCI samples and their model proneness, thereby assisting category prediction. The details of the five features are introduced as follows.

- **ReplaceRate:** Ignoring the code edits whose s_i^{cd} and $s_i^{'cd}$ are both meaningless symbols, i.e., ".", "<*con*>" and "$\varnothing$", it is the ratio of code edits with the "replace" action

among those with non-equal actions in a code change sequence:

$$ReplaceRate = \frac{n_{replace}}{n_{replace} + n_{insert} + n_{delete}} \tag{5.8}$$

where $n_{replace}$, n_{insert}, and n_{delete} denote the number of code edits with the action of "replace", "insert", and "delete" in a code change sequence, respectively. The intuition is that the code change sequence (as shown in Figure 5.2) that contains more code edits with the "replace" action in proportion is more likely to be the code-indicative changes, and tends to be easily and correctly synchronized by HebCUP.

- **MatchedLevelsNum:** This feature counts the number of matching levels of HebCUP that are satisfied by a CCI sample. The HebCUP totally designed three matching levels, i.e., tokens, sub-tokens, and sub-tokens without <*con*>. Therefore, the range of this feature is 0-3. The fact behind it is that if none of the matching levels is reached, HebCUP cannot conduct the substitution with its prepared replacement pairs, thus, the old comment will not be synchronized at all. On the contrary, if there are more levels of matchings found, HebCUP will have more chances to conduct the *correct synchronization* for old comments. More formally, we define the MatchedLevelsNum as below:

$$MatchedLevelsNum = isMatch_{(1)} + isMatch_{(2)} + isMatch_{(3)} \tag{5.9}$$

where $isMatch_{(i)}$ denotes whether the matching algorithm designed by HebCUP at the level i is satisfied. If it is satisfied, return 1; otherwise, return 0.

- **NonLetterCount:** Ignoring the meaningless symbols in code, i.e., ".", "<*con*>" and

"∅", it counts the number of non-letter sub-tokens [9] in the union set of respective distinctive sub-token sets of the old and new code sequence. The intuition of this feature is that the more non-letter sub-tokens in the changes between old and new code, the more difficult for HebCUP to sense the correct correspondences between code changes and comment changes. More formally, the NonLetterCount can be defined as below:

$$NonLetterCount = Count_{NonLetterToken}((S^{'cd} - S^{cd}) \cup (S^{cd} - S^{'cd})) \tag{5.10}$$

where $S^{'cd}$ and S^{cd} denote the sets of sub-tokens in a new code sequence and an old code sequence, respectively; $Count_{NonLetterToken}$ denotes the function of counting for non-letter sub-tokens in a given set.

- **LongestChangedSeq:** It is the length of the longest continuous changed sub-sequence in the code change sequence after ignoring the code edits whose s_i^{cd} and $s_i^{'cd}$ are both meaningless symbols, i.e., ".", "<*con*>" and "∅". The motivation for designing this feature is that HebCUP has difficulty in finding the relationship between code changes and comment changes when the length of the continuously changed code sub-sequence is long [66].

- **TotalChangedNum:** Ignoring the code edits whose s_i^{cd} and $s_i^{'cd}$ are both meaningless symbols, i.e., ".", "<*con*>" and "∅", it is the total number of code edits with non-equal actions in a code change sequence:

$$TotalChangedNum = n_{replace} + n_{insert} + n_{delete} \tag{5.11}$$

[9] A non-letter sub-token refers to the sub-token with characters apart from a-z and A-Z, such as "link1" and "&&"

The principle behind this feature is that the more changed sub-tokens exist in the code change sequence, the more difficult for HebCUP to correctly replace the sub-tokens in old comments when synchronizing [66].

To interpret our proposed features more clearly, we illustrate the CCI sample shown in Listing 5.1 as an example and list each feature value of the sample in Table 5.4. As can be seen, the ReplaceRate value of the CCI sample is 0.5, because it contains one code edit with the "replace" action among a total of two code edits with non-equal actions after ignoring the code edits whose s_i^{cd} and $s_i^{'cd}$ are both meaningless symbols. The MatchedLevelsNum value is 3, because HebCUP can find all matching levels of tokens, sub-tokens, and sub-tokens without <*con*> between code changes and the old comment according to its program running result. The NonLetterCount value is 0 because there is no non-letter sub-token in the code change sequence after ignoring the meaningless symbols. Both the LongestChangedSeq and TotalChangedNum values are 2 since the longest continuous changed sub-sequence and the total changed code edits are both {<"the","this","replace">, <"∅", "onos", "insert">} after ignoring the meaningless symbols.

Table 5.4: The feature values of the CCI sample shown in Listing 5.1

ReplaceRate	**MatchedLevelsNum**	**NonLetterCount**	**LongestChangedSeq**	**TotalChangedNum**
0.5	3	0	2	2

Subsequently, we perform a series of statistical tests on the training set to further prove and quantify the correlation between those features and the class label (i.e., either heuristic-prone or non-heuristic-prone). Since the features we proposed are continuous variables and the class label is a binary variable, we employ the Point-Biserial Correlation Coefficient (PBCC), a measurement suitable for evaluating the correlation between a

binary variable and a continuous variable [63], to measure the degree of correlation between each feature and the class label. The larger the absolute value of the correlation coefficient, the larger the degree of correlation, and the sign of the correlation coefficient represents the positive or negative correlation. According to Table 5.5, we find that ReplaceRate is the most relevant feature to discriminate heuristic-prone and non-heuristic-prone samples, while TotalChangedNum is the least relevant feature, but still has a relatively great correlation with the class label. In addition, since the distribution of features in each category is unknown, we conduct a series of Mann-Whitney U tests [81] to evaluate whether the correlation between features and the class label is statistically significant. In addition, we also utilize the Benjamini-Hochberg (BH) procedure to adjust *p-value*s because we perform multiple independent tests, adjusting *p-value*s with BH helps us to control for the fact that sometimes small *p-value*s (less than 0.05) happen by chance, which could lead us to incorrectly reject the true null hypotheses [26]. As can be seen in Table 5.5, the last four features statistically significantly correlate with the label at the significance threshold of 0.05, 0.01, 0.005, and 0.001, and the first feature statistically significantly correlates with the label at the significance threshold of 0.05, 0.01, and 0.005. Figure 5.8 presents the training sample distribution of each category by violin plots among each proposed feature, where the white point represents the median value, and the black box indicates the interquartile range. Intuitively, heuristic-prone samples tend to have higher ReplaceRate and MatchedLevelsNum values, while non-heuristic-prone samples generally have higher NonLetterNum, LongestChangedSeq, and TotalChangedNum values. The distribution of samples indicates the intuitions behind our proposed features are correct.

Table 5.5: The description of our proposed five features

Features	**Description**	**PBCC (p-value)**
ReplaceRate	Ratio of code edits with the "replace" action among those with non-equal actions in a code change sequence.	0.3641 (2.51e-3)
MatchedLevelsNum	The number of matching levels of HebCUP that are satisfied by a CCI sample.	0.3510 (7.47e-5)
NonLetterCount	The number of non-letter sub-tokens in the union set of respective distinctive sub-token sets of the old and new code sequences.	-0.3150 (5.61e-8)
LongestChangedSeq	Length of the longest continuous changing sub-sequences between old and new code.	-0.2758 (1.14e-6)
TotalChangedNum	Total number of code edits with non-equal actions in a code change sequence.	-0.2617 (1.29e-5)

$^{\Phi}$ PBCC is the abbreviation of Point-Biserial Correlation Coefficient

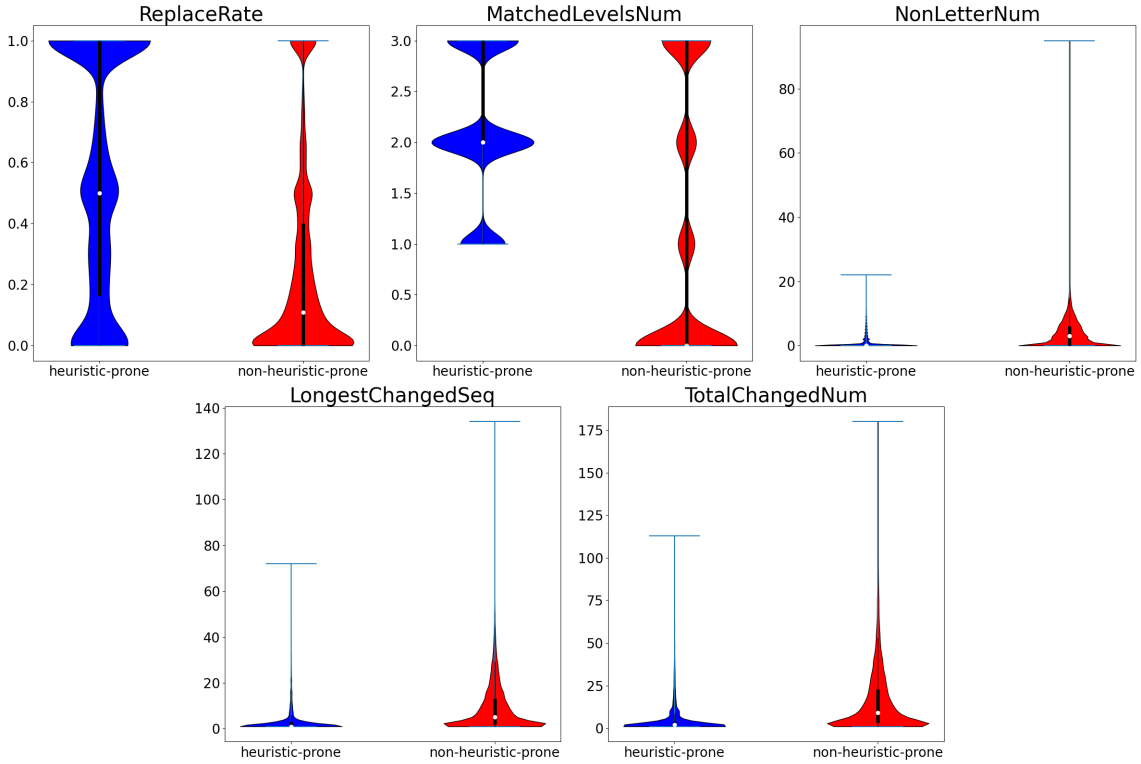


Figure 5.8: The distributions of heuristic-prone and non-heuristic-prone samples among each proposed feature

5.3.5 Multi-Subsets Ensemble Learning

After labeling the samples and extracting the five features, the ith CCI sample for category prediction can be represented as M_i =($\boldsymbol{x}_i$,y_i), where $\boldsymbol{x}_i$=(x_i^1, x_i^2 , ... , x_i^5) is a 5-dimensional feature vector, and y_i is its category. Then, the whole dataset of CCI samples can be represented as:

$$S = \{M_1, M_2, ..., M_n\} \tag{5.12}$$

where n is the number of samples in S. Since the dataset S contains more non-heuristic-prone samples (i.e., the majority class samples) than heuristic-prone ones (i.e., the minority class samples) as stated in the statistics of Section 5.3.3. The prediction model trained on the imbalanced dataset will focus more on the non-heuristic-prone samples, and are prone to predict the testing samples as non-heuristic-prone. However, accurately classifying the heuristic-prone samples is critical for boosting the performance of our proposed CBS approach, because the comments of more samples can be correctly synchronized by HebCUP. Inspired by Wu et al.'s study [132], we propose a Multi-Subsets Ensemble Learning (MSEL) algorithm to alleviate the class imbalance problem for category prediction. More formally, the MSEL algorithm can be defined as below:

$$\hat{y}_i = F(\boldsymbol{x}_i) \tag{5.13}$$

where the $\hat{y}_i$ denotes the predicted category via the MSEL algorithm for the sample $\boldsymbol{x}_i$, F is the function describing the MSEL algorithm. Algorithm 1 lists the detailed process, and Figure 5.9 shows the overall framework of the MSEL algorithm, which consists of two stages with multiple steps.

The offline training stage includes the two steps, i.e., Balanced Multi-Subsets Construction and Ensemble Learning on Multi-Subsets.

- Balanced Multi-Subsets Construction: According to the ratio of the non-heuristic-prone samples to the heuristic-prone ones, we set the number of constructed subsets to be $K = \lfloor N_{major}/N_{minor} + 0.5 \rfloor$, where N_{major} is the number of the majority class samples (i.e., non-heuristic-prone samples) and N_{minor} is the number of the minority class samples (i.e., heuristic-prone samples) (Line 2). Then, we randomly choose N_{minor} non-heuristic-prone samples without replacement $K-1$ times to obtain $K-1$ chunks (Lines 4-5), and the remaining non-heuristic-prone samples are used as the last chunk (Line 7). Therefore, the K chunks are mutually exclusive, and each chunk, except for the last chunk, [10] contains the same number of non-heuristic-prone samples. Finally, we construct K balanced subsets by merging each chunk with all heuristic-prone samples (Line 9).

- Ensemble Learning on Multi-Subsets: After constructing multiple balanced subsets, we train a classifier h_k on each subset S_k (Line 12). Therefore, we can obtain multiple trained base classification models with heterogeneous decision boundaries.

For the online testing stage, with the input of the five features extracted from the testing sample M_{test}, each base classification model returns its predicted category, which counts as one vote (Lines 15-17). Finally, the MSEL algorithm counts the votes and assigns the category with the most votes to M_{test} (Line 18), thereby classifying for M_{test}.

5.4 Experiments for Category Prediction

This section describes the experiments for category prediction, including the experimental setup, evaluation metrics, and experimental results.

[10] If the last chunk accounts for less than half the size of other chunks, it merges into the second last chunk, thus the last chunk will be larger than other chunks; otherwise, we remain it intact, and the last chunk will be smaller compared with other chunks.

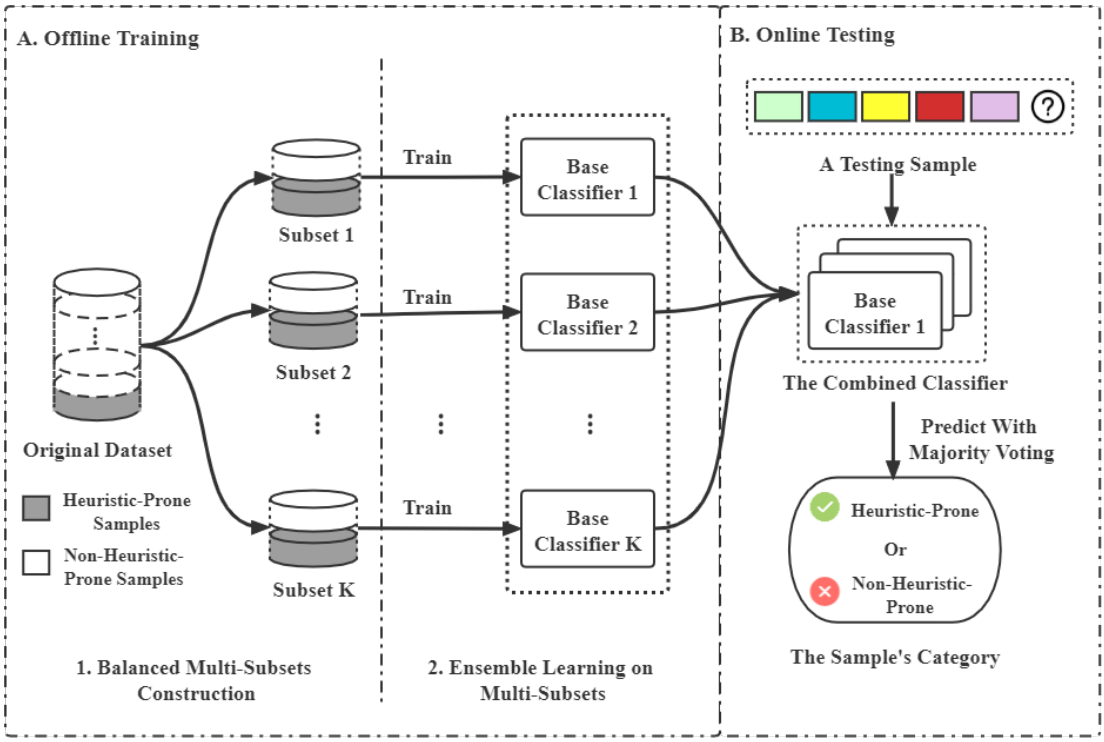


Figure 5.9: The overview of the Multi-Subsets Ensemble Learning (MSEL) algorithm

Algorithm 1 The MSEL algorithm

Input: (1) The training set S_{train} containing N_{minor} heuristic-prone samples and N_{major} non-heuristic-prone samples; (2) A testing sample M_{test}.

Output: The predicted category y_{test} of the testing sample M_{test}.

/*offline training stage*/
Compute the number of subsets $K = \lfloor N_{major}/N_{minor} + 0.5 \rfloor$;
for $k = 1$ to K **do**
  if $k < K$ **then**
    $chunk_k$ = the randomly chosen N_{minor} samples from the non-heuristic-prone samples without replacement;
  else
    $chunk_k$ = the rest of non-heuristic-prone samples;
  end if
  Construct the kth subset S_k by merging $chunk_k$ and all heuristic-prone samples;
end for
for $k = 1$ to K **do**
  Train the base classifier h_k on the subset S_k;
end for
/*online testing stage*/
for $k = 1$ to K **do**
  y_{test}^k = The predicted category of M_{test} by the base classifier h_k;
end for
Majority voting among $\{ y_{test}^1, y_{test}^2, \ldots, y_{test}^K \}$ to get the final prediction result of M_{test};
return The predicted category of M_{test};

5.4.1 Experimental Setup

Since there is no previous study on the CCI sample category definition and classification, here we adopt some widely used machine learning classifiers to plug in the MSEL algorithm as baselines in this field. Specifically, we investigate the performance of the five machine learning classifiers, [11] i.e., Decision Tree [13], Random Forest [12], LightGBM [51], Naive Bayes [102], and Multi-Layer Perceptron (MLP) [33] in this chapter, where their inputs are our proposed features introduced in Section 5.3.4. Our adoption of these classifiers is inspired by a similar task of comment classification, in which they have been proved to be effective [17, 94, 95, 138]. Besides, we also adopt Bi-directional Long Short-Term Memory (Bi-LSTM) [42, 104] and Convolutional Neural Network (CNN) [3] as extra classifiers to extract the semantics of code changes and old comments directly in the category classification for CCI samples. [12]

More specifically, for the construction of the Bi-LSTM classifier, we use four embedding layers to vectorize old code, new code, edit actions, and old comments, respectively. Then, the embedded old code, new code, and edit actions of each sample are concatenated horizontally for alignment, such that we can capture the code change semantics. Next, the Bi-LSTM network accepts the embeddings of each pair of code change and its associated old comment, and inputs them into their respective layers with two concatenated LSTMs (i.e., Bi-LSTM) for each. After horizontally propagating the semantics of code change and old comment from the two directions, respectively, their embeddings are concatenated together and go through a softmax layer to obtain the output labels. Whereas for the construction of the CNN classifier, we replace the Bi-LSTM

[11] We adopt sci-kit learn toolkit [96] to implement the Decision Tree, Random Forest, Naive Bayes, and Multi-Layer Perceptron. We adopt the official library [51] to implement LightGBM.

[12] We adopt TensorFlow (https://www.tensorflow.org/) to implement the Bi-LSTM and CNN classifiers.

structures with 1-dimensional CNN structures in the Bi-LSTM classifier, and keep other structures no change.

Given the labeled training and validation set as described in Section 5.3.3, we merge them together as a mixed dataset, and employ the 10-fold cross-validation on it to evaluate each base classifier embedded in the MSEL algorithm in terms of the metrics described in Section 5.4.2. The 10-fold cross-validation divides the mixed dataset into 10 consecutive folds. Each fold (i.e., 8941 samples) is used once for evaluation while the remaining nine folds (i.e., 80476 samples) are used for training, thereby ensuring each fold is used as both the training and validation data. For the deep learning classifiers in particular, we train the models with 50 epochs and randomly selects 10% of data for in-training-validation in each fold. Besides, we adopt an early stopping strategy that if the loss of in-training-validation stops decreasing for three epochs, the model will stop training.

Since hyper-parameter tuning is essential in machine learning and deep learning experiments, we generate candidate hyper-parameter settings (see Table 5.6 for details) according to a series of relevant hyper-parameter optimization studies [31, 116, 49, 55], and control the capacity of them based on a given budget threshold (i.e., the tuneLength) for evaluation. The budget threshold refers to the number of different values to be evaluated for each hyper-parameter. As suggested by Kuhn [59], we use a budget threshold of 5. For each 10-fold cross-validation, we can only obtain an evaluation result of one classifier with one hyper-parameter combination. As such, to explore the best hyper-parameter combination of each classifier and compare them with each other, we utilize the grid search [11] such that we can conduct an exhaustive search through our manually specified set of the hyper-parameter space [116]. The tuned classifier that obtains the best performance is selected and plugged into the MSEL algorithm of CBS for online testing.

Table 5.6: Hyper-parameter tuning for category prediction

Classifier	Hyper-parameter Name	Hyper-parameter Description	Candidate Values
Naive Bayes	var_smoothing	For calculation stability.	{0, 1e-9, 1e-7, 1e-5, 1e-3}
Random Forest	max_depth	The maximum depth of the tree.	{7, 9, 11, 13, 15}
	n_estimators	The number of trees in the forest.	{50, 100, 150, 200, 250}
Decision Tree	max_depth	The maximum depth of the tree.	{7, 9, 11, 13, 15}
LightGBM	max_depth	The maximum depth of the tree.	{7, 9, 11, 13, 15}
	n_estimators	The number of trees in the forest.	{50, 100, 150, 200, 250}
	learning_rate	The boosting learning rate.	{0.001, 0.005, 0.01,0.05,0.1}
MLP	hidden_layer_sizes	The number of neurons in the hidden layer.	{32, 64, 128, 256, 512}
	learning_rate_init	The initial learning rate used.	{0.001, 0.005, 0.01,0.05,0.1}
Bi-LSTM	units	Dimensionality of the output space.	{32, 64, 128, 256, 512}
	learning_rate	The initial learning rate used.	{0.001, 0.005, 0.01,0.05,0.1}
CNN	kernel_size	The length of the 1D convolution window	{1, 3, 5, 7, 9}
	filters	Dimensionality of the output space.	{32, 64, 128, 256, 512}
	learning_rate	The initial learning rate used.	{0.001, 0.005, 0.01,0.05,0.1}

5.4.2 Evaluation Metrics for Category Prediction

We employ Precision, Recall, and F1-Score to evaluate the effectiveness of the MSEL algorithm with different base classifiers, and Table 5.7 lists the definitions of the metrics. In our experiment, the heuristic-prone samples are regarded as positive, and the non-heuristic-prone samples are regarded as negative. In Table 5.7, TP (True Positive) is the number of heuristic-prone samples that are correctly predicted to be heuristic-prone, FP (False Positive) is the number of non-heuristic-prone samples that are wrongly predicted to be heuristic-prone, FN (False Negative) is the number of heuristic-prone samples that are wrongly predicted to be non-heuristic-prone, and TN (True Negative) is the number of non-heuristic-prone samples that are correctly predicted to be non-heuristic-prone.

Precision is the ratio of the correctly predicted heuristic-prone samples to all predicted heuristic-prone samples.

Recall is the ratio of the correctly predicted heuristic-prone samples to all actual heuristic-prone samples.

F1-Score is the harmonic mean of Precision and Recall.

Table 5.7: The evaluation metrics for category prediction.

	Actual	Predicted	
		heuristic-prone	non-heuristic-prone
	heuristic-prone	TP	FP
	non-heuristic-prone	FN	TN
Precision	TP/(TP+FP)		
Recall	TP/(TP+FN)		
$F1-Score$	2×Precision×Recall/(Precision + Recall)		

Since the proportion of categories in each fold may vary more or less, we aggregate the TP/FP/TN/FN instances across the 10 folds and then compute precision, recall, and F1-Score, rather than computing these three metrics for each fold and then averaging them, thereby obtaining a more accurate evaluation for each classifier.

5.4.3 Experimental Results

Table 5.8 presents the performance of the MSEL algorithm with different base classifiers, where the hyper-parameters of these classifiers are fully tuned before conducting the comparison. The best hyper-parameter combinations are listed in the last column. According to our observation, we have the following findings:

(1) LightGBM performs best in terms of F1-Score. Besides, it also ranks the second in terms of Recall and ranks the third in terms of Precision. This indicates that MSEL

with LightGBM can reach the best balance between increasing the true positive rate and decreasing the false positive rate.

(2) Naive Bayes performs the best in terms of Recall. However, its performance on Precision and F1-Score are the lowest among seven classifiers. It indicates that the MSEL algorithm with the Naive Bayes as the base classification model tends to predict the samples to be heuristic-prone, which can recognize more heuristic-prone samples but with lots of false positives.

(3) CNN obtains the best performance on Precision, and the second best performance on F1-Score. Nevertheless, it does not perform well on Recall comparing with other classifiers, where it ranks the sixth. This indicates that MSEL with CNN has the highest true positive rate but can classify out relatively small number of true heuristic-prone samples.

(4) F1-Score is the harmonic mean of Precision and Recall, which is a more comprehensive metric for the evaluation of category prediction for CCI samples. Therefore, although Naive Bayes performs better in terms of Recall, and CNN performs better in terms of Precision, LightGBM is selected as the base classifier of the MSEL algorithm for its best performance in terms of F1-Score.

Conclusion

LightGBM embedded in the MSEL algorithm performs the best on the category prediction for CCI samples. Thus, LightGBM is selected as the base classifier of the MSEL algorithm in our CBS approach.

Table 5.8: The performance of the MSEL algorithm with different base classifiers across the 10-fold cross-validation.

Classifier	Precision	Recall	F1-Score	Best Hyper-parameter Combination
Naive Bayes	47.69%	**91.07%**	62.60%	{"var_smoothing": 1e-9}
Random Forest	59.36%	88.89%	71.18%	{"max_depth": 13, "n_estimators": 200}
Decision Tree	59.46%	88.68%	71.19%	{"max_depth": 15}
LightGBM	59.69%	90.57%	**71.96%**	{"max_depth": 11, "n_estimators": 250, "learning_rate":0.1}
MLP	56.77%	87.81%	68.96%	{"hidden_layer_sizes": 512, "learning_rate":0.01}
Bi-LSTM	60.20%	85.36%	70.60%	{"units": 128, "learning_rate":0.01}
CNN	**61.00%**	86.88%	71.68%	{"filters": 32, "kernel_size":3, "learning_rate":0.01}

5.5 Experiments for Code-Comment Synchronization

In this section, we illustrate the experiments of code-comment synchronization, which contains the experimental setup, evaluation metrics, and experimental results.

5.5.1 Experimental Setup

After evaluating and selecting the best base classifier, we re-train the MSEL algorithm on the whole mixed dataset. Then the trained MSEL algorithm is applied to the testing set to conduct the category prediction for CCI samples. Finally, with the HebCUP and trained CUP [13], the CBS approach can be constructed as a whole. In this section, we compare the performance of CBS, CUP, and HebCUP on the testing set in terms of Accuracy, Recall@5, AED, RED, BLEU-4, and ESS ratio. As such, we can explore whether the category prediction can further boost the code-comment synchronization performance via the comprehensive assessment. Subsequently, we design a CBS-R approach with random

[13]Here we re-train the CUP with the same hyper-parameters and training strategies as recorded in its original paper [73].

classification for all CCI samples and make a comparison with CBS to verify whether the effect of the category prediction model is significant. Finally, we further devise a CBS-Max approach with completely correct classification for all CCI samples and make a comparison with CBS to report the maximum capability of the CBS approach.

5.5.2 Evaluation Metrics

As we mentioned above, we utilize Accuracy, Recall@5, AED, RED, BLEU-4, and ESS ratio as our evaluation metrics to assess the performance of each model. The first five metrics are utilized by the previous studies as we illustrated in Section 5.2.2. The last metric, ESS ratio, is first proposed in this chapter to evaluate the ratio of samples whose edit distances are reduced after the code-comment synchronization. These samples can be referred to as Effectively Synchronized Samples (ESS) because their synchronized comments are closer to the reference comments, which can be regarded as effective synchronizations. This metric measures the ratio of samples (i.e., code methods) that code-comment synchronizers can relieve the effort of developers, the higher, the better. The difference between ESS Ratio and AED as well as RED is the measuring granularity, where the former is on the sample-level while the latter two are on the sub-token-level. As the complementary metrics of AED and RED, ESS ratio can be formally defined as below, where N denotes the total number of the samples, $N_{ED_diff<0}$ denotes the number of samples whose edit distances are reduced.

$$ESS\ ratio = \frac{N_{ED_diff<0}}{N} \tag{5.14}$$

Considering the existence of CUP in our proposed CBS approach, we run the experiment ten times and report confidence interval ($\mu \pm \theta$) at the confidence level of

95% for each of the metrics, [14] because training for CUP has the randomness characteristic due to its deep learning-based structure, and we also hope to obtain a stable result. More formally, we show the computation of μ and θ as below, where μ is the sample mean, θ is the range around μ that we estimated, $t_{0.025}(N_e - 1)$ is the two-tails t distribution table value with the degrees of freedom of $N_e - 1$ at the confidence level of 95%, [15] s is the sample standard deviation, N_e is the number of experiments, and v_i represents the value of Accuracy, Recall@5, AED, RED, BLEU-4, or ESS ratio in the ith experiment.

$$\mu = \frac{1}{N_e}\sum_{i=1}^{N_e} v_i \tag{5.15}$$

$$\theta = \frac{s}{\sqrt{N_e}} t_{0.025}(N_e - 1) \tag{5.16}$$

$$s = \sqrt{\frac{1}{N_e - 1}\sum_{i=1}^{N_e}(v_i - \mu)} \tag{5.17}$$

5.5.3 Experimental Results

Visualizations: In Table 5.9, we list a detailed description for the performance of CBS. The rows of HebCUP Side and CUP Side demonstrate their respective code-comment synchronization performance on their assigned testing samples (i.e., samples predicted as heuristic-prone are piped into the HebCUP side for code-comment synchronization while samples predicted as non-heuristic-prone are piped into the CUP side). After combining the performance of HebCUP Side and CUP Side, the final performance of CBS is displayed in the last row. Since we run the experiment ten times to eliminate the randomness of CBS and CUP, we report the confidence interval as we mentioned in

[14] The adoption of the confidence level at 95% is a follow of statistical convention [58, 22].

[15] We adopt t distribution to approximate the distribution of the repetitive experimental results of CBS because its population distribution is unknown [10]

Section 5.5.2 for each evaluation metric. However, HebCUP need not be reported in this way due to the heuristic characteristic. Table 5.10 presents the performance of CBS, CUP, and HebCUP in terms of Accuracy, Recall@5, AED, RED, BLEU-4, and ESS ratio on the testing set also via confidence intervals. In order to investigate whether the superiority of our proposed CBS approach towards the two baselines is statistically significant, we conduct the Wilcoxon signed-rank test [130] on the experiment results for each evaluation metric. We adopt Wilcoxon signed-rank test because it is a paired difference test without the assumption that the paired samples are normally distributed. Similarly, our comparisons between each approach and CBS are paired in each experiment and their distributions are also unknown to us. In addition, we also utilize the BenjaminiâHochberg (BH) procedure to adjust *p-value*s for the same reason as we mentioned in Section 5.3.4 [26]. In Table 5.10, the performance value labeled with a star (*) indicates CBS statistically significantly outperforms the baseline in terms of the corresponding metrics. Besides, the row of CBS-R demonstrates the performance of CBS when the samples are randomly assigned, which is a lower limit customized by ourselves. And the row of CBS-Max presents the best performance that CBS can reach when the categories of all samples can be correctly predicted. The rows of -HebCUP, -CUP, -CBS-R under the W/T/L (Win/Tie/Loss) record how many experiments CBS obtains a win, tie, and loss compared to CUP, HebCUP, and CBS-R in terms of each evaluation metric.

Results: We have the following findings from the Table 5.9 and Table 5.10:

(1) According to Table 5.9, 3465 (37.65%) samples are predicted to be heuristic-prone while 5739 (62.35%) samples are predicted to be non-heuristic-prone by the MSEL algorithm among the whole testing set of 9204 samples in total. On the basis of the sample allocation results, HebCUP Side achieves an average score of 59.51% in terms of Accuracy, 60.72% in terms of Recall@5, 2.050 in terms of AED, 0.729 in terms of RED,

81.77 in terms of BLEU-4, and 69.52% in terms of ESS ratio on the 3465 predicted heuristic-prone samples, while CUP Side achieves an average score of 8.83%, 20.54%, 4.474, 1.018, 67.96, and 15.53% in terms of Accuracy, Recall@5, AED, RED, BLEU-4, and ESS ratio on the 5739 predicted non-heuristic-prone samples. Combining above performance of the two models, CBS finally achieves an average score of 27.91% in terms of Accuracy, 35.67% in terms of Recall@5, 3.561 in terms of AED, 0.937 in terms of RED, 73.16 in terms of BLEU-4, and 35.85% in terms ESS ratio.

(2) Similar to the experimental results on the validation set in Section 5.2.4, CUP still outperforms HebCUP in terms of Recall@5, AED, RED, and BLEU-4, whereas HebCUP has its superiority in terms of Accuracy and ESS ratio on the testing set as shown in Table 5.10.

(3) Comparing with the two baselines, CBS performs the best in terms of all evaluation metrics as shown in Table 5.10. Specifically, CBS outperforms HebCUP by 8.53% in terms of Accuracy, 35.16% in terms of Recall@5, 4.85% in terms of AED and RED, 1.67% in terms of BLEU-4, and 12.09% in terms of ESS ratio. CBS also outperforms CUP by 38.41% in terms of Accuracy, 10.52% in terms of Recall@5, 1.24% in terms of AED and RED, 1.62% in terms of BLEU-4, and 26.69% in terms of ESS ratio.

(4) The row of CBS-R in Table 5.10 shows the performance of CBS with random category classification. We find that CBS outperforms CBS-R by 22.48% in terms of Accuracy, 22.50% in terms of Recall@5, 3.12% in terms of AED and RED, 1.89% in terms of BLEU-4, and 16.20% in terms of ESS ratio. This demonstrates that the accurate category prediction is substantially significant to CBS for code-comment synchronization. Besides, the row of CBS-Max in Table 5.10 shows the upper limit of CBS. As can be seen, the Accuracy, Recall@5, BLEU-4, and ESS ratio of CBS can reach to 90.35%, 91.53%,

96.89%, 94.53% of those of CBS-Max. Both AED and RED values of CBS can reach to 105.44% of those of CBS-Max. [16]

(5) The row of -HebCUP, -CUP, and -CBS-R under W/T/L in Table 5.10 shows that CBS outperforms HebCUP, CUP, and CBS-R ten times in terms of all evaluation metrics. The Wilcoxon signed-rank test also shows that CBS statistically significantly outperforms CUP, HebCUP, and CBS-R for all evaluation metrics at the significance threshold of 0.05, 0.01, and 0.005, where the *p-value*s for each are all equal to 1.95e-3. [17]

Conclusion

(1) CBS statistically significantly outperforms CUP and HebCUP in terms of all evaluation metrics, because of the classifying before synchronizing strategy.

(2) The accurate category prediction is substantially significant to CBS for code-comment synchronization.

5.6 Further Exploration For CBS

In this section, we propose a series of Research Questions (RQs) to further analyze the performance of CBS and discuss the contributions of each module.

[16]The ratio here exceeds 100% because, as we mentioned in Section 5.2.2, for RED and AED, the smaller, the better. 105.44%=0.937/0.889 (i.e., the RED of CBS/the RED of CBS-Max, taking RED an example), which illustrates CBS-Max is better than CBS in terms of RED and AED.

[17]The *p-value*s are the same here because Wilcoxon signed-rank test only takes into account the signs of the differences of values of every pair of data, and it does not take into account how large is such a difference. In our comparison, CBS outperforms each of the other approaches on all evaluation metrics, and each experiment, thus the *p-value*s via Wilcoxon signed-rank test are the same.

Table 5.9: The performance of our proposed CBS approach on the testing set

Approach	**Accuracy (%)**	**Recall@5 (%)**	**AED**	**RED**	**BLEU-4 (%)**	**ESS ratio (%)**
HebCUP Side	59.51[2062.00/3465]	60.72	2.050	0.729	81.77	69.52
CUP Side	8.83±0.34[506.80±19.53/5739]	20.54±0.40	4.474±0.045	1.018±0.010	67.96±0.21	15.53±1.14
CBS	27.91±0.21[2568.80±19.53/9204]	35.67±0.25	3.561±0.028	0.937±0.007	73.16±0.13	35.85±0.71

$^{\Phi}$ The number of *correct synchronization*s are presented in square brackets via the format of "[$\mu \pm \theta$/assigned testing samples]".

Table 5.10: The performance of our proposed CBS approach and baselines on the testing set

Approach	**Accuracy (%)**	**Recall@5 (%)**	**AED**	**RED**	**BLEU-4 (%)**	**ESS ratio (%)**
HebCUP	25.71[2367.00]*	26.39*	3.742*	0.985*	71.96*	31.99*
CUP	20.16±0.73[1855.90±67.03]*	32.28±0.74*	3.606±0.050*	0.949±0.013*	71.99±0.24*	28.30±1.21*
Ours						
CBS-R	22.79±0.24[2094.40±11.05]*	29.12±0.22*	3.676±0.021*	0.968±0.005*	71.80±0.01*	30.85±0.87*
CBS	**27.91±0.21[2568.80±19.53]**	**35.67±0.25**	**3.561±0.028**	**0.937±0.007**	**73.16±0.13**	**35.85±0.71**
CBS-Max	30.88±0.09[2842±8.55/9204]	38.97±0.31	3.378±0.031	0.889±0.008	75.51±0.14	37.93±0.74
W/T/L						
- HebCUP	10/0/0	10/0/0	10/0/0	10/0/0	10/0/0	10/0/0
- CUP	10/0/0	10/0/0	10/0/0	10/0/0	10/0/0	10/0/0
- CBS-R	10/0/0	10/0/0	10/0/0	10/0/0	10/0/0	10/0/0

$^{\Phi}$ The number of *correct synchronization*s are presented in square brackets via the format of "[$\mu \pm \theta$]", where the number of total testing samples is 9204. Besides, * refers to that CBS outperforms the corresponding approach statistically significantly and their *p-value*s all equal to 1.95e-3 < 0.005.

5.6.1 RQ1: How do other classifiers influence the performance of CBS?

Motivation: Although we have selected LightGBM as the base classifier of the MSEL algorithm due to its achievement of the highest F1-Score under the 10-fold cross-validation, exploring the performance of CBS with other classifiers is also necessary, thus we also list their corresponding experimental results (reported by confidence intervals) in Table 5.12 to do further analysis. In addition, since the code-comment synchronization results depend heavily on the sample allocation guided by the MSEL algorithm, we also present Table 5.11 to demonstrate the category prediction results of the MSEL algorithm with different base classifiers on the testing set.

Results: According to Table 5.11, still, LightGBM embedded in MSEL performs better than other classifiers in terms of F1-Score on the testing set, Naive Bayes achieves the highest Recall value. Decision Tree substitutes CNN and performs best in terms of Precision. Except for the Bi-LSTM and CNN, the performance of the rest of the classifiers is almost consistent with those under the 10-fold cross-validation. It is worth noting that, comparing with the 10-fold cross-validation, the performance of Bi-LSTM and CNN drops a lot on the testing set. Thus we infer that the patterns learned by Bi-LSTM and CNN directly from code changes and old comments do not match those in the test set, leading to the overfitting of these two models. The unmatched pattern learned may attribute to the difficulty in directly learning on the raw data, which is not intuitive enough to capture the differences between heuristic and non-heuristic samples, especially compared with our proposed features. On the contrary, other classifiers exploit our proposed features can intuitively learn the differences between heuristic and non-heuristic samples, where their learned patterns can also be easily generalized to the unseen data.

On the other hand, as shown in Table 5.12, using LightGBM as the base classifier of the

MSEL algorithm makes CBS perform best in terms of all evaluation metrics. We can also observe that the performance of CBS is approximately positively related with the F1-Score of category prediction models, which further indicates that CBS chooses the classifier with the highest F1-Score to embed in MSEL is reasonable.

In addition, as shown in Table 5.10 and Table 5.12, CBS with all base classifiers, except for Naive Bayes, Bi-LSTM, and CNN, outperforms HebCUP and CUP in terms of all evaluation metrics. Even for using Naive Bayes as the base classification model, CBS still performs better than CUP and HebCUP in terms of Accuracy, Recall@5, BLEU-4, and ESS ratio. Besides, although Bi-LSTM and CNN do not exploit our proposed features, adopting them in CBS still outperforms HebCUP and CUP on Recall@5, BLEU-4, AED, RED and ESS ratio. Above results indicate that as long as the classifier embedded in the MSEL algorithm has some ability in the category prediction for CCI samples, CBS is definitely able to further boost the performance of code-comment synchronization.

Table 5.11: The category prediction results of the MSEL algorithm with different base classifiers on the testing set

Classifier	**Precision**	**Recall**	**F1-Score**
Naive Bayes	49.78%	**89.17%**	63.89%
Random Forest	59.50%	86.07%	70.36%
Decision Tree	**59.87%**	85.77%	70.52%
LightGBM	59.51%	87.11%	**70.71%**
MLP	59.57%	83.79%	69.64%
Bi-LSTM	49.73%	68.85%	57.74%
CNN	53.24%	70.45%	60.65%

Table 5.12: The code-comment synchronization performance of CBS with different base classifiers on the testing set

Classifier	**Accuracy (%)**	**Recall@5 (%)**	**AED**	**RED**	**BLEU-4 (%)**	**ESS Ratio (%)**
Naive Bayes	27.57±0.22[2537.20±20.04]	34.65±0.25	3.625±0.028	0.954±0.007	72.65±0.14	35.62±0.71
Random Forest	27.79±0.24[2557.60±22.16]	35.61±0.28	3.563±0.028	0.938±0.007	73.19±0.13	35.70±0.73
Decision Tree	27.77±0.21[2555.70±19.15]	35.58±0.27	3.580±0.029	0.942±0.008	73.01±0.14	35.77±0.72
LightGBM	**27.90±0.21[2567.80±19.53]**	**35.67±0.25**	**3.562±0.028**	**0.937±0.007**	**73.16±0.13**	**35.85±0.71**
MLP	27.69±0.23[2549.00±21.50]	35.49±0.26	3.564±0.030	0.938±0.008	73.15±0.14	35.62±0.72
Bi-LSTM	25.47±0.24[2344.60±22.27]	33.71±0.27	3.567±0.027	0.939±0.007	72.79±0.12	32.89±0.72
CNN	25.70±0.22[2366.10±20.70]	34.05±0.27	3.564±0.031	0.938±0.008	72.84±0.13	32.77±0.73

$^{\Phi}$ The performance on each metric is recorded via confidence interval ($\mu \pm \theta$) as introduced in Section 5.5.2, and the number of *correct synchronization*s are presented in square brackets, where the number of total testing samples is 9204.

5.6.2 RQ2: How does the Multi-Subset Ensemble Learning (MSEL) influence the performance of CBS?

Motivation: In this study, we propose a Multi-Subsets Ensemble Learning (MSEL) algorithm to alleviate the class imbalance problem in the category prediction for CCI samples. The class imbalance problem would result in lots of actual heuristic-prone samples being predicted to be non-heuristic-prone due to their low percentage (around 23%-25% shown in Table 5.3) in the overall dataset, thus causing a great loss in the accurate code-comment synchronization of our CBS approach. Therefore, we validate the effectiveness of the MSEL algorithm in this subsection.

Results: Table 5.13(A) presents the category prediction results of MSEL with LightGBM and the individual LightGBM algorithm on the testing set. As can be seen, using the multi-subsets ensemble learning strategy can largely improve the Recall value (up to 87.11%), which indicates that many more actual heuristic-prone samples (35.03% higher than using the individual LightGBM algorithm) can be found and assigned to HebCUP for *correct synchronization*. Although the Precision value decreases by 19.64%, the values of Recall and F1-score increase by 35.03% and 2.55%, respectively. It indicates that using the MSEL algorithm contributes to the overall performance improvement of category prediction for CCI samples.

Table 5.13(B) presents the code-comment synchronization results (reported by confidence intervals) of CBS with and without MSEL on the testing set. Despite the certain loss in terms of AED, RED (losses by 1.46%), and BLEU-4 (losses by 0.30%), CBS with MSEL outperforms that without MSEL in terms of Accuracy (improves by 9.49%), Recall@5 (improves by 3.23%), and ESS ratio (improves by 7.79%), showing that utilizing the MSEL algorithm can largely improve the success rate of *correct synchronization*s on both the first and five attempts. Simultaneously, the improvement in

ESS ratio demonstrates the synchronized comments of more CCI samples become closer to their references. The difference in performance between CBS with and without MSEL originates from the sample allocation guided by the category prediction model. More specifically, despite the high precision, the CBS approach without MSEL finds fewer actual heuristic-prone samples (1527 of 2367 in total), leading to most of the samples being assigned to and handled by CUP Side. As such, CUP Side dominates the CBS framework and exhibits its advantages of the generalization capability on a large proportion of the total samples, thereby achieving better performance in terms of AED, RED, and BLEU-4. However, since HebCUP Side only conducts the synchronization on a small part of samples, it contributes little to Accuracy, which further declines its contribution on Recall@5. Similarly, the ESS ratio drops due to the same reason. Thus the CBS approach without MSEL does not perform well on these three metrics. On the contrary, although with relatively low precision, the CBS approach with MSEL identifies most of the actual heuristic-prone samples (2062 of 2367 in total), thus causing an opposite result compared with the CBS without MSEL.

In addition, we find that CBS without MSEL performs worse than HebCUP by 0.89% in terms of Accuracy, while CBS with MSEL statistically significantly outperforms CUP and HebCUP in terms of all evaluation metrics as we mentioned in Section 5.5.3. We further conduct the Wilcoxon signed-rank test [130] with Benjamini-Hochberg (BH) for *p-value* adjustment [26] to analyze whether there is a statistically significant difference between CBS without MSEL and the two baselines. The results show that the performance superiority of HebCUP against CBS without MSEL in terms of Accuracy is not statistically significant, where the *p-value* is $0.084 > 0.05$. Yet, CBS without MSEL statistically significantly outperforms CUP and HebCUP in terms of all other metrics, where the *p-value*s are all equal to $1.95e\text{-}3 < 0.05$. Since both approaches achieve better

performance compared with the state-of-the-art baselines, we recommend practitioners adopt the MSEL algorithm in CBS approach when they prefer more correct and effective synchronizations. On the contrary, if they prefer the CBS approach to be more generalized on CCI samples, they had better adopt the MSEL-free version of our CBS approach.

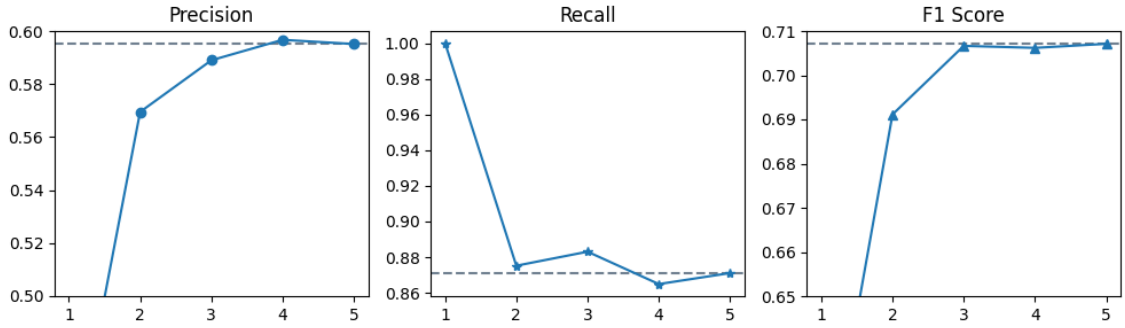


Figure 5.10: The category prediction results on each evaluation metric among different feature levels

5.6.3 RQ3: How do the proposed features influence the performance of CBS?

Motivation: We propose five original features, which we adopt in the MSEL algorithm with lightGBM, and obtain the best category prediction performance. Nevertheless, since different features have their heterogeneous characteristics of different degrees, and different combinations among these features will produce various performance on category prediction and further affect the code-comment synchronization results. Therefore, we investigate the impact of our proposed five features on CCI sample category prediction and code-comment synchronization.

Results: There are totally $31 (= C_5^1 + C_5^2 + C_5^3 + C_5^4 + C_5^5)$ different feature combinations. For each one, we first conduct the 10-fold cross-validation experiment on the mixed dataset (i.e., training set + validation set) to evaluate its performance on the CCI sample

Table 5.13: The experimental results for exploring the effect of MSEL algorithm

(A) The Category Prediction Result with/without MSEL Algorithm

Structure	Precision	Recall	F1-Score
The individual LightGBM	**74.05%(1527/2062)**	64.51%(1527/2367)	68.95%
MSEL With LightGBM	59.51%(2062/3465)	**87.11%(2062/2367)**	**70.71%**

(B) The Code-Comment Synchronization Results of CBS with/without MSEL Algorithm

Structure	Accuracy (%)	Recall@5 (%)	AED	RED	BLEU-4 (%)	ESS Ratio (%)
CBS without MSEL	25.49±0.25[2346.1±22.89]	34.55±0.39	**3.510±0.035**	**0.924±0.009**	**73.38±0.16**	33.26±0.83
CBS with MSEL	**27.90±0.21[2567.80±19.53]**	**35.67±0.25**	3.562±0.028	0.937±0.007	73.16±0.13	**35.85±0.71**

$^{\Phi}$ The performance on each metric in (B) is recorded via confidence interval ($\mu \pm \theta$) as introduced in Section 5.5.2, and the number of *correct synchronization*s are presented in square brackets, where the number of total testing samples is 9204.

Table 5.14: Top performing feature set of each level evaluated via 10-fold cross-validation

Level	**Feature Set**	**Precision**	**Recall**	**F1-Score**
1	MatchedLevelsNum	40.73%	100%	57.89%
2	MatchedLevelsNum; TotalChangedNum	55.82%	89.46%	68.75%
3	ReplaceRate; MatchedLevelsNum; TotalChangedNum	58.14%	90.02%	70.65%
4	ReplaceRate; MatchedLevelsNum; LongestChangedSeq; TotalChangedNum	59.21%	90.21%	71.49%
5	ReplaceRate; MatchedLevelsNum; NonLetterCount; LongestChangedSeq; TotalChangedNum	59.69%	90.57%	71.96%

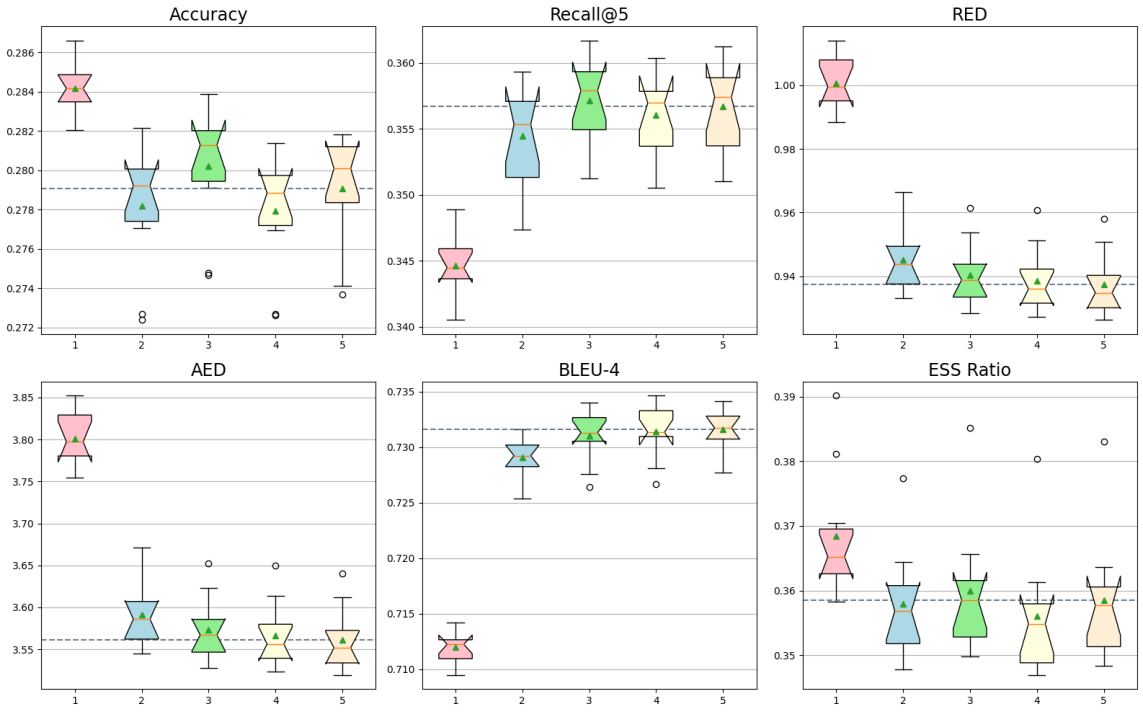


Figure 5.11: The code-comment synchronization performance on each evaluation metric among different feature levels

category prediction. To simplify the experimental analysis, we partition the 31 feature combinations to five levels of feature sets (i.e., from *level*-1 with one feature to *level*-5 with five features). Next, the best performing (evaluated by F1-Score) feature combination among each level is selected for further comparison: {*level*-1: *MatchedLevelsNum*; *level*-2: *MatchedLevelsNum*, *TotalChangedNum*; *level*-3: *ReplaceRate*, *MatchedLevelsNum*, *TotalChangedNum*; *level*-4: *ReplaceRate*, *MatchedLevelsNum*, *LongestChangedSeq*, *TotalChangedNum*; *level*-5: *Repl- aceRate*, *MatchedLevelsNum*, *NonLetterCount*, *LongestChangedSeq*, *TotalChangedNum*}, as shown in Table 5.14. This step cannot be conducted directly on the testing set. Because before the real comparison among the five groups, we hope to keep the testing set unseen to us.

Hereon, we narrow down the comparisons between feature combinations from 31 groups to 5 groups. Then, we adopt each selected feature combination of each level to experiment on the testing set for CCI sample category prediction, and present Figure 5.10 to display the variation of category prediction results with different levels of feature sets. We can find that CBS on feature set *level*-4 performs the best in terms of Precision (59.67%), CBS on feature set *level*-1 performs the best in terms of Recall (100%), and CBS on feature set *level*-5 performs best in terms of F1-Score (70.71%). The feature set *level*-1 only includes *MatchedLevelsNum*, and CBS trained on this feature can recognize all actual heuristic-prone samples, thus achieving the score of 100% on Recall, but with very low Precision as introduced in Section 5.3.4. Afterward, with considering more and more features, decision boundaries of classification gradually become rigorous, leading to an increasing trend of the Precision and F1-Score.

Finally, we present the code-comment synchronization results of CBS with different feature combinations on the testing set in Figure 5.11. We find that (1) CBS with the feature set *level*-1 achieves the best performance in terms of Accuracy (28.42%) and ESS

ratio (36.84%) on average but performs the worst in other metrics. Due to its extremely unbalanced performance on different evaluation metrics, we do not recommend adopting only one feature for category prediction in our CBS approach; (2) CBS with the feature set *level*-3 performs the best in terms of Recall@5 (35.72%) on average and ranks second in terms of Accuracy (28.07%) and ESS ratio (36.08%). Besides, it ranks third on RED (0.941), AED (3.574), and BLEU-4 (73.08). (3) CBS with the feature set *level*-5, on average, performs the best in terms of RED (0.937), AED (3.562), and BLEU-4 (73.16), ranks second in terms of Recall@5 (35.67%), and ranks the third in terms of Accuracy (27.90%) and ESS ratio (35.85%); The different superiority between CBS with the feature sets *level*-3 and *level*-5 is due to their different performance in the category prediction for CCI samples. Firstly, both feature sets achieve quite high performance on F1-Score, ranking second (70.66%) and best (70.71%), respectively. Besides, if we do not consider the feature set *level*-1, CBS with the feature set *level*-3 achieves the highest performance on Recall (88.31%), which causes relatively more actual heuristic-prone samples to be allocated to HebCUP Side. Therefore, in the subsequent code-comment synchronization, CBS with the feature set *level*-3 achieves higher performance in terms of Accuracy and ESS ratio, where a large amount of correctly synchronized samples further contributes to Recall@5 and makes it higher. CBS with the feature set *level*-5 performs better in terms of Precision (59.51%) but with lower Recall (87.11%), compared with feature set *level*-3. Therefore, relatively more samples are allocated to CUP Side for code-comment synchronization, and higher RED, AED, and BLEU-4 are achieved. However, since HebCUP Side only synchronizes a small part of samples in this case, it contributes little to ESS ratio and Accuracy, which further declines its contribution to Recall@5.

In summary, we recommend practitioners select appropriate features according to their preferences when utilizing our CBS approach to code-comment synchronization. If they

prefer more correct and effective synchronizations, they can adopt $\{ReplaceRate, MatchedLevelsNum, TotalChnagedNum\}$ to train the category classification model in CBS; while if they prefer the approach with a better generalization capability, they had better choose all five features.

5.7 Discussion

This section discusses the situations CBS may fail and the implications of improving code-comment synchronization via category prediction of CCI samples.

Table 5.15: The examples where CBS fails

ID	Code Change	Comments

1	Project: plutext/docx4j
	Commit ID: 3151cd20c677418e5c5f4f0a3b9bc05ce7fc3611
	- public String getValue() {
	+ public AppearanceType getAppearance() {
	- return value;
	+ return appearance;
	}
		Old: Gets the value of the value property.
		New: Gets the value of the appearance property.
		HebCUP: Gets the Appearance of the Appearance property.
		CUP: Gets the value of the appearance property.
		CBS: Gets the Appearance of the Appearance property.

2	Project: OpenGamma/Strata
	Commit ID: 09358e034d05edb45dd6872234b56964d1b46701
	- public double getPrice() {
	+ public Optional<TradedPrice> getTradedPrice() {
	- return price;
	+ return Optional.ofNullable(tradedPrice);
	}
		Old Comment: Gets the price that was traded, in decimal form.
		New Comment: Gets the price that was traded, together with the trade date, optional.
		HebCUP: Gets the Optional.ofNullable that was traded in decimal form.
		CUP: Gets the price that was traded,in decimal form.
		CBS: Gets the price that was traded,in decimal form.

5.7.1 Where Does CBS Fail

Based on our analysis on the experimental results of CBS, we find it fails in two situations, i.e., (1) misclassification, (2) even correctly classified, CUP Side may not be able to conduct the *correct synchronization*. We present Table 5.15 with two samples to illustrate the above situations, where tokens to be deleted in old code or old comments are

highlighted in red, tokens to be added or correctly updated are highlighted in green, and tokens that are wrongly updated are highlighted in orange.

Sample 1 is one of the failed examples induced by misclassification. This sample is non-heuristic-prone, but with a high ReplaceRate (0.75) and MatchedLevelsNum (3), meanwhile, with relatively small values for NonLetterCount (0), LongestChangedSeq (2), and TotalChangedNum (3). Thus it is undoubtedly predicted as heuristic-prone and allocated to HebCUP Side to handle. As one of the potential reasons, the misclassification can be summarized as a lack of discriminative features, the currently proposed features cannot cover all samples' characteristics, thus failing on some of them. On the other hand, this sample also exposes one of the weaknesses of HebCUP. It has difficulties conducting *correct synchronization*s when the replacement pairs cover the tokens that do not need to be updated in old comments. For example, the replacement pair found by HebCUP in this sample is *key:* "value" and *value:* "Appearance". However, there are two tokens of "value" in the old comment, where the first one should be kept intact, and the second one should be updated. In this case, HebCUP cannot tell and uniformly replaced them with "Appearance".

Fundamentally, the second situation can be attributed to the difficulties for some of the CCI samples that the current code-comment synchronizers cannot handle. As we mentioned in Section 5.3.3, we partition the samples into heuristic-prone and non-heuristic-prone according to the code-comment synchronization results of HebCUP. Thus, in category prediction, samples that cannot be correctly synchronized by HebCUP are almost allocated to CUP, where though CUP has a more powerful generalization capability than HebCUP, there is still a great amount of them that are hard to be correctly synchronized via CUP. As such, the failure of the second situation occurs. For example, the reference comment (i.e., new comment) in Sample 2 has a series of tokens (e.g.,

"together", "with", "the", "trade", and "date") that are not shown in both old and new code. Thus it is hard to infer the *correct synchronization* for the old comment, even for CUP with relatively powerful generalization capability. When we refer to the file associated with this sample, we find "tradedPrice" in the new code is an object of class "TradedPrice" that extends the class "Optional" and has two values, one is "tradeDate", and the other one is "price". Therefore, CCI samples like this are nearly impossible for current state-of-the-art code-comment synchronizers to handle successfully due to the unseen information outside the method. Therefore, combining context information may be a potential solution for these tricky samples.

5.7.2 Implications

Implications for professionals: As we mentioned in Chapter 1, inconsistent or obsolete comments (i.e., bad comments) are enormous risks during program development and code maintenance, which may inject unintended bugs and hinder program comprehension. On the other hand, such kind of bad comments are always ignored by developers and manually modifying them is labor-intensive work. Hence, accurately and efficiently synchronizing comments with code changes in an automatic manner has become vitally important. However, CCI samples in practice are complicated. For example, some change contents in comments are not shown in the corresponding code changes, such cases (e.g., Sample 1 in Figure 5.7) indeed are hard to design hand-crafted rules to cover. Whereas, other samples may have obvious replacement pairs between old and new code, but due to some interference factors (e.g., high-frequency tokens [139, 124, 28]), those deep learning-based approaches may fail to conduct the *correct synchronization* (e.g., Sample 2 in Figure 5.7). Instead, heuristic-based approaches can easily handle them. To this end,

this chapter provides a potential solution, which is classifying CCI samples before synchronizing them with code changes.

In practice, professionals can leverage both the deep learning-based (i.e., CUP) and heuristic-based approach (i.e., HebCUP) in code-comment synchronization via a category prediction for each CCI sample in advance. On the other hand, the classification is based on a machine learning model (i.e., lightGBM) and our proposed five features, thus the classification is also very efficient. According to our statistics on the 9204 testing samples, the category prediction costs each of them only 0.063 seconds at most, showing that the category prediction phase can be totally transparent for professionals, but, as presented in this chapter, the performance of code-comment synchronization can be largely boosted. Therefore, we claim that it is substantially beneficial for professionals to adopt CBS for code-comment synchronization in their program development and maintenance.

Implications for researchers: In this chapter, CUP and HebCUP are combined via category prediction, and the extensive experiment results have proved the effectiveness of such a combination that relies on the prediction for model proneness of CCI samples. Therefore, as the first attempt, this work testifies to the feasibility of a new direction that is different from improving a single general model in the code-comment synchronization field. To this end, this chapter can be seen as a footstone and vitally important, because its methodology can be further popularized and iterated with the development of the code-comment synchronization field. For example, with more and more code-comment synchronizers are proposed in the future, the CCI samples can be further subdivided and continue to improve the performance via multi-categorical classification and combination.

On the other hand, based on our definition and partition of heuristic-prone and non-heuristic-prone samples, designing models for samples of specific categories became actionable and promising to further improve the performance. For example, CCI samples

whose comment change contents are not shown in the code changes are tough to handle even for the current deep learning approach, i.e., CUP, as we mentioned in Section 5.7.1. Therefore, the proposal of solutions for such samples can be a way forward in the field of code-comment synchronization, such as combining the context information around the code change snippets.

Finally, as we always highlight in this chapter, promoting the performance of category prediction is also a helpful research direction. One of the most promising perspectives is exploring more discriminative features to differentiate between categories because such a perspective can effectively improve the classification results while ensuring the efficiency of code-comment synchronization.

5.8 Threats to Validity

In this section, we clarify the threats to internal, external, and construct validity.

5.8.1 Internal validity

Similar to a vast amount of studies [143, 17, 44, 52, 71], the implementation of approaches and evaluation metrics can be one of the threats to the internal validity in this chapter. For baseline replication, we directly used the source code published in their original papers. For the implementation of the evaluation metrics that appeared in baselines, we adopted Lin et al.'s [66] code to implement Accuracy and Recall@5 since the case and punctuation marks in the comments indeed do not affect developers' comprehension to code comments. Then, we adopted Liu et al.'s [73] code to implement AED, RED, and BLEU-4 except we ignored the case since Lin et al. [66] did not publish the source code of these metrics. Besides, to reduce the threats to the implementation of our proposals (e.g., CBS framework and

ESS Ratio), we double-checked and carefully tested the code. Besides, we also published the whole project of CBS to enable other researchers to replicate and extend our work. Therefore, the threat of implementation is limited.

On the other hand, artifacts adopted in this chapter may be another threat to the internal validity, which is also referred to as instrumentation threats [1]. We adopted a series of third-party tools across the whole work, such as TensorFlow [18] and sci-kit learn toolkit [96] for model construction, and the *diff* tool [19] for code change alignment. Since the above artifacts are continuously tested and widely used in both academia and industries, we believe the threats of instrumentation to internal validity can be minimized.

Besides, CCI samples with relatively large comment changes were filtered out by Liu et al. [73] because they argued that these filtered samples should be handled by code summarization models to generate new comments instead of code-comment synchronization models. However, the filtering criterion was based on their experience, leading to the adopted dataset may not cover all the code-comment synchronization situations, thus the selection of CCI samples may affect the internal validity.

Last but not least, due to the restriction of time and computation resources, the tuning scopes of hyper-parameters for base classifiers in this chapter are limited. Thus, other hyper-parameter settings of lighGBM or other classifiers may yield better results, which may become one of the threats to internal validity.

5.8.2 External Validity

Threats to external validity focus on the experimental dataset used in this work. The dataset is only built from Java repositories and consists of comments only in the

[18] https://www.tensorflow.org/
[19] https://docs.python.org/3/library/difflib.html

granularity of the method level, which may not be representative of all programming languages and comment types. Nevertheless, Java is undoubtedly one of the most universal and popular programming languages, while method comments are also widely studied and referred to by researchers in other fields of program comprehension, such as comment generation [43, 44, 134, 60] and comment classification [94, 95]. In addition, the dataset is extracted from the 1496 popular Github repositories, which reflects a relatively strong representation of this dataset. On the other hand, our proposed CBS approach is also language-agnostic and can be easily applied to projects of other programming languages. Therefore, there is little threat to external validity.

5.8.3 Construct Validity

Threats to construct validity are related to the adoption of the evaluation metrics. In this chapter, we employ Accuracy, Recall@5, AED, RED, BLEU-4, and ESS ratio to evaluate the performance of code-comment synchronization approaches. The first five metrics are summarized from the works of Liu and Lin et al. [66, 73] whereas the last one is proposed in this work as complementation. Accuracy and Recall@5 evaluate to what extent an approach can synchronize comments correctly. AED and RED measure the average edits that developers need to perform to correctly update comments after using the code-comment synchronizers. ESS ratio is a complementary metric for AED and RED, which measures the ratio of samples whose edit distances are reduced after the synchronization. BLEU-4 is an evaluation metric widely used in machine translation tasks [119, 38, 97, 142] and related software engineering studies [43, 2, 44, 108], which can provide us an effective measurement of comment quality. Although these automated evaluation metrics cannot perfectly represent human judgment, they can provide a quick and quantitative assessment of code-comment synchronization approaches.

5.9 Chapter Summary

In this chapter, we investigate the performance of two state-of-the-art code-comment synchronization approaches, i.e., the deep learning-based CUP and the heuristic-based HebCUP on the unified evaluation criteria. We find that the two approaches of different mechanisms have their heterogeneous characteristics and are prone to accurately synchronize different kinds of Code-Comment Inconsistent (CCI) samples. Motivated by this finding, we define two categories (i.e., heuristic-prone and non-heuristic-prone) for the CCI samples and propose five features to assist category prediction. Then, we propose a composite approach named CBS, which combines the advantages of CUP and HebCUP. CBS firstly constructs a Multi-Subsets Ensemble Learning (MSEL) classification model based on the training samples. Then, CBS employs the trained MSEL model to predict whether the new CCI sample can be successfully handled by HebCUP (i.e., the predicted category of the sample is heuristic-prone). If so, CBS allocates the sample to HebCUP to conduct the code-comment synchronization; otherwise, the sample is assigned to CUP. The experimental results show that our composite approach CBS outperforms CUP and HebCUP that do not consider the category prediction for CCI samples by 8.53%–38.41% in terms of Accuracy, by 10.52%–35.16% in terms of Recall@5, by 1.24%–4.58% in terms of AED and RED, by 1.62%–1.67% in terms of BLEU-4, and by 12.09%–26.69% in terms of ESS ratio. In summary, our research emphasizes that considering automatically category classification for CCI samples before synchronizing their comments is conducive and can significantly boost the performance of code-comment synchronization.

Chapter 6

Conclusions and Future Work

This thesis proposes a series of code representation learning approaches to tackle the research challenges that are implied in three intelligent software engineering tasks, i.e., smart contract vulnerability auditing, code summarization, and code-comment synchronization. In this chapter, we provide a summary of the work accomplished in this thesis and propose directions for future research.

6.1 Conclusion

In the area of smart contract vulnerability auditing, we propose MSgram. It represents smart contract source code with three semantic sequences, including the aspects of textual, structural, and combined. Subsequently, we make use of n-gram language models as the learning algorithm to capture their distinctive semantics and carry out the vulnerability prediction, i.e., vulnerability auditing. Finally, prediction results are summarized by either *Intersection* or *Union* strategy from each semantic sequence, thereby improving the model performance. Extensive experiments are conducted on 7,200 smart contract samples and demonstrate our proposed method is capable of detecting vulnerabilities and is

competitive with the baseline, with improved precision of over 44% when *Intersection* is applied, and with improved Recall by over 300% and F-measure by 220% at most when *Union* is applied.

In the area of code summarization, we propose MMTrans. It exploits multiple modalities of AST (i.e., SBT sequences and graphs) as source code representations. Besides, we incorporate Transformer structures to learn the intrinsic semantics of both modalities and encode long-term dependencies between code tokens. We carry out experiments on smart contracts and accordingly build an exclusive dataset with over 347,410 <method, comment>. Afterward, we extensively assess the performance of the MMTrans against three recently proposed approaches and reveal that the MMTrans performs better than the baselines by 17.23%-58.45% in terms of S-BLEU, by 20.74%-62.49% in terms of C-BLEU, by 6.78%-55.60% in terms of ROUGE-LCS F1, and by 10.17%-45.50% in terms of METEOR.

In the area of code-comment synchronization, we put forward CBS, which means classifying before synchronizing. Specifically, CBS first conduct a classification (either heuristic-prone or non-heuristic-prone) for CCI samples, then allocate them to their suitable models (i.e., HebCUP or CUP) for synchronization. In order to improve the classification performance, we manually craft five features to represent code changes and their interaction with old comments. Subsequently, we design an MSEL algorithm as the machine learning-based classifier. Extensive experiments reveal the powerful classification performance of our proposal, by which the capability of code-comment synchronization is also improved. To be specific, CBS statistically significantly outperforms the two baselines, i.e., CUP and HebCUP, and obtains an average improvement of 23.47%, 22.84%, 3.04%, 3.04%, 1.64%, and 19.39% in terms of Accuracy, Recall@5, AED, RED, BLEU-4, and ESS ratio, respectively, which highlights

that category prediction for CCI samples can boost the code-comment synchronization performance.

6.2 Future Work

Although our proposals for code representation learning obtain significant success, there are still some potential directions to achieve further improvements based on our proposed approaches.

In the area of smart contract vulnerability auditing, we currently make use of n-gram language models to capture sequence semantics from multiple perspectives. When combining the auditing results of different perspectives, we let the practitioners select either *Union* or *Intersection* strategy. In the future, we will design a more generalized framework to fully exploit the advantages of the above two strategies rather than applying them individually in practice. Besides, n-gram language models, although efficient, cannot construct an end-to-end mapping between code representation and their learning objectives [101]. To this end, we will investigate the utilization of deep neural networks in this field and fuse the deep learning techniques into our framework to construct a more robust learning paradigm.

In the area of code summarization, we represent source code as two modalities of AST. Nonetheless, source code can also be represented as other heterogeneous modalities, such as control-flow graphs and program dependency graphs. Different program representations hold different semantics of different aspects. In the future, we will attempt other modalities of source code and incorporate them into the MMTrans framework, such that the model can understand the code functionalities or intentions more deeply.

In the area of code-comment synchronization, we propose a two-phase approach,

namely CBS, which classifies samples' categories and then allocates them to suitable models to synchronize comments. To accomplish this idea, we manually craft five features to represent code changes and their interactions with old comments, thereby facilitating the follow-up classification powered by the proposed MSEL algorithm. However, CBS is only based on two models in the code-comment synchronization field. Based on this methodology of classifying before synchronizing, a potential future direction is further subdividing CCI samples according to their proneness to various code-comment synchronization models. In this way, code-comment synchronization performance can be consistently improved with more and more powerful models proposed. Besides, as there're still some failure cases induced by category misclassification, we will continue exploring more discriminative features to remedy these failed scenarios.

Bibliography

[1] Silvia Abrahao, Carmine Gravino, Emilio Insfran, Giuseppe Scanniello, and Genoveffa Tortora. Assessing the effectiveness of sequence diagrams in the comprehension of functional requirements: Results from a family of five experiments. *IEEE Transactions on Software Engineering*, 39(3):327–342, 2012.

[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online, July 2020. Association for Computational Linguistics.

[3] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. IEEE, 2017.

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[5] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *7th International Conference*

on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net, 2019.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[7] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6*, pages 164–186. Springer, 2017.

[8] Morena Barboni, Andrea Morichetta, and Andrea Polini. Sumo: A mutation testing strategy for solidity smart contracts. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 50–59. IEEE, 2021.

[9] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

[10] William H Beyer. *Handbook of tables for probability and statistics*. CRC Press, 2019.

[11] Xavier Bouthillier and Gaël Varoquaux. *Survey of machine-learning experimental methods at NeurIPS2019 and ICLR2020*. PhD thesis, Inria Saclay Ile de France, 2020.

[12] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[13] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC Press, 1984.

[14] Nghi DQ Bui, Lingxiao Jiang, and Yijun Yu. Cross-language learning for program classification using bilateral tree-based convolutional neural networks. In *The Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*, 2018.

[15] Boxing Chen and Colin Cherry. A systematic comparison of smoothing techniques for sentence-level bleu. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 362–367, 2014.

[16] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)*, 53(3):1–43, 2020.

[17] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. Why my code summarization model does not work: Code comment improvement with category prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–29, 2021.

[18] Alfonso Cimasa, Anna Corazza, Carmen Coviello, and Giuseppe Scanniello. Word embeddings for comment coherence. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 244–251. IEEE, 2019.

[19] Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. Coherence of comments and method implementations: a dataset and an empirical investigation. *Software Quality Journal*, 26(2):751–777, 2018.

[20] Sergio Cozzetti B de Souza, Nicolas Anquetil, and Káthia M de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd*

annual international conference on Design of communication: documenting & designing for pervasive information, pages 68–75, 2005.

[21] Michael Denkowski and Alon Lavie. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 376–380, 2014.

[22] Thomas J DiCiccio and Bradley Efron. Bootstrap confidence intervals. *Statistical Science*, 11(3):189–228, 1996.

[23] Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. Evaluating source code summarization techniques: Replication and expansion. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 13–22. IEEE, 2013.

[24] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020.

[25] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. Structured neural summarization. In *International Conference on Learning Representations*, 2019.

[26] JA Ferreira, AH Zwinderman, et al. On the benjamini–hochberg method. *Annals of Statistics*, 34(4):1827–1849, 2006.

[27] Markus Freitag and Yaser Al-Onaizan. Beam search strategies for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation*, pages 56–60, Vancouver, August 2017. Association for Computational Linguistics.

[28] Cuiyun Gao, Wenjie Zhou, Xin Xia, David Lo, Qi Xie, and Michael R. Lyu. Automating app review response generation based on contextual knowledge. *ACM Transactions on Software Engineering and Methodology*, 31(1):1–36, 2021.

[29] Verena Geist, Michael Moser, Josef Pichler, Stefanie Beyer, and Martin Pinzger. Leveraging machine learning for software redocumentation. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 622–626, 2020.

[30] Mingyang Geng, Shangwen Wang, Dezun Dong, Shanzhi Gu, Fang Peng, Weijian Ruan, and Xiangke Liao. Fine-grained code-comment semantic interaction analysis. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2022.

[31] Sergio Gonzalez, Salvador Garcia, Javier Del Ser, Lior Rokach, and Francisco Herrera. A practical tutorial on bagging and boosting based ensembles for machine learning: Algorithms, software tools, performance study, practical perspectives and opportunities. *Information Fusion*, 64:205–237, 2020.

[32] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations*, 2020.

[33] Kevin Gurney. *An introduction to neural networks*. CRC Press, 1997.

[34] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 223–226. IEEE, 2010.

[35] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*, pages 35–44, 2010.

[36] Sangmok Han, David R Wallace, and Robert C Miller. Code completion from abbreviated input. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 332–343. IEEE, 2009.

[37] Dorsaf Haouari, Houari Sahraoui, and Philippe Langlais. How good is your comment? a study of comments in java programs. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 137–146, 2011.

[38] Di He, Yingce Xia, Tao Qin, Liwei Wang, Nenghai Yu, Tie-Yan Liu, and Wei-Ying Ma. Dual learning for machine translation. In *Advances in Neural Information Processing Systems*, volume 29, pages 820–828, 2016.

[39] Kenneth Heafield. Kenlm: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 187–197, 2011.

[40] Emily Hill, Lori Pollock, and K Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *2009 IEEE 31st International Conference on Software Engineering*, pages 232–242. IEEE, 2009.

[41] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 518–529, 2020.

[42] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[43] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE, 2018.

[44] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, 25(3):2179–2217, 2020.

[45] Xing HU, Ge LI, Xin XIA, David LO, Shuai LU, and Zhi JIN. Summarizing source code with transferred api knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI 2018), Stockholm, Sweden, 2018 July 13*, volume 19, pages 2269–2275.

[46] Walid M Ibrahim, Nicolas Bettenburg, Bram Adams, and Ahmed E Hassan. On the relationship between comment update practices and software bugs. *Journal of Systems and Software*, 85(10):2293–2304, 2012.

[47] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 2073–2083, 2016.

[48] Bo Jiang, Ye Liu, and Wing Kwong Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269, 2018.

[49] Liangxiao Jiang, Dianhong Wang, Zhihua Cai, and Xuesong Yan. Survey of improving naive bayes for classification. In *International Conference on Advanced Data Mining and Applications*, pages 134–145. Springer, 2007.

[50] Hong Jin Kang, Tegawendé F Bissyandé, and David Lo. Assessing the generalizability of code2vec token embeddings. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1–12. IEEE, 2019.

[51] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, volume 30, pages 3146–3154, 2017.

[52] Dong Jae Kim, Nikolaos Tsantalis, Tse-Hsun Chen, and Jinqiu Yang. Studying test annotation maintenance in the wild. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 62–73, 2021.

[53] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[54] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[55] Serkan Kiranyaz, Onur Avci, Osama Abdeljaber, Turker Ince, Moncef Gabbouj, and Daniel J Inman. 1d convolutional neural networks and applications: A survey. *Mechanical Systems and Signal Processing*, 151:107398, 2021.

[56] Carsten Kolassa, Dirk Riehle, and Michel A Salim. The empirical commit frequency

distribution of open source projects. In *Proceedings of the 9th International Symposium on Open Collaboration*, pages 1–8, 2013.

[57] Bonan Kou, Shengmai Chen, Zhijie Wang, Lei Ma, and Tianyi Zhang. Is model attention aligned with human attention? an empirical study on large language models for code generation. *arXiv preprint arXiv:2306.01220*, 2023.

[58] FDC Kraaikamp and HLL Meester. A modern introduction to probability and statistics, 2005.

[59] Max Kuhn. Building predictive models in r using the caret package. *Journal of Statistical Software*, 28:1–26, 2008.

[60] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 184–195, 2020.

[61] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 795–806. IEEE, 2019.

[62] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[63] Joseph Lev et al. The point biserial coefficient of correlation. *Annals of Mathematical Statistics*, 20(1):125–126, 1949.

[64] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. A survey on the

security of blockchain systems. *Future Generation Computer Systems*, 107:841–853, 2020.

[65] Yuding Liang and Kenny Zhu. Automatic generation of text descriptive comments for code blocks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[66] Bo Lin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F Bissyandé. Automated comment update: How far are we? In *2021 29th IEEE/ACM International Conference on Program Comprehension (ICPC)*, pages 36–46. IEEE, 2021.

[67] Bo Lin, Shangwen Wang, Zhongxin Liu, Xin Xia, and Xiaoguang Mao. Predictive comment updating with heuristics and ast-path-based neural learning: A two-phase approach. *IEEE Transactions on Software Engineering*, 49(4):1640–1660, 2022.

[68] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[69] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 65–68, 2018.

[70] Han Liu, Chao Liu, Wenqi Zhao, Yu Jiang, and Jiaguang Sun. S-gram: towards semantic-aware security auditing for ethereum smart contracts. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 814–819, 2018.

[71] Zhiyong Liu, Huanchao Chen, Xiangping Chen, Xiaonan Luo, and Fan Zhou. Automatic detection of outdated comments during code changes. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pages 154–163, 2018.

[72] Zhongxin Liu, Xin Xia, David Lo, Meng Yan, and Shanping Li. Just-in-time obsolete comment detection and update. *IEEE Transactions on Software Engineering*, 49(1):1–23, 2023.

[73] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. Automating just-in-time comment updating. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 585–597. IEEE, 2020.

[74] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. A neural architecture for generating natural language descriptions from source code changes. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 287–292, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[75] Yangyang Lu, Zelong Zhao, Ge Li, and Zhi Jin. Learning to generate comments for api-based code snippets. In *Software Engineering and Methodology for Emerging Domains*, pages 3–14. Springer, 2017.

[76] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269, 2016.

[77] Xiaoxue Ma, Jacky Keung, Zhen Yang, Xiao Yu, Yishu Li, and Hao Zhang. Casms: Combining clustering with attention semantic model for identifying security bug reports. *Information and Software Technology*, 147:1–13, 2022.

[78] Yi-Fan Ma and Ming Li. Learning from the multi-level abstraction of the control flow graph via alternating propagation for bug localization. In *2022 IEEE International Conference on Data Mining (ICDM)*, pages 299–308. IEEE, 2022.

[79] Christopher Manning and Hinrich Schutze. *Foundations of statistical natural language processing*. MIT Press, 1999.

[80] Paul W McBurney and Collin McMillan. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(2):103–119, 2015.

[81] Patrick E McKnight and Julius Najab. Mann-whitney u test. *The Corsini Encyclopedia of Psychology*, pages 1–1, 2010.

[82] Tom Michael Mitchell et al. *Machine learning*, volume 1. McGraw-hill New York, 2007.

[83] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[84] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B Dasgupta, and Subhrajit Bhattacharya. Anomaly detection using program control flow graph mining from execution logs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 215–224, 2016.

[85] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys (CSUR)*, 33(1):31–88, 2001.

[86] Chao Ni, Wei Wang, Kaiwen Yang, Xin Xia, Kui Liu, and David Lo. The best of both worlds: integrating semantic features with expert features for defect prediction and localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 672–683, 2022.

[87] James R Norris. *Markov chains*. Number 2. Cambridge University Press, 1998.

[88] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. Listening to programmers - taxonomies and characteristics of comments in operating system code. In *2009 IEEE 31st International Conference on Software Engineering*, pages 331–341, 2009.

[89] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J Mooney. Deep just-in-time inconsistency detection between comments and source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 427–435, 2021.

[90] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond Mooney. Learning to update natural language comments based on code changes. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1853–1868, 2020.

[91] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[92] Reza M Parizi, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Amritraj Singh. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, pages 103–113, 2018.

[93] David Lorge Parnas. Precise documentation: The key to better software. In *The Future of Software Engineering*, pages 125–148. Springer, 2011.

[94] Luca Pascarella and Alberto Bacchelli. Classifying code comments in java open-source software systems. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 227–237. IEEE, 2017.

[95] Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. Classifying code comments in java software systems. *Empirical Software Engineering*, 24(3):1499–1537, 2019.

[96] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[97] Martin Popel, Marketa Tomkova, Jakub Tomek, Łukasz Kaiser, Jakob Uszkoreit, Ondřej Bojar, and Zdeněk Žabokrtskỳ. Transforming machine translation: a deep learning system reaches news translation quality comparable to human professionals. *Nature Communications*, 11(1):1–15, 2020.

[98] Pooja Rani, Sebastiano Panichella, Manuel Leuenberger, Andrea Di Sorbo, and Oscar Nierstrasz. How to identify class comment types? a multi-language approach for class comment classification. *Journal of Systems and Software*, 181:1–17, 2021.

[99] Inderjot Kaur Ratol and Martin P Robillard. Detecting fragile comments. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 112–122. IEEE, 2017.

[100] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 428–439, 2016.

[101] Hazem Peter Samoaa, Firas Bayram, Pasquale Salza, and Philipp Leitner. A systematic mapping study of source code representation for deep learning in software engineering. *IET Software*, 16(4):351–385, 2022.

[102] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press, 2008.

[103] Yusuke Shinyama, Yoshitaka Arahori, and Katsuhiko Gondow. Analyzing code comments to boost program comprehension. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 325–334, 2018.

[104] Kamilya Smagulova and Alex Pappachen James. A survey on lstm memristive neural network architectures and applications. *The European Physical Journal Special Topics*, 228(10):2313–2324, 2019.

[105] Xiaotao Song, Hailong Sun, Xu Wang, and Jiafei Yan. A survey of automatic generation of source code comments: Algorithms and techniques. *IEEE Access*, 7:111411–111428, 2019.

[106] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software engineering*, pages 43–52, 2010.

[107] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 101–110. IEEE, 2011.

[108] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 2–13, 2020.

[109] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality analysis of source code comments. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 83–92, 2013.

[110] Nataliia Stulova, Arianna Blasi, Alessandra Gorla, and Oscar Nierstrasz. Towards detecting inconsistent comments in java source code automatically. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 65–69. IEEE, 2020.

[111] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT Press, 2018.

[112] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: Bugs or bad comments?*/. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, pages 145–158, 2007.

[113] Lin Tan, Ding Yuan, and Yuanyuan Zhou. Hotcomments: how to make program comments more useful? In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, pages 1–6, 2007.

[114] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. acomment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 11–20. IEEE, 2011.

[115] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 260–269, 2012.

[116] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering*, 46(11):1200–1219, 2020.

[117] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 664–676, 2018.

[118] Betty Van Aken, Julian Risch, Ralf Krestel, and Alexander Löser. Challenges for toxic comment classification: An in-depth error analysis. pages 33–42, 2018.

[119] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[120] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones,

Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[121] Anh Viet Phan, Minh Le Nguyen, and Lam Thu Bui. Convolutional neural networks over control flow graphs for software defect prediction. In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 45–52, 2017.

[122] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 397–407, 2018.

[123] Danqing Wang, Pengfei Liu, Yining Zheng, Xipeng Qiu, and Xuanjing Huang. Heterogeneous graph neural networks for extractive document summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6209–6219, Online, July 2020. Association for Computational Linguistics.

[124] 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)*, 30(4):1–30, 2021.

[125] Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F. Wong, and Lidia S. Chao. Learning deep transformer models for machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1810–1822, Florence, Italy, July 2019. Association for Computational Linguistics.

[126] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 708–719, 2016.

[127] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, S Yu Philip, and Guandong Xu. Reinforcement-learning-guided source code summarization using hierarchical attention. *IEEE Transactions on software Engineering*, 48(1):102–119, 2020.

[128] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. Code generation as a dual task of code summarization. In *Advances in Neural Information Processing Systems*, pages 6563–6573, 2019.

[129] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 53–64. IEEE, 2019.

[130] Frank Wilcoxon. *Individual comparisons by ranking methods*. Springer, 1992.

[131] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[132] Fei Wu, Xiao-Yuan Jing, Shiguang Shan, Wangmeng Zuo, and Jing-Yu Yang. Multiset feature learning for highly imbalanced data classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[133] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2018.

[134] Zhen Yang, Jacky Keung, Xiao Yu, Xiaodong Gu, Zhengyuan Wei, Xiaoxue Ma, and Miao Zhang. A multi-modal transformer-based code summarization approach for smart contracts. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 1–12, 2021.

[135] Zhen Yang, Jacky Keung, Miao Zhang, Yan Xiao, Yangyang Huang, and Tik Hui. Smart contracts vulnerability auditing with multi-semantics. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 892–901. IEEE, 2020.

[136] Zhen Yang, Jacky Wai Keung, Xiao Yu, Yan Xiao, Zhi Jin, and Jingyu Zhang. On the significance of category prediction for code-comment synchronization. *ACM Transactions on Software Engineering and Methodology*, 32(2):1–41, 2023.

[137] Xiao Yu, Jin Liu, Zijiang Yang, Xiangyang Jia, Qi Ling, and Sizhe Ye. Learning from imbalanced data for predicting the number of software defects. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 78–89, 2017.

[138] Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. Cpc: Automatically classifying and propagating natural language comments via program analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1359–1371, 2020.

[139] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1385–1397. IEEE, 2020.

[140] Xin Zhou, DongGyun Han, and David Lo. Assessing generalizability of codebert. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 425–436. IEEE, 2021.

[141] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing apis documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 27–37. IEEE, 2017.

[142] Jinhua Zhu, Yingce Xia, Lijun Wu, Di He, Tao Qin, Wengang Zhou, Houqiang Li, and Tie-Yan Liu. Incorporating BERT into neural machine translation. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[143] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 341–353, 2021.

[144] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. Smart contract vulnerability detection using graph neural networks. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pages 3283–3290, 2021.