Mucha: Multi-channel based Code Change Representation Learning for Commit Message Generation

Chuangwei Wang, Yifan Wu, and Xiaofang Zhang*
School of Computer Science and Technology, Soochow University, Suzhou, China
20215227033@stu.suda.edu.cn, 20214227066@stu.suda.edu.cn, xfzhang@suda.edu.cn
*corresponding author

Abstract—Commit messages provide a natural language description of the changes made to the code, enabling developers to swiftly comprehend the alterations without delving into the implementation complexities. Nevertheless, generating commit messages faces a considerable challenge due to the semantic and structural differences between code and natural language. Several researchers have put forward automated techniques aimed at generating commit messages. However, the full potential of code-related information is not yet fully harnessed. In this paper, we propose a Multi-channel based Code Change Representation Learning for Commit Message Generation(Mucha). We first compare the changed code and the corresponding AST before and after the change. Subsequently, we extract the altered information from various granularities and employ a multi-channel approach to capture the code changes, utilizing the extracted information as the basis for our analysis. In addition, we also use the query mechanism and attention mechanism to assist in learning the final code change representation. We build the experimental dataset, since there is still no publicly sufficient dataset for this task. The release of this dataset would serve as a valuable contribution towards advancing research in this particular field. We conduct a comprehensive experiment to assess the effectiveness of Mucha. The experimental evaluation demonstrates that our model outperforms the baseline model, which has significant improvements of at least 18.2%, 72.2%, and 10.5% against the baselines.

Keywords–code change; commit message generation; code representation learning

1. Introduction

Version control in software development usually requires developers to write a commit message for every commit manually. A commit message precisely describes the modification made in the code for a particular commit or the reasons behind those changes, often presented in natural language. Nonetheless, crafting a meaningful commit message demands considerable effort from the developer.

Consequently, researchers have proposed many approaches to automate generating commit messages. In the early stages of research, commit messages were primarily generated by applying pre-defined templates to the changed code [1], [2]. However, this template-based technique has limited suitability for scenarios where it is not generalizable and cannot understand the intent behind code changes. Later, some studies adopted retrieval-based techniques, which generate corresponding commit messages by searching for similar codes from known datasets and reusing their commit messages [3], [4], [5], [6], [7]. Although this approach enhances flexibility, its effectiveness heavily relies on the availability of similar code snippets in known datasets and the extent of their similarity.

With the promising results achieved by learning-based techniques in recent years [8], [9], [10], [11], [12], [13], [14], researchers have used them for commit message generation task. The most typical of these is the application of neural machine translation models (NMT). It regards commit message generation as a translation task achieved by translating the change code into natural language. Despite the impressive performance exhibited by NMT models, they encounter challenges with out-of-vocabulary (OOV) words.

Furthermore, there are general limitations in prior research on code representations: these methods either input the whole code before and after the change or just the code of the changed part. They do not fully utilize the change information of the changed code. For example, several studies [13], [15] employ the concatenation of flat tokens or AST paths from different change versions to represent code changes. By comparing the code representations of different change versions without explicitly highlighting fine-grained code changes, their coarse-grained representation can limit the analysis and understanding of code changes. In addition, the structure of the code contains rich syntactic information. In this paper, we exploit the change information of the code from multiple channels. We consider the semantic information of the code and use the structural information of the changed code.

Considering the limitations mentioned above and inspired by the success of pre-trained language models [16], [17], [18], [19], [20], [21], we propose a novel commit message approach in this paper, Mucha. We use multiple channels to build the semantics and syntax of the code. Moreover, we boost the understanding of the model on code-change-related tasks by initializing it with parameters provided by the pre-trained model. It can form a good bias when fine-tuning on commit message generation task to learn code change representations.

Specifically, given a code change, we first compare the code before and after the change, and record the alignment information between them. At the same time, we also extract the structure of the code change according to the AST of the

2693-9177/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS60937.2023.00064

code before and after the change, and record the information of the changed structure in the corresponding AST. Next, a pre-trained code model is applied to compute the contextual embedding of the changed code and the corresponding AST. Then, a multi-channel mechanism is used to capture information about the code changes from the contextual embedding. It will use the information recorded above to locate the changed code and part in the corresponding AST, and extract feature vectors from them to capture the change details. This feature vector can be used by attention as a query to retrieve relevant contextual information from the code before and after the change [21], and generate a final code change representation.

Meanwhile, we constructed an experimental dataset to assess the performance of Mucha as well as other state-of-the-art (SOTA) techniques. The results of the evaluation demonstrate that Mucha outperforms all the compared techniques. We then conducted an ablation study to analyze the effectiveness of Mucha, which also reveals the significance of the multi-channel approach.

The main contributions of this paper are:

- We propose a multi-channel technique for code changes, which can learn the changed code representation from multiple perspectives and select the necessary information from the changed code, especially for changes to AST.
- We propose a novel approach for automated commit message generation named Mucha, which consists of a pre-trained code model and multi-channel technique.
- We construct an experimental dataset for commit message generation, built from the original form of the changed code, namely the old and new versions.
- We evaluated Mucha and compared it with other techniques on our dataset, demonstrating the effectiveness of our approach through experimental analysis and an ablation study.

The remaining part of this paper is structured as follows: In Section 2, we provide the background knowledge of some techniques used in this study ans motivation for the study. In Section 3, we illustrate our approach. Then, we detail the experimental setup and experimental results in Sections 4 and 5 respectively. We discuss the threats to validity and related work in Sections 6 and 7. Finally, we conclude the paper in Section 8.

2. Background

In this section, we briefly introduce the background of the information presented, the AST used in our study, and the motivation that inspired us to conduct this study.

2.1. GitHub Commits

In version control, a commit usually contains the code before and after the change, and the corresponding natural language describing the code change, i.e., the commit message. In a commit, the changed code is marked with '+' or '-' ahead of the line, while the code that has not changed is unmarked and shown only once. The developers usually have to manually write commit messages for code changes so that others can quickly understand the code changes.

2.2. Abstract Syntax Tree

Abstract Syntax Tree (AST) is a tree designed to represent the abstract syntactic structure of source code [22]. Each node in AST represents different elements in the tree, which contains rich structural information [23], [24]. Such as, node type, node position and node label. It can provide all the details that cannot be found in the original code.

2.3. Pre-trained Code Model

The pre-trained model has been widely used for code representation, and it can build strong code representation from large-scale corpora. By using the initialization parameters provided by the pre-trained model, a good bias can be formed when fine-tuning on downstream tasks to learn code change representations. In addition, pre-trained models can prevent overfitting on small sample datasets. The pre-trained models, such as CodeBERT [25], CodeT5 [26] and UniXcoder [27], show significantly promising results on code-related tasks. These models are based on the Transformer architecture. The impressive performance of these models inspires us to integrate their advantages in the model.

2.4. Motivation of Multi-Channel

Researchers aim to generate a natural language description for a single code commit, focusing on describing the code changes using the changed code as the basis. Therefore, a model designed to generate natural language descriptions of code changes should be able to comprehend how the code is modified. By learning from the changes in the code, the model should be capable of automatically generating accurate descriptions of the changes.

```
1     public int add() {
2   -     int res = ans != null ? ans. add() : 0;
3   -     res = res * 12 + score. add();
4         return res;
5     }
```

(a) The before version of source code

```
1       public int add() {
2   +       int res = ans. add();
3   +       res = res * 22 + 2 * score. add();
4           return res;
5       }
```

(b) The after version of source code

Figure 1. An example of a code change

Figure 1 illustrates an example of a code change. Figure 1(a) displays the source code before the change, and Figure 1(b) displays the source code after the change. We can observe the changes in the code snippets by comparing these two versions

of the code. When observing at the line-level granularity, it is evident that the second and third lines have changed. However, upon considering a finer granularity level, we can observe that the second line change entails the removal of the variables 'ans' and the literal '$null$', i.e., the structure of the InfixEpression '$ans! = null$' is deleted. Furthermore, the method call '$ans.add()$' is relocated to the beginning, and the NumberLiteral '0' is deleted, leading to the deletion of the ConditionalExpression as well.

Regarding the change in the third line, the NumberLiteral '12' is updated to '22', and an InfixExpression is inserted into the structure, specifically '$2 * score.add()$'. This involves moving the method call '$score.add()$' and inserting the Infix_Expression_Operator '$*$' and NumberLiteral '2' in front of it. Thus, for this fine-grained change, we can directly obtain more fine-grained editorial changes from the structure of the AST.

However, existing techniques [11], [12], [13], [14], [15] obtain code representations by simply concatenating the old and new versions of the changed code without explicitly emphasizing fine-grained AST node changes for the commit message generation task. Through the comparison of representations from different code versions, models may can potentially identify fine-grained editing operations on their own, such as the deletion of '$res! = null$' or the movement of '$res.add()$'.

Therefore, by analyzing the alterations in the attributes of AST nodes, we can describe the code changes at a multi-grain level. In particular, when describing code modifications, we can extract more comprehensive information beyond mere additions and deletions of code lines. For instance, we can accurately describe the movement of a code line by observing changes in its 'position' attribute. Furthermore, we can pinpoint the specific updated part of a code line, such as alterations in the value of a NumberLiteral. This approach grants us access to a wealth of semantic information.

Finally, since AST nodes can be categorized into various levels [28], ranging from word-level to statement-level, this provides the model with the capability to comprehend code changes from different perspectives. By considering changes at these diverse levels, the model can gain a more comprehensive understanding of the code modifications, which has the potential to enhance its prediction performance. Therefore, our proposed method uses multi-channel techniques to better capture information at three levels of granularity (i.e., Line-level, Token-level, and AST-level).

3. Proposed Approach

In this section, we illustrate our approach, Mucha. The architecture of the Mucha is depicted in Figure 2, comprises three main components:

Data diff. Given a code change, split it into before and after change codes, i.e., C_b and C_a, and obtain the corresponding ASTs, i.e., A_b and A_a, respectively. Mucha compares the C_b and C_a sequences to find the modified codes and build code change C from the stored line alignment information. In addition, the A_b and A_a are compared to find the changed node and store their information in the corresponding AST.

Contextual Embedding. The C, A_b, and A_a in the data processing are fed through a pre-trained model to obtain the corresponding contextual embedding vectors.

Multi-channel. The vectors obtained from contextual embedding and the information stored in data processing are used as input to produce the final code change representation of C through multiple channels.

3.1. Data diff

In this paper, our model is designed to process changes in code at the method level. Thus, given a code change, we split it into the before change C_b and the after change code C_a. We use difflib [29], a package provided by Python, to perform code diff to obtain the code in which the change happened, including the new lines added in C_a, the lines deleted in C_b, and the lines unchanged. Meanwhile, we use GumTree [30] for C_b and C_a, a tool that can parse AST, to get the corresponding AST, i.e., A_b and A_a. Then, we will extract the corresponding change information from the three different granularities of changed code lines, token, and AST node changes, respectively.

Line Aligning. For the before and after changed code C_b and C_a, we first use difflib to perform line alignment and record their line alignment information. Specifically, for the same lines of code in C_b and C_a, i.e., lines of code that have not changed, we record them as keep and mark them with a special token $[KEEP]$ in front of the line. We only record such lines of code that have not changed once. For added lines of code in C_a that do not exist in C_b, we record them as add and mark them with a special token $[ADD]$ in front of the line. Similarly, for deleted lines of code that are in C_b and do not exist in C_a, we record them as delete and mark them with a special token $[DEL]$ in front of the line. Finally, we obtain the line-level code difference sequence through the above procedure, namely C, as shown in Figure 3. In addition, we use a flag to identify the changed line, where 1 indicates that the line changed and 0 indicates the line no changed.

Token Aligning. For the lines in C_b and C_a that have changed, we use difflib to perform token alignment and record their token alignment information. Specifically, after performing the line aligning, we identify the changed token for the changed lines, which we will identify with a flag, where 1 indicates that the token changed and 0 indicates the token no changed.

AST Aligning. After obtaining C_b and C_a from the code changes, we use the GumTree tool to parse them and obtain the corresponding ASTs, namely A_b and A_a. Meanwhile, we use the GumTree tool to compare the differences between the ASTs of C_b and C_a and locate the changes in the AST nodes. For the difference comparison of AST nodes, there are five different types: Match, Insert, Delete, Move, and Update, as shown in Figure 4.

- Insert: As illustrated by the orange node in Figure 4(b). The code node, such as $NumberLiteral$ and

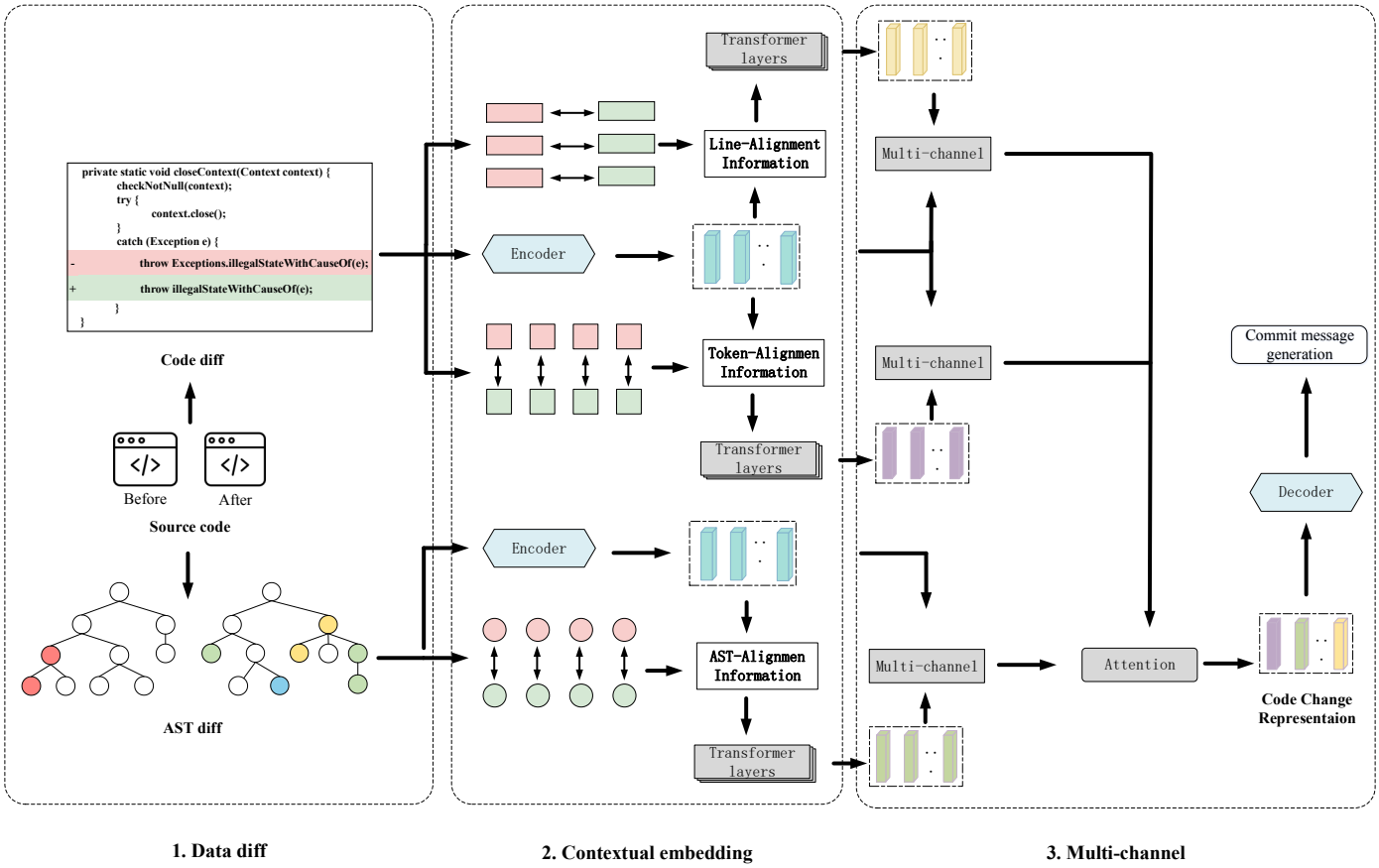

Figure 2. Overview of model

```
[KEEP]   public int add() {
[DEL]    int res = ans != null ? ans.add() : 0;
[ADD]    int res = ans.add();
[DEL]    res = res * 12 + score.add();
[ADD]    res = res * 22 + 2 * score.add();
[KEEP]   return res;
[KEEP]   }
```

Figure 3. Code_Diff part of a real Java code input before and after the change.

$InfixExpression$, does not exist in AST_{old} but exists in AST_{new}. The changed new node is inserted in AST_{new}.

- Delete: This changed code node exists in AST_{old} but not in AST_{new}, such as $NumberLiteral$ and $InfixExpression$. As indicated by the red node in Figure 4(a).
- Move: The code node $MethodInvocation$ exists in both AST_{old} and AST_{new}, but the position of the AST node in AST_{old} is changed in AST_{new}. Generally, only statement-level AST nodes are modified, as indicated by the green node in Figure 4(b), which do not include its child nodes.
- Update: This type of changed code node occurs in both AST_{old} and AST_{new} and generally indicates that part of a statement or expression has changed (for example, a change of variable name). As indicated by the blue node in Figure 4(b)(i.e., NumberLiteral).

In this paper, we select only the changed nodes among them, i.e., discard the Match type. We align the changed nodes to the corresponding nodes in A_b and A_a by node position and type information. We also use the same flags as above to identify the changed nodes in AST.

3.2. Contextual Embedding

In this paper, we integrate pre-trained models to build strong code representations. Therefore, we use the tokenizer used in the pre-trained model to tokenize the change code C and the corresponding AST. Our model has three encoders, each with 12 transformer encoder layers and 12 decoder layers. We integrate the two encoders about AST into one encoder, as shown in Figure 2. We initialize Mucha with the parameters of UniXcoder. UniXcoder utilizes mask attention matrices to enhance code representation with cross-modal content like AST and code comments. It is widely used for code-related tasks. We then take the change code C, and the corresponding AST as input and obtain the corresponding contextual embedding vectors by encoding it. For the change code C after encoding is denoted as $H^C = [h_1, h_2, ...h_{|C|}]$. Moreover, the corresponding ASTs for the change codes,

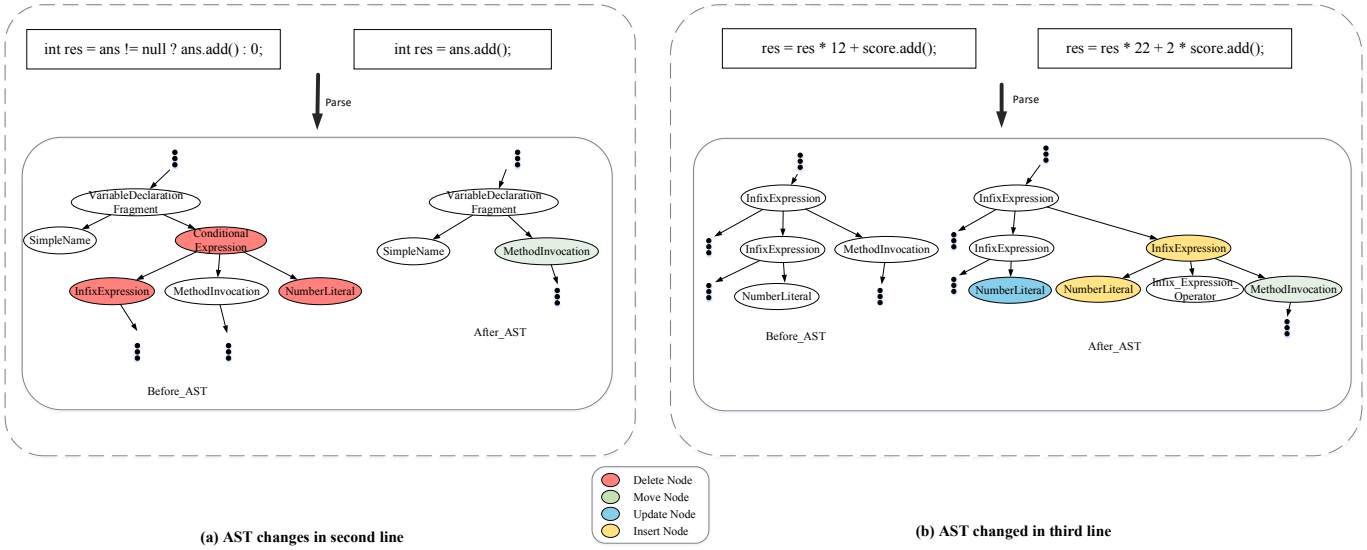

Figure 4. A real world AST_Diff part of a Java change before and after code input. Due to space constraints, we only show part of the AST_ Diff sequence.

i.e., A_b and A_a, are denoted as $H_b^A = [h_1, h_2, ...h_{|A_b|}]$ and $H_a^A = [h_1, h_2, ...h_{|A_a|}]$, respectively. We then use these three levels of alignment information to get the change information from the contextual embedding. We capture the semantic and syntactic information of the change code at three different granularities.

3.3. Multi-Channel

This part produces the final representation vector for the given change code. In this paper, we follow the query back mechanism used in previous work [21] and combine it with our proposed multi-channel to help Mucha better capture code change information. We use the changed code part as a query q by the query back mechanism and then q to get important information from the changed code C. Finally, we combine the attention mechanism to output the final vector.

Specifically, this paper combines the contextual embeddings H^C, H_b^A, and H_a^A generated by the pre-trained model with the alignment information obtained in the data diff to generate the final changed code representations. We describe the combination of these three channels in detail: Line-level, Token-level, and AST-level.

Line-Level Channel. After getting the change codes C_b and C_a in data diff, we align them and identify the changed lines with flags. We then follow the query back mechanism in previous work [21]. Based on these flags, we extract the changed information from the contextual embedding (H^C) generated by the pre-trained model. We combine the changed information with the query back mechanism to obtain the final change code representation v_l^c.

Token-Level Channel. Similarly, after we do the line alignment in data processing, we align the token to the changed lines and mark the changed token with a flag. We then combine the query back mechanism with the changed information extracted from H^C based on these flags to get the final change code representation v_t^c.

AST-Level Channel. In data processing, we obtain the changed nodes by comparing A_b and A_a and use flags to identify the nodes that changed in A_b or A_a. We also use the query back mechanism and combine the change information, which is extracted from H_b^A and H_a^A based on these flags, to generate the final change code representations v_b^a and v_a^a.

Then, we linearly project them into the same feature space, normalize them with layer normalization, and merge them to generate the final change code representation v^h.

$$v^h = W_l^c v_l^c + W_t^c v_t^c + W_a^a v_a^a \tag{1}$$

where W_l^c, W_t^c and W_a^a are learnable parameters of this module.

4. Experimental Design

4.1. Datasets

In this paper, we extract change information at different granularities (i.e., Line-level, Token-level, and AST-level) for change codes. These include constructing ASTs on the changed code and extracting the changed AST nodes from them to obtain the structural information of the differences. Our model is designed to process changes in code at the method level. However, the previous datasets on the commit message generation task either focus on the changed part without involving the changed context codes or store the old and new versions serialized in diff formats. In addition, A code diff usually contains one or more hunks. We present the dataset in the experiment as follows from dataset collection and construction.

Dataset Collection. We obtain the dataset required for our experiments by constructing their open-source dataset provided by Tufano et al. [31], which is at the method level.

Thus, we guide the evaluation experiments on the dataset released based on the previous work. The original dataset is collected from popular Java projects on GitHub.

The dataset contains 167k triples $<m_s, c_{nl}, m_r>$ of changed code and corresponding code reviews after their pre-processing (e.g., filtering out noisy comments), where m_s is a method submitted for the review; c_{nl} is a suggested comment from the reviewer for code changes to m_s; m_r is the revised version of m_s that incorporates the recommendations provided by the reviewers denoted as c_{nl}. Our work is different from previous work, which only focuses on the suggestions given by the reviewers(m_s) during the code review process. In contrast, our work concentrates on the commit message before submitting the code review.

Dataset Contruction. We construct our dataset in triplets $<c_b, msg, c_a>$ form, where c_b is the code before the change; c_a is the code after the change; and msg is the change information describing the code. Then, since the text sequences in open-source dataset [31] were already pre-processed and contains special tokens such as $<START>$, $<END>$ and $<technical\_language>$ to indicate the msg and its corresponding codes, we automatically transformed them into our code format with regular expressions.

To ensure that our processed dataset is clear, we design a script to validate these datasets, which have been transformed into a code format, by discarding data that cannot be built as an AST. Finally, we discarded 3.2% of the dataset with the designed scripts.

4.2. Research Questions

For commit message generation, the primary challenge is ensuring that the generated change messages, which are in natural language, effectively represent the corresponding code changes. To investigate whether multi-channel with the different granularity of Mucha outperforms baselines and the impact of each channel, we design the following research questions (RQs):

[RQ1:] How effective is Mucha compared with the SOTA baselines on commit message generation?

[RQ2:] What role does each component play in Mucha?

[RQ3:] What is the impact of experimental parameters on the performance of our approach?

In RQ2, we explore the impact of each channel on the model's performance by removing them one by one and evaluating the model's results. This analysis will help us understand the individual contributions of each channel to Mucha's effectiveness in generating commit messages. For the impact of the pre-trained code model, we use different pre-trained models to investigate their contribution to Mucha's effectiveness.

4.3. Baseline Models

In this section, we introduce the baselines utilized in our experiments. A brief description of each baseline is provided below:

- **CODISUM** [11]: CODISUM is an NMT-based approach aimed at generating commit messages through learning techniques. It leverages both the code structure and code semantics to produce more accurate and informative commit messages. Moreover, CODISUM employs the copy mechanism to mitigates the OOV problem.
- **CoreGen** [32]: CoreGen presents a two-stage framework for commit message generation, building upon the Transformer model. They construct the code semantics using contextualized code information.
- **CCRep** [21]: CCRep is a code change representation approach that consists of a pre-trained model and the query-back mechanism. It mainly uses a query mechanism at the line level and token level for changed code, and applies code change related downstream tasks. In this paper, we take the model structure from their work and process the data to the appropriate line-level and token-level for input.
- **CodeBERT** [25]: CodeBERT is a bimodal pre-trained model for programming and natural language. They pre-trained with two pre-trained objectives. In this paper, we use the CodeBERT encoder-decoder architecture, where the encoder and decoder layer parameter settings follow those from their work, and the input is the code before and after the change.
- **UniXcoder** [27]: UniXcoder employs mask attention matrices to exert control over the model's behavior and enhances the code representation by incorporating cross-modal content like AST and code comments, making model becomes proficient in supporting code-related understanding and generation tasks.

To ensure the fairness and stability of the experiments, we followed the parameter settings in their experiment and used our dataset for it. We conducted all methods on our dataset and repeated each experiment 10 times.

4.4. Evaluation Metrics

To measure the quality of the generated commit message, we use BLEU [33], METEOR [34], and ROUGE-L [35] as metrics in the experiment. All these metrics score from 0 to 100, where 0 means a no match and 100 means a perfect match.

BLEU uses an n-gram matching rule to compare overlapping n-grams. It is used for evaluating the quality of generated comments and is considered an accurate metric. In this paper, we use the BLEU-4 in our experiments. The score is calculated as:

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{2}$$

where p_n refers to the ratio of identical subsequences with length n in the reference. w_n are positive weights that sum to 1. In this paper, we set N to 4 and w_n to 1/4. BP is brevity penalty:

$$BP = \begin{cases} 1 & c > 0 \\ e^{(1-r/c)} & c \leq 0 \end{cases} \tag{3}$$

where c is the length of the generated and r is the actual length of the reference.

ROUGE-L evaluates the similarity between the generated and the reference. It computes the F-score based on the longest common sub-sequence to evaluate the generated text. The score is calculated as follows:

$$ROUGE-L = \frac{\left(1+\beta^2\right)R_{lcs}P_{lcs}}{R_{lcs}+\beta^2 P_{lcs}} \quad (4)$$

R_{lcs} and P_{lcs} represent the recall rate and accuracy rate respectively. β is set to a very big number. Therefore, only R_{lcs} is considered.

$$R_{lcs} = \frac{LCS(X,Y)}{m} \quad (5)$$

$$P_{lcs} = \frac{LCS(X,Y)}{n} \quad (6)$$

where $LCS\,(X,Y)$ refers the length of the longest common subsequence between X and Y. m and n respectively represent the length of the reference text and the generated text.

METEOR is a recall-oriented metric and calculates the harmonic mean of accuracy and recall between the generated and reference text. The score is calculated as:

$$METEOR = (1 - Pen)\, F_{mean} \quad (7)$$

where Pen is calculated according to the number of chunks (ch) and the number of matches (m):

$$Pen = \gamma * \left(\frac{ch}{m} \right)^{\beta} \quad (8)$$

The values of β and γ parameters are set to 0.20 and 0.60, respectively, following the prior work [34].

4.5. Experimental Setting

In our experiments, we use GumTree tools to parse ATS for before and after changes on the Java code and compare ASTs. We implement our model using HuggingFace's Transformer python package and the deep learning framework PyTorch. We train the model on a server with 24 cores of 3.8GHz CPU and an NVIDIA GeForce RTX 3090GPU. The coefficient λ of 2 regularization item was set to 10e-5. During the training phase, we utilize the Adam optimizer with a learning rate of 5e-5 and linear warmup.

5. Results Analysis

In this section, we show the results of our experiments. We first show the performance of our model on the commit message generation task and its results compared to baselines. Additionally, we investigate each channel's individual impact and the pre-trained model's initial parameters on Mucha's performance. Due to hardware resource constraints, we analyze the impact of different experimental parameters on our model.

5.1. Answering Research Question1

[RQ1:] How effective is Mucha compared with the SOTA baselines on commit message generation?

To evaluate the quality of the commit messages generated by the model, we measure the difference between the generated messages and the reference sequences using the metrics mentioned above (i.e., BLEU, METEOR, and ROUGE-L). The results presented in Table I demonstrate the effectiveness of Mucha for commit message generation on the java method. our proposed model outperforms the compared models in metrics such as precision (measured by BLEU, ROUGE-L) and recall (measured by ROUGE-L, METEOR). The significant improvements of at least 18.2%, 72.2%, and 10.5% against the baselines on the experimental dataset. For the decreased ROUGE-L scores, we consider that this is also relevant to the model's parameters. For this reason, we further discuss the impact of the model parameters in RQ3.

TABLE I
METRICS EVALUATION RESULTS FOR THE BASELINES AND MUCHA

Methods	BLEU	METEOR	ROUGE-L
CODISUM	3.56	1.30	5.64
CoreGen	6.72	2.51	6.59
CCRep	3.50	4.80	2.65
CodeBert	6.26	2.39	6.42
UniXcoder	9.62	4.46	9.46
Mucha	11.37	8.27	7.28

Although our models outperform the compared baseline model on the experimental dataset, the results on these three metrics (BLEU, METEOR, and ROUGE-L) are still relatively low. This also indicates that it is challenging to generate change information for change codes, which are generally diverse and not unique. This is because the commit message generated by the model may be semantically similar to the ground truth, but the form or word it presents is very different.

5.2. Answering Research Question2

[RQ2:] What role does each component play in Mucha?

This RQ is designed to explore the impact of each component on Mucha's performance. We perform the ablation study by removing each/all channels and initializing the parameters with different pre-trained models. Different from the pre-trained model parameters of UniXcoder used in this paper's approach, in the ablation study, we use CodeBERT pre-trained model parameters for initialization (i.e., Mucha-CodeBERT). The experimental results are shown in Table II.

TABLE II
ABLATION STUDIES OF EACH CHANNEL AND PRE-TRAINED CODE MODEL.

Methods	BLEU	METEOR	ROUGE-L
Mucha	11.37	8.27	7.28
-w/o AST	9.82	7.70	5.32
-w/o Line	10.58	7.85	6.44
-w/o Token	10.14	7.91	5.74
Mucha-CodeBERT	4.75	4.95	3.63

The first observation we can make is that the model performance is degrading when removing channels or initializing

other pre-trained model parameters. When we remove one of these three channels, the performance of the model decreases. Among them, the drop effect is most significant when the AST channel is removed. This also shows that learning the changed structure information of AST can be helpful in bridging the gap between code changes and natural language. While removing Line or Token channel does not have as much impact on model performance as removing AST channel, both channels have a higher impact on model performance than the model without channel. The results also show that the performance of the model is improved by integrating the two channels.

In addition, we can observe that when we initialize Mucha with different pre-trained model parameters, the performance of the model varies. We set the initial parameters of UniXcoder for Mucha, which has better results in the three metrics. Choosing the suitable pre-training model parameters has a positive effect on the model improvement.

By ablation study, we can find that different channels have different degrees of influence on the model, and appropriate model initialization are helpful in improving the model performance.

5.3. Answering Research Question3

[RQ3:] What is the impact of experimental parameters on the performance of our approach?

Due to hardware resource limitations, we evaluate the experimental parameters' impact on our method and the model's sensitivity to some parameters. We explore the impact of different parameter settings on model performance by varying the learning rate, and input length for Java methods in our experiments. Figure 5 and 6 shows the model's metric evaluation results with different parameter settings on our experimental dataset.

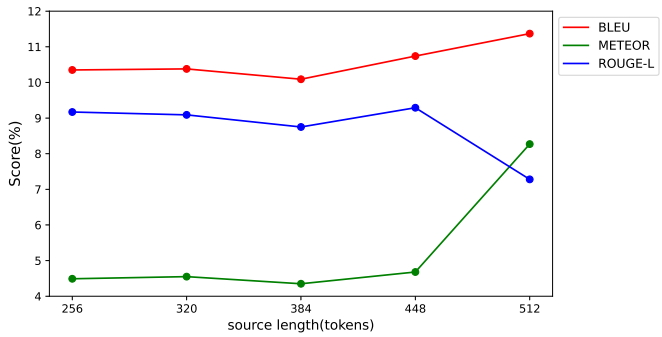

Figure 5. The evaluation scores of the models at different input lengths

Figure 5 shows the performance of Mucha for different input lengths. We can first observe a general trend that the score of the model under the metrics increases with the input length. Then, when the input length comes to 512, the model's metrics scores change relatively dramatically. Although ROUGE-L's score has decreased, the change is not floated a lot. This may be related to the dataset, where the parts of the dataset that have changed may be somewhat similar. The reason is that as the input increases, the information learned by the model increases accordingly, making the text generated by the model on the test dataset very similar to the reference text, while semantically different.

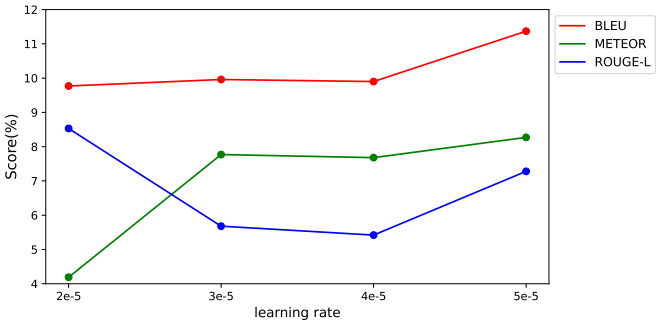

Figure 6. The evaluation scores of the models at different learning rates

Figure 6 shows the performance of Mucha for different learning rates. We can observe that different learning rates make the model have different scores. Choosing the appropriate learning rate also helps the model's performance.

In conclusion, different model parameters have different effects on the experiments. However, the overall performance fluctuation of the model is still relatively stable. In this paper, we set the source length and learning rate to take the values of 512 and 5e-5, respectively.

6. Threats to Validity

In this section, we discuss the threats to the validity of the approach as follows:

External validity. Threats to external validity in this study primarily stem from the dataset collection and utilization process. The dataset we collected is implemented in Java, which is collected from the dataset of Tufano et al [31], by adapting it to obtain the corresponding change descriptions. The findings and conclusions drawn from this study might not directly apply to projects implemented in different programming languages, regardless of whether they are open-source. In the future, we will explore more diverse projects and incorporate various programming languages to enhance our understanding of what constitutes a high-quality message and reduce this threat.

Internal validity. In our method, the factors most likely to affect the internal validity are the implementation of compared techniques and the configuration of the experimental environment. To mitigate the risk of compared techniques implementation, we implement them directly from their reproducible open-source packages. If their package is not available or executable, we reproduce them as described in the corresponding papers.

Construct validity. In our study, the threat to construct validity is the evaluation metrics. To reduce the threat introduced by metrics, we adopt three evaluation schemes: BLEU, METEOR, and ROUGE. However, we consider that using these automated metrics to evaluate models may not represent human evaluation. There may be cases where a

model produces a grammatically and semantically valid text, but it does not match the reference text generated by humans. Therefore, there is no evaluation of whether a reference text is suitable for a given method. So, in the future, we will further perform a human evaluation of the model to mitigate the pitfalls involved in automated metrics and help us interpret the effectiveness of our model through experimental results.

7. Related works

7.1. Commit Message Generation

The existing work on commit message generation can be categorized as template-based, retrieval-based, and learning-based.

Template-based. Researchers generate commit messages by analyzing the code changes and employing pre-defined templates to construct the messages. For instance, Buse and Weimer [36] use path prediction to design a templates for changed code. On the other hand, Shen et al. [1] design templates based on the method stereotypes and the type of code changes, offering insights not only into what has been altered but also the rationale behind those changes.

Retrieval-based. Later work uses information retrieval techniques by retrieving similar codes from the training set and thus reusing their commit messages. For example, Liu et al. [3] propose to generate concise commit messages using the nearest neighbor algorithm. From the training set, they use cosine similarity and BLEU as metrics to select the most similar code changes. Similarly, Huang et al. [4] use syntactic and semantic similarity to find code changes.

Learning-based. Recently, researchers have been exploring the use of deep learning techniques for commit message generation. Jiang et al. [8] and Loyola et al. [9] employ NMT models directly for translation purposes. They establish a connection between code changes and natural language by using NMT to translate code changes into human-readable commit messages. Building upon this foundation, Later work by Loyola et al. [10] further enhance the commit message generation quality by leveraging the contextual information of the code changes. allows their model to improve the generation quality. Nie [32] propose a novel approach to learn contextual code representations by leveraging contextual information. These learned representations are subsequently utilized to fine-tune the Transformer model specifically for commit message generation.

In this paper, our approach goes beyond solely considering the changed code's semantic information. Instead, we also consider the structural information of the changed code at multiple granularities to achieve a more comprehensive understanding of the code changes.

7.2. Code Representation Learning

Deep learning has become increasingly prevalent in software engineering, leading to a growing interest in extracting semantic features directly from source code. By employing deep learning techniques, this approach enables the model to acquire semantic information directly from the code, leading to substantial enhancements in its performance. Besides, several prior studies have shown that semantic features can capture much information and significantly improve performance [37], [38], [39].

Moreover, owing to the remarkable achievements of pre-training methods in recent years [16], [17], [18], [19], researchers have attempted to apply these pre-training techniques to programming languages, aiming to enhance the advancement of code intelligence. Kanade et al. [40] adopt two key objectives, namely masked language modeling and next sentence prediction, as part of their pre-training process for CuBERT, a model specifically designed for Python code. CodeBERT [25], a bi-directional transformer model pre-trained on NL-PL pairs in six programming languages, excels at learning code representations. GraphCodeBERT [41] considers code data flow based on CodeBERT and designs a pre-training task for code data flow. Encoder-decoder models, like PLBART [15] and CodeT5 [26], as well as UniXcoder [27], are employed for both comprehension and generation tasks. These models take either source code or textual input as their input data. On the other hand, CodeReviewer [42] follows a different approach by utilizing the diff format of the code. It leverages four pre-training tasks to learn code changes and effectively applies them to automate code review activities. While CCRep [21] learns the representation of the change code by a pre-trained model as well as the query mechanism.

In this paper, We consider the importance of pre-trained models, and we integrate pre-trained models and multi-channel to learn the representation of changed code from multiple levels.

8. Conclusion and Future Work

In this paper, we present Mucha, a multi-channel based code change representation learning for generating commit messages. Additionally, we build a dataset for commit message generation to conduct our experimental evaluation.

1) A new input representation addresses the inadequate utilization of code change information. This new representation incorporates code change details from three different levels of channels, effectively leveraging a multi-channel approach. 2) The experimental results show that Mucha outperforms the compared models. Furthermore, the ablation experiments provide valuable insights into the effectiveness of each channel in Mucha and highlight the significance of the pre-trained model in contributing to the overall performance improvement.

In the future,we plan to conduct further evaluations of the proposed model on larger-scale datasets encompassing various programming languages. Moreover, we will explore and experiment with other models that can capture a more comprehensive range of change information in the changed code, and perform a human evaluation of the model to interpret the effectiveness of our model through experimental results. Our code and data are publicly available at https://github.com/cmgads/Mucha.

ACKNOWLEDGMENT

This work is partially supported by the National Natural Science Foundation of China (62172202), Collaborative Innovation Center of Novel Software Technology and Industrialization, the Major Program of the Natural Science Foundation of Jiangsu Higher Education Institutions of China under Grant Nos. 22KJA520008 and the Priority Academic Program Development of Jiangsu Higher Education Institutions.

REFERENCES

[1] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, D. Poshyvanyk, On automatically generating commit messages via summarization of source code changes, in: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, IEEE, 2014, pp. 275–284.

[2] J. Shen, X. Sun, B. Li, H. Yang, J. Hu, On automatic summarization of what and why information in source code changes, in: 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), Vol. 1, IEEE, 2016, pp. 103–112.

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

[4] Y. Huang, Q. Zheng, X. Chen, Y. Xiong, Z. Liu, X. Luo, Mining version control system for automatically generating commit comment, in: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2017, pp. 414–423.

[5] Y. Huang, N. Jia, H.-J. Zhou, X.-P. Chen, Z.-B. Zheng, M.-D. Tang, Learning human-written commit messages to document code changes, Journal of Computer Science and Technology 35 (6) (2020) 1258–1277.

[6] Z. Li, Y. Cheng, H. Yang, L. Kuang, L. Zhang, Retrieve-guided commit message generation with semantic similarity and disparity, in: 2022 29th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2022, pp. 357–366.

[7] E. Shi, Y. Wang, W. Tao, L. Du, H. Zhang, S. Han, D. Zhang, H. Sun, Race: Retrieval-augmented commit message generation, in: Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, 2022, pp. 5520–5530.

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

[9] P. Loyola, E. Marrese-Taylor, Y. Matsuo, A neural architecture for generating natural language descriptions from source code changes, arXiv preprint arXiv:1704.04856.

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

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

[12] Q. Liu, Z. Liu, H. Zhu, H. Fan, B. Du, Y. Qian, Generating commit messages from diffs using pointer-generator network, in: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), IEEE, 2019, pp. 299–309.

[13] S. Liu, C. Gao, S. Chen, N. L. Yiu, Y. Liu, Atom: Commit message generation based on abstract syntax tree and hybrid ranking, IEEE Transactions on Software Engineering.

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

[15] W. U. Ahmad, S. Chakraborty, B. Ray, K.-W. Chang, Unified pre-training for program understanding and generation, arXiv preprint arXiv:2103.06333.

[16] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., Language models are few-shot learners, Advances in neural information processing systems 33 (2020) 1877–1901.

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

[18] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, L. Zettlemoyer, Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, arXiv preprint arXiv:1910.13461.

[19] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu, et al., Exploring the limits of transfer learning with a unified text-to-text transformer., J. Mach. Learn. Res. 21 (140) (2020) 1–67.

[20] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, Combining code context and fine-grained code difference for commit message generation, in: Proceedings of the 13th Asia-Pacific Symposium on Internetware, 2022, pp. 242–251.

[21] Z. Liu, Z. Tang, X. Xia, X. Yang, Ccrep: Learning code change representations via pre-trained code model and query back, arXiv preprint arXiv:2302.03924.

[22] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272), IEEE, 1998, pp. 368–377.

[23] Y. Wang, H. Li, Code completion by modeling flattened abstract syntax trees as graphs, in: Proceedings of the

AAAI Conference on Artificial Intelligence, Vol. 35, 2021, pp. 14015–14023.
[24] J. Zhang, X. Wang, H. Zhang, H. Sun, X. Liu, Retrieval-based neural source code summarization, in: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), IEEE, 2020, pp. 1385–1397.
[25] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codebert: A pre-trained model for programming and natural languages, arXiv preprint arXiv:2002.08155.
[26] Y. Wang, W. Wang, S. Joty, S. C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, arXiv preprint arXiv:2109.00859.
[27] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, J. Yin, Unixcoder: Unified cross-modal pre-training for code representation, arXiv preprint arXiv:2203.03850.
[28] M. Cvitkovic, B. Singh, A. Anandkumar, Open vocabulary learning on source code with a graph-structured cache, in: International Conference on Machine Learning, PMLR, 2019, pp. 1475–1485.
[29] https://docs.python.org/3/library/difflib.html.
[30] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Monperrus, Fine-grained and accurate source code differencing, in: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp. 313–324.
[31] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, G. Bavota, Towards automating code review activities, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, 2021, pp. 163–174.
[32] L. Y. Nie, C. Gao, Z. Zhong, W. Lam, Y. Liu, Z. Xu, Coregen: Contextualized code representation learning for commit message generation, Neurocomputing 459 (2021) 97–107.
[33] K. Papineni, S. Roukos, T. Ward, W.-J. Zhu, Bleu: a method for automatic evaluation of machine translation, in: Proceedings of the 40th annual meeting of the Association for Computational Linguistics, 2002, pp. 311–318.
[34] M. Denkowski, A. Lavie, Meteor universal: Language specific translation evaluation for any target language, in: Proceedings of the ninth workshop on statistical machine translation, 2014, pp. 376–380.
[35] C.-Y. Lin, Rouge: A package for automatic evaluation of summaries, in: Text summarization branches out, 2004, pp. 74–81.
[36] R. P. Buse, W. R. Weimer, Automatically documenting program changes, in: Proceedings of the IEEE/ACM international conference on Automated software engineering, 2010, pp. 33–42.
[37] S. Wang, T. Liu, L. Tan, Automatically learning semantic features for defect prediction, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE, 2016, pp. 297–308.
[38] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, L. Zhang, A grammar-based structural cnn decoder for code generation, in: Proceedings of the AAAI conference on artificial intelligence, Vol. 33, 2019, pp. 7055–7062.
[39] H. Liang, Y. Yu, L. Jiang, Z. Xie, Seml: A semantic lstm model for software defect prediction, IEEE Access 7 (2019) 83812–83824.
[40] A. Kanade, P. Maniatis, G. Balakrishnan, K. Shi, Learning and evaluating contextual embedding of source code, in: International Conference on Machine Learning, PMLR, 2020, pp. 5110–5121.
[41] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al., Graphcodebert: Pre-training code representations with data flow, arXiv preprint arXiv:2009.08366.
[42] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu, et al., Codereviewer: Pre-training for automating code review activities, arXiv preprint arXiv:2203.09095.