Exploring the Utilization of Program Semantics in Extreme Code Summarization: An Experimental Study Based on Acceptability Evaluation

Jiuli Li, Yan Liu*
School of Software Engineering, Tongji University, Shanghai, China

***Abstract*—With the rise of deep learning methods, neural network architecture adopted from neural machine translation has been widely studied in code summarization by learning the sequential content of code. Given the inherent nature of programming languages, learning the representation of source code from the parsed structural information is also a typical way for constructing code summarization models. Recent studies show that the overall performance of the neural models for code summarization can be improved by utilizing sequential and structural information in a hybrid manner. However, both of these two kinds of information fed to the neural models for code summarization fail to embrace the semantics of source code snippets in an explicit way. Is it really a good way to just leave the semantics as hidden things in the source code and have the neural models capture whatever they can get? To observe the utilization of program semantics in automatic code summarization, we conducted an experimental study by analyzing the acceptability of the extreme code summaries generated from neural models. To make the models aligned in the same context for this experimental study and to focus on the observation of the semantics, we re-implement the neural models from three selected studies as extreme code summarization solutions. After an intuitive observation and exploration of the generated summaries with the models trained from a Java dataset, we identify five acceptability aspects: (1) function name format; (2) function naming style; (3) semantic level similarity; (4) the differences in hitting rate of representative words; and (5) the correlation between extreme code summaries with function body. Based on the false negative and false positive phenomena in the results, ablation experiments have shown that the use of program semantics has a positive effect on generating high-quality abstracts in neural models. Our work proves the potential of utilizing the program semantics explicitly in code summarization, and the possible directions are also indicated.**

Keywords*—*Extreme code summarization; program semantics utilization; acceptability analysis of code summary

I. INTRODUCTION

The task of code summarization refers to the automatically creating readable summaries describing the function of the given code snippets, and identify the roles and responsibilities of software units [1]. A good summary can help developers understand, reuse and maintain code more easily, and greatly improve production efficiency. However, problems exist in code summaries, including missing information, errors, and outdated comments. Human-written summaries also require professional domain knowledge, making the entire process time-consuming. Hence, machine-generated summaries are gaining popularity, with their effectiveness acknowledged in many studies.

The majority of automatic code summarization algorithms rely on techniques such as information retrieval, stereotype identification, machine learning and artificial neural network, and natural language processing [2] [3]. Among them, deep learning techniques have demonstrated the benefits of modeling programs recently [4] [5]. Specifically, guided by neural machine translation, early code summarization models focus on the sequential content of code [6]. Yet, leading approaches have recognized the significance of integrating structural information derived from Abstract Syntax Trees (ASTs).

However, both traditional and deep learning techniques have limitations in generating natural language summaries. Traditional approaches struggle with extracting keywords when identifiers and methods are poorly named, and proper summaries cannot be generated if similar code snippets are absent. Moreover, the majority of deep learning-based approaches treat the source code as plain text, resulting in the omission of crucial information, such as naming conventions for identifiers and usage patterns of application programming interfaces [7] [8]. Since sequences of tokens parsed from AST are typically fed into the sequence-to-sequence framework, this approach may fail to capture long dependencies between code tokens [9]. These limitations may lead to the underutilization of program semantics at both the code text level and structural level, as evaluated using the acceptability of generated code summaries. However, there are currently no systematic studies to address this issue. To assess the acceptability of code summaries generated by neural models, we selected representative models from various categories for extreme code summarization tasks, and intend to get insights from the experimental results. The main contributions of our study are as follows:

- To explore the acceptability of the code summaries generated from neural models, we re-implement the neural models from three selected studies for extreme code summarization. Following an intuitive observation of the generated summaries, we proposed five acceptability aspects for further analysis.

*Corresponding Author.

- To identify which limitations of the selected models aggravate to the lower acceptability, we conducted a comprehensive analysis, focusing on the misjudgment in generated summaries. We found that false negatives in extreme code summaries can be attributed to issues, such as text-level semantic similarity in code, variations in function hit rates, and the correlation between function names and their respective bodies. Besides, the format and naming conventions of function names may result in false positives in extreme code summaries. In accordance with these observations, further hypotheses are formulated to improve automatic code summarization, including from the perspective of underutilization of function body semantics by neural models and potential issues related to dataset preprocessing.
- To verify our hypothesis, we conducted the ablation experiments based on the selected models. We discovered that phrases with similar semantics have a greater impact on false negatives in generated summaries, while the format of function names has a stronger influence on false positives in the results. Subsequently, we provide directions for improvement in three aspects: dataset preprocessing, external data source and the model's learning process. These directions serve as a valuable reference for future research in the field.

II. Related Work

A. Overview of Common Models in the Field of Code Summarization

At present, several representative neural models which can be used to perform the task of code summarization in relevant field, including CODE-NN [10] model based on attention mechanism, Deep-Com [11] model based on code structure analysis, summary generation model based on reinforcement training and so on. Several classic code summarization models are as follows.

- CODE-NN is an end-to-end summary generation system built directly by using the structure of circular neural network, and relevant summary are generated according to the word vectors of source code. The introduction of attention mechanism not only highlights the contribution of key words in the decoding process, but also solves the problem that the summary generated by long code is difficult to understand.
- The code summarization model based on sequence-to-sequence learning algorithm [12] is also popular. The encoder and decoder of this model are built by independent LSTM neural networks, which can extract lexical features of source code and generate summaries. It inputs the key vocabulary sequence of the source code function and outputs the English summary related to the function.
- Deep-Com [11] based on code structure analysis is also a mainstream model in this field. To extract the hidden structural information in the source code, Deep-Com firstly outputs the summary syntax tree as a sequence of nodes in a specific order through a special traversal algorithm [13], and then generates the summary of the target code by using the classic encoder-decoder model. The author thinks that the traversal algorithm used by Deep-Com can express the structural characteristics of the summary syntax tree without loss, and the generated summary can also accurately describe the functional characteristics of the source code.
- The reinforcement learning model for parameter training based on actor-critic mode recently proposed by wan et al gradually becoming popular [14]. Different from the common code summarization model in the field, the author innovatively uses reinforcement learning to update the model parameters, which can further reduce the exposure bias.

In addition, there are also several neural models that can be used directly to perform the task of extreme code summarization, such as Code2Vec, Code2seq, Code-Transformer are shown below:

- Code2Vec [15], which transforms code fragments into vectors with fixed length and continuous distribution, which can be used to predict the semantic information of code fragments. To achieve this goal, Code2Vec is first decomposed into a set of paths in its corresponding AST, and then the neural network is used to learn the representation of each path and how to integrate the representations of all paths. The effectiveness of Code2Vec has been verified by the task of predicting the function name with vector representation of function body.
- Code2seq [16], which uses the syntax structure in programming language to encode the source code. In this model, a part of paths are extracted from AST of code fragments, and the target sequence is generated by Attention after LSTM coding. Code2seq uses the way of encoding the sample of code fragment AST to extract grammatical information better. The effectiveness of Code2seq has been verified in the extreme code summarization task.
- Code-Transformer [17], which jointly learns the sequential and structural information in source code. Compared with other neural models, it only depends on language-independent features, and can directly calculate the source code and features from AST. The performance of the Code-Transformer model is also validated on the task of predicting function name based on function body.

Although the above models have good performance in code summarization generation, due to the lack of learning about structural information or semantic information of source code, sometimes it is inevitable that the generated summaries are difficult to understand or have poor readability.

B. Performance Analysis of Code Summarization Model

The code summarization algorithm based on deep neural network uses the neural machine translation technology to select the corresponding words from the corpus according to the maximum similarity principle with the help of the previous generated words. It transforms the sequence data by using the good transformation ability of the classical encoder-decoder framework, which transforms the source language sequence into the target language sequence. The classic structure has achieved good translation results, despite of the obvious structural and hierarchical characteristics of programming languages, when the neural machine translation method is applied to the generation of code summary, the source code will be treated as an ordinary text. This will inevitably cause the lack of source code structure and make the summarization effect of neural code summarization algorithm worse. Generally speaking, the accuracy of automatic code summarization system based on neural network is not high [18].

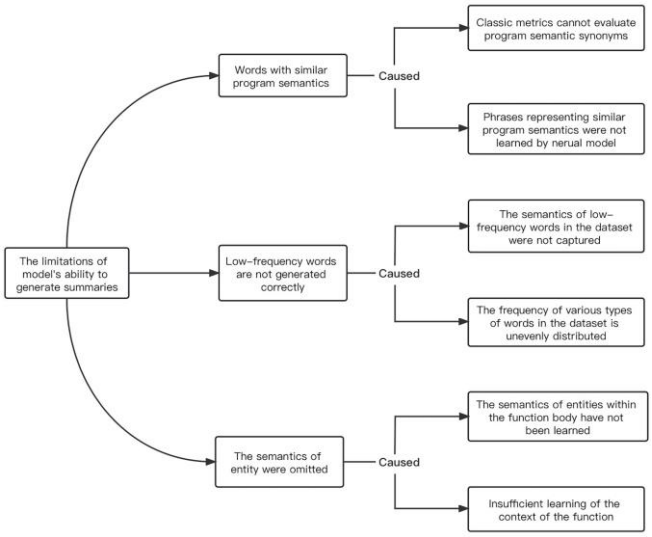

Fig. 1. The limitations of the model's ability to generate summaries.

To sum up, the neural model algorithm used for code summarization has two limitations, as shown in Fig. 1. First, only the sequence information in the code is taken into account by the encoder-decoder structure while the hidden semantics such as the structural information in the code are ignored [19] [20]. Second, the neural code summarization model based on maximum similarity will encounter the problem that low-frequency words or unknown words in the training data cannot be generated correctly during testing [21][22]. In this situation, even if the training data set is large enough and the quality is good enough, low-frequency words cannot be generated correctly; Moreover, when the summary model is applied to a code file in a different domain, there is also the problem of not being able to generate an accurate summary because of words for related domains that are not present in the training set. Above two kinds of findings are the main problems of neural code summarization algorithm.

Although scholars in related fields have identified these hidden dangers, there is currently no targeted solution for these specific problems in code summarization. Therefore, our study attempts to analyze the generated summaries by neural models to observe these phenomena and propose improvement ideas.

III. Research Methodologies

A. Extreme Code Summarization Task

To observe the utilization of program semantics in automatic code summarization, we conducted an experimental study by analyzing the acceptability of the code summaries generated from neural models. To determine whether our experiment can be generalized to different versions of the neural models, we re-implement the neural models from three selected studies as extreme code summarization solutions. The executive process of the neural model for the task of extreme code summarization is shown in Fig. 2. These neural models are trained by different procedures and can be used directly.

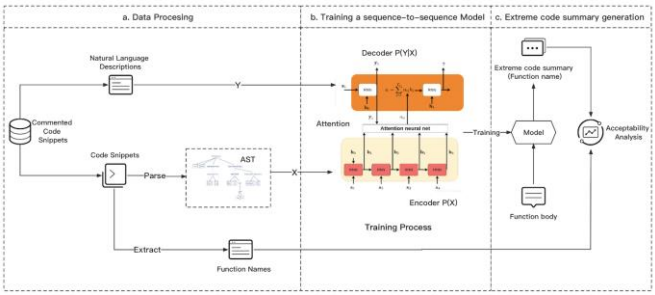

Fig. 2. Overview of models for extreme code summarization.

For the sake of better evaluating the universality of our research, we selected three representative models from different categories. Their different architectures may result in different focuses on learning source code semantics. Among them, code2Vec extracts AST path from the abstract syntax tree (AST) of Code, learns the vector representation of each path through the deep learning model and how to aggregate multiple paths into one vector to represent the entire Code; Code2seq uses LSTMs to encode paths node-by-node (rather than monolithic path embeddings as in code2vec), and an LSTM to decode a target sequence (rather than predicting a single label at a time as in code2vec); Code-Transformer is a Transformer based architecture that learns both source code (context) and an abstract syntax tree (AST) for parsing. In view of their different model architectures result in different ways of learning source code semantics, we infer that there may also be some differences in the generated summaries.

Therefore, Code2vec, Code2seq and Code-Transformer represent a set of diverse but representative models. Using the same dataset to evaluate the task of extreme code summarization on Code2vec, code2seq, and code-transformer highlights the potential risk of false negative and false positive generation when using neural models. Although we can't say for sure, other neural models trained on similar data set may exhibit similar behavior.

B. Dataset

For the task of extreme code summarization, a high-quality dataset plays a crucial role in the quality and acceptability of

the summary generated by neural model. Therefore, we chose the Java dataset proposed by Hu et al., which has been used to evaluate code summarization models such as Code-NN and Deep-Com by using common metrics of bleu, rouge, and meteor, and has achieved relatively complete experimental results.

Java dataset [5], including Java methods extracted from Java projects from 2015 to 2016, collected from GitHub. The first sentence of Javadoc is extracted as a natural language description, which describes the functions of Java methods. The quantity distribution of the dataset is shown in Table I.

TABLE I. JAVA DATASET STATISTICS

Dataset	Samples
Train	26142
Test	8714
Validation	8714

C. Evaluation Metrics

In order to better evaluate the quality of generated extreme code summaries, we selected three commonly used metrics in the field of code summarization: bleu, rouge, and meter.

1) BLEU: BLEU is used to compare the overlapping degree of n-gram in candidate translation and reference translation [23]. N-gram accuracy refers to the ratio of the total number of n-gram matches between the evaluated generated summary and the reference summary to the total number of n-grams in the reference summary. BLEU is often applied to evaluate the similarity between generated summary and reference text.

$$BLEU\_N = BP \cdot (\exp \sum_{n=1}^{N} \omega_n \log p_n) \quad (1)$$

Here Pn refers to the accuracy rate of n-gram; Wn refers to the weight of n-gram; BP is a penalty factor.

2) ROUGE: ROUGE is a quality evaluation method of text summary based on recall, it calculates the similarity between generated summary and reference text [23]. ROUGE-L is often applied to evaluate the quality of code summarization.

$$ROUGE - N = \frac{\sum_{S \in \{Reference\}} \sum_{gram_N \in S} Count_{match}(gram_N)}{\sum_{S \in \{Reference\}} \sum_{gram_N \in S} Count(gram_N)} \quad (2)$$

The denominator of the formula here is to count the number of n-grams in the reference translation, while the numerator is to count the number of n-grams shared by the reference translation and the machine translation [24].

3) Meteor: Meteor is used to calculate the score based on the clear word-word matching degree between the generated summary and the reference text [23], so it is often applied to evaluate the quality of the generated summary according to the score.

$$METEOR = \left(1 - \gamma \cdot \left(\frac{ch}{m}\right)^{\beta}\right) \cdot \frac{P \cdot R}{\alpha \cdot P + (1-\alpha) \cdot R} \quad (3)$$

Here, P and R are 1-gram accuracy and recall, ch is the number of blocks, M is the matching number.

4) Limitations of metrics: These three types of metrics are all calculated based on the degree of matching at the text level, and cannot be used to evaluate the degree of semantic similarity. All of them have a clear bias towards the order of words, which may lead to some false negatives and misjudgments in the results of extreme code summarization.

IV. EXPERIMENTS

A. Research Questions and Experimental Process

In order to explore the acceptability of the code summaries generated from neural models, we re-implement the models of code-transformer, code2vec and code2seq to perform the task of extreme code summarization, and conduct statistics and analysis for the preliminary experimental results. We found that different models have different qualities for summaries generated from the same piece of code, such as the length of generated summaries and the omission of semantic information.

Based on relevant development experience and previous research evidence, we propose the following research questions regarding the preliminary results of the task of extreme code summarization:

RQ1-1: How effectively do existing models employ program semantics for text-level matching?

RQ1-2: Why do many generated function names shrink in length compared to the original function names in the extreme code summaries generated by neural models?

RQ1-3: Whether different types of naming styles of function names affect the accuracy of the model in capturing semantics?

RQ2-1: Whether some synonyms representing the same program semantics can be identified during model learning?

RQ2-2: Whether the model's ability to capture the semantics of verbs greater than that of nouns?

RQ2-3: Will neural models only capture the semantics of words with the same name as function names while ignoring other important semantics?

Afterwards, we will design our experimental plan based on these research questions. The experimental process steps are shown in the following Fig. 3, and the experimental design plan and result analysis are shown in Section IV(B).

B. Experimental Analysis

We re-implement the models of code-transformer, code2vec and code2seq to perform the task of extreme code summarization, and get the preliminary experimental results. Then we use BLEU metric to divide the hit degree into four levels, we define the BLEU value greater than 0.7 as a high hit level and the BLEU value between 0.3 and 0.7 as a low hit level [25]. We calculated the proportion of the data sets

generated by the three models in each hit level; the preliminary experimental results are shown in Table II:

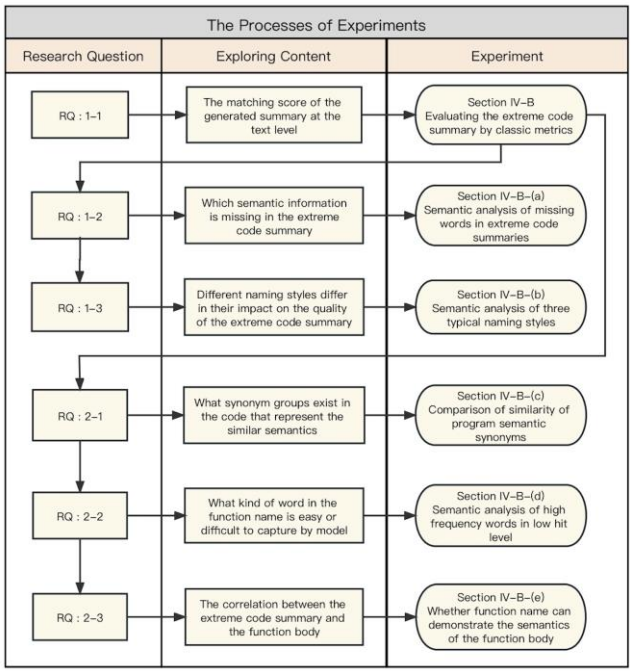

Fig. 3. Experimental process.

TABLE II. HIT LEVEL CATEGORY

Hit level	BLEU	Code2Vec	Code2Seq	Code-Transformer
Code-Equal	1.0	17.8%	18.1%	19.8%
High-Match	0.7~1.0	34.7%	36.9%	38.9%
Low-Match	0.3~0.7	32.8%	31.5%	31.4%
Code-Wrong	0~0.3	13.1%	12.8%	8.2%

From the above results in Table II, we found that there are many low matching phenomena between the extreme summaries generated by three models and the original function names. After an intuitive exploration of the generated summaries with the models trained from a Java dataset and based on relevant program development experience, we identify five acceptability aspects to be analyzed in detail: (a) the format of the function name; (b) function name naming style; (c) the semantic similarity in code; (d) the differences in hitting rate of functions; (e) the correlation between function name and function body. We found that the above five aspects of problems are common in the results generated by the three models, so we chose the Code-Transformer model with the best experimental result to analyze its result data from these five aspects in detail. The analytical process of the experiments as follows:

1) The format of the function name: We conducted preliminary observations on the generated results of models and found that it is very common that the generated extreme code summary is inconsistent with the length of the original function name after word segmentation. Compared with the length of the original function name, part of the extreme code summary generated by the model shrinks and part of the extreme code summary extends. Then, we counted the proportion of each phenomenon to analyze whether these phenomena are caused by the model's omission or analytic error of the semantic information of the function body.

TABLE III. THE FORMAT OF THE FUNCTION NAME

Preliminary experimental analysis	We compared and analyzed the length of the original function name and extreme code summary.
Observation and discovery	The generation of extreme code summaries has more shrinkage phenomenon and less extension phenomenon.
Put forward hypothesis	The semantic information within the function body has not been fully extracted and utilized by the neural model.
Verification Experiment	Which semantic information in function name were missed during the learning process of neural model.
Problem Analysis	The semantic information of function body is not fully utilized by neural model.

The analysis process is shown in Table III. Firstly, we do word segmentation for the original function names and the extreme code summaries and compare the length of them. Then we divided the results into three categories for statistical analysis, the ratio of them is shown in the Fig. 4. We observed that among the three categories, The model has more shrinkage and less extension for the generation of function names. Therefore, we put forward the hypothesis that the semantics of function is not fully extracted and utilized by neural model, leading to the serious shrinkage phenomenon.

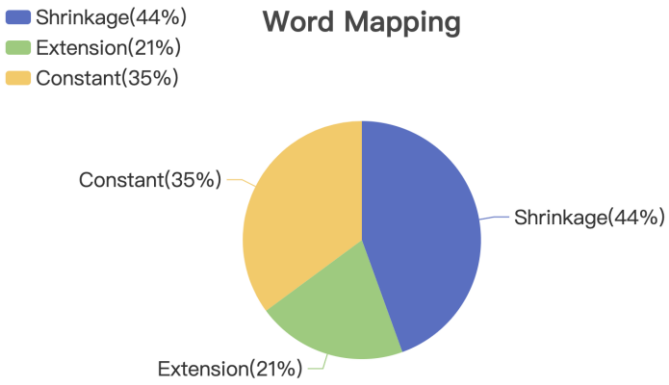

Fig. 4. Word length mapping.

In order to verify the hypothesis, we conducted a verification experiment; we analyze the function name, parameter list and function body in three categories respectively from the following two scenarios.

- Shrinkage Scenario: (including 3166 pieces of data): Function names in this category map from multiple words to fewer words. For example, the noun information in the parameter list of function is omitted: We select three types of the highest frequency verbs (get, set, add) to analyze their representative examples as shown in Table IV:

TABLE IV. THE HIT RATIO OF SHRINKAGE WORD

Origin_name	Prediction_name	Percentage
getProperties(Properties)	**get**	**36%**
setDisplay(dsiplay)	set	23%
addRenderer(Renderer)	add	21%

- Extension Scenario: (including 1069 pieces of data): Function names in this category map from fewer words to multiple words. For example, a preverb is added before the noun in the function name. Then we selected three kinds of data with the highest frequency according to the frequency of occurrence as shown in Table V below:

TABLE V. THE HIT RATIO OF EXTENSION WORD

Origin_name	Prediction_name	Percentage
id	**getid**	**19%**
Max	getMax	11%
XML	setXML	8%

Problem analysis: From the verification experiment results, we can conclude that part of the semantic information of the function body (such as the nouns in the parameter list) has been ignored during the process of model learning, which leads to the highest proportion of shrinkage in the results, resulting in the false positive in generated results.

2) Naming style of function name: Based on the preliminary observation of the results generated by the models, several representative words were selected and classified according to the program development experience: (1) Function names starting with "is" to indicate the judgment semantics; (2) Function names containing conjunctions (such as "to", "as", "of"); (3) Function names starting with common verbs. We want to explore how these different naming styles differ in generated extreme code summaries. The analysis process is shown in Table VI.

TABLE VI. FUNCTION NAMING STYLE

Preliminary experimental analysis	We selected several representative categories to classify the function names.
Observation and discovery	Function names with different naming styles appear in different hit levels.
Put forward hypothesis	Some commonly used conjunction words in function names will cause false positive in generated results.
Verification Experiment	Whether the semantics of function names with different naming styles can be fully captured
Problem Analysis	Function names with conjunctions and judgment words are not further preprocessed.

Firstly, we made quantitative statistics on their frequency in four different hit levels, as shown in the Fig. 5. Then we find that in the category of function names representing judgment, the proportion of low hit level is significantly higher than that of high hit level; In the category of function names containing conjunctions, the ratio difference between low and high hit levels is larger than that of the category representing judgment. In the function name category consisting of verb and noun classes, there is little difference in the proportion of low and high hit level. Therefore, we put forward the hypothesis that these function names with representative naming styles are not preprocessed, so the classic metrics cannot evaluate them correctly and result in false positive results.

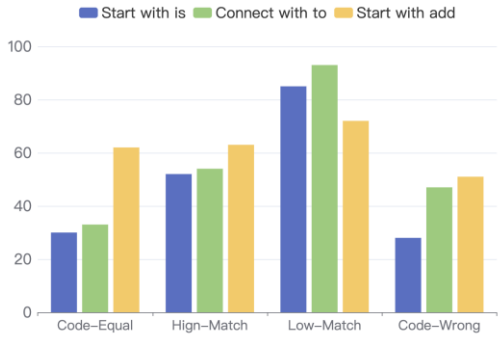

Fig. 5. Hit frequency in different level.

In the verification experiment, we compare the original function name with the generated function name data set after word segmentation. In the category of low hit level, we check for missing connectors in the generated extreme code summaries. However, the result is not as we expected, the conjunction such as "to" have not been omitted. We also find that the main reason why such words appear in low hit level frequently is that the nouns immediately after conjunctions are often omitted. Therefore, our hypothesis that conjunctions are omitted was overturned.

In the category of representing judgment function names starting with "is", the neural model focuses on capturing the semantic information of embedded function names during the learning process, resulting in a high frequency of occurrence in the category with a lower hit level.

In the category where the function names consisting of verb and noun, we infer that the noun that carries the important semantic information of the function body is often omitted, which leads to the phenomenon of false negative in result. Then we selected a representative high-frequency word in each of three categories and calculated their proportion in the same category is shown in the Table VII below.

TABLE VII. THE HIT RATIO OF REPRESENTATIVE WORDS

Origin_name	Prediction_name	Analysis	Percentage
isDoubleFile	isFile	Embedded function	4%
tobyte	tostring	Error in capturing business semantics	3%
addRenderer	**add**	**Parameter nouns are omitted**	**7%**

Problem analysis: From the verification experiment results, we can conclude that due to much important semantic information is not captured during model learning, resulting in the false positive in generated results.

3) The semantic similarity in code: We conducted preliminary observations on the results generated by the models and found that some frequent words in function names have specific program semantics; These words with special program semantics have more similar variants in the actual code, that is, there is the semantic similarity in program representation, and these variants can describe the semantics of similar function bodies. Therefore, we put forward the hypothesis that these phrases with similar program semantics cannot be captured by neural model; moreover, the metric cannot evaluate their similarity and result in false negative results.

TABLE VIII. THE SEMANTIC SIMILARITY IN CODE

Preliminary experimental analysis	We searched program semantic synonyms by wordnet thesaurus and artificial selection.
Observation and discovery	A lot of function names have similar variants in code, which can describe the similar semantics.
Put forward hypothesis	The semantic similarity in code cannot be evaluated by classic metrics.
Verification Experiment	Evaluation of representative synonyms with the same program semantics.
Problem Analysis	Function names with similar program semantics are not captured by neural model.

The analysis process is shown in Table VIII. Firstly, 213 pairs of synonyms identified from wordnet thesaurus were integrated with 84 pairs of synonyms selected manually for k-means cluster analysis, then four groups of synonyms with the highest frequency were selected, as shown in the Fig. 6.

Fig. 6. The four highest frequency synonym groups.

To verify our hypothesis, we use the bleu metric to calculate the similarity of each group of words after stemming, and the results of similarity calculation are all 0%, as shown in the Table IX, but the synonyms in each group can all represent the semantic of the function body. So it can be seen that the model will produce false negative results because these verbs with similar program semantics cannot be captured by neural model and evaluated by classic metrics.

TABLE IX. SEMANTIC SIMILARITY OF SYNONYMS

Origin_name	Prediction_name	Similarity
add	increase, append, plus	0%
create	find, get, insert	0%
write	build, comment, encode	0%
clear	free, reset	0%

Problem analysis: From the verification experiment results, we can conclude that since function names with similar program semantics are not captured by neural models, resulting in the false negative in generated results.

4) The differences in hitting rate of functions: We have preliminarily observed the generated results of models: The four types of words (“add”, “remove”, “write”, “read”) that represent addition, deletion, modification and selection in database operation for a high proportion in the generated results, and each type of words has a certain frequency in different hit levels. We want to make statistics on the occurrence frequency of these four representative words in different hit levels to explore whether the semantic of function body is not fully utilized, leading to the occurrence of these representative words in low hit levels.

TABLE X. THE DIFFERENCES IN HITTING RATE OF FUNCTIONS

Preliminary experimental analysis	We count the hitting ratio of four high-frequency verbs in different hitting levels.
Observation and discovery	High frequency words also appear frequently in low hit levels.
Put forward hypothesis	Common prefix verbs in function names can cause false positives in the generated extreme summary.
Verification Experiment	Whether function names that only contain verbs can represent the semantics of the function body.
Problem Analysis	Many nouns that represent business semantics have not been captured by neural model.

The analysis process is shown in Table X. We sampled four kinds of verbs with the highest frequency from the data set including the “verb + noun” combination whose first word is this verb, the four kinds of verbs are “add”, “remove”, “write” and “read” respectively.

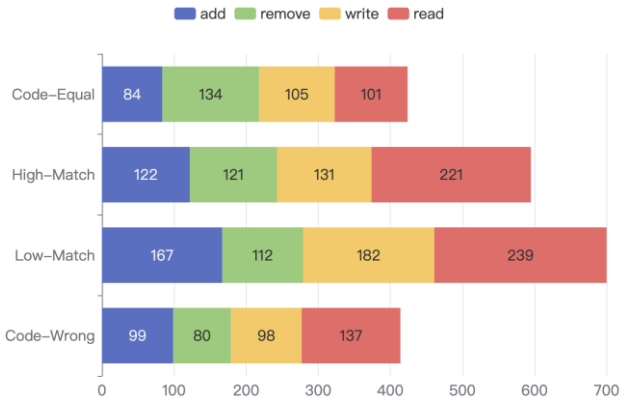

Fig. 7. Four kinds of words hit frequency in different level.

Then, we count the numbers of these four words in above four hit levels proposed in Table II, the statistic results are shown in the Fig. 7. By preliminary observation, we find that four types of words appear frequently both in high hit and low hit levels, so we put forward the hypothesis that the semantics in function body are not fully extracted and utilized by the neural model, which leads to false negative results.

To verify our hypothesis, we made a statistical analysis on the function body of four kinds of words extracted from the original data set. We observed that there are embedded function names with the same name as the extreme code summary generated by neural model in these function bodies, which may cause the model to ignore the semantic information of other nouns within the function body, and leads to the false negative results of the model. Then we calculate the proportion of highest frequency words in four types of categories in the low hit level as shown in Table XI:

TABLE XI. THE HIGHEST HIT RATIO OF FOUR TYPES OF WORD

Origin_name	Prediction_name	Percentage
add	append	31%
remove	**delete**	**44%**
write	encode	26%
read	find	30%

Problem analysis: From the verification experiment results, we can conclude that due to the fact that many nouns that represent business semantics in the function body, except for verbs, has not been captured by the model during learning process, resulting in false negative in generated results.

5) The correlation between function name and function body: By comparing the generated extreme code summary with the function body of the original data set, we find that many generated extreme code summaries are inconsistent with the original function names, but they are consistent with the embedded function names in the original function body. We propose the hypothesis that this phenomenon may be caused by the model concentration learning the semantics of the embedded function body while ignoring other important semantics.

TABLE XII. THE CORRELATION BETWEEN FUNCTION NAME AND FUNCTION BODY

Preliminary experimental analysis	We compare the generated function name with the function body of the original dataset.
Observation and discovery	Many generated function names are consistent with the embedded function names in the function body.
Put forward hypothesis	The semantic information of the function body was not fully captured by the model.
Verification Experiment	Whether embedded function names can represent the semantics of their function body.
Problem Analysis	Other important semantic information within the function body was not captured by neural model.

Embedding function name: First, we define the embedded function name: that is, the function name that appears in a one-line statement in the function body. For example, “write” is the embedding function name in Fig. 8 below.

```
public void write (String key, byte[] newValue) {
    Map<String, byte[]> entry = new HashMap<>();
    entry.put(key, newValue);
    write(entry);
}
```

Fig. 8. The embedded function name “write”.

The analysis process as shown in Table XII. According to the development experience, we classify these embedded function names into four categories: (1) including common verb, (2) including conjunctions, (3) mathematical functions, (4) representing judgement category; We count the function names with the highest frequency in these four categories by frequency, the result as shown in the Fig. 9.

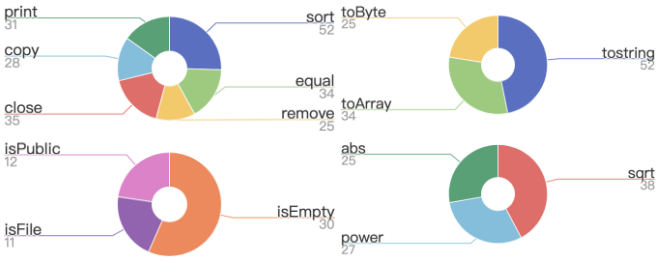

Fig. 9. The high frequency words in four groups of phrases.

To verify the hypothesis, we counted and analyzed the mapping number between the above four class function names and the function names embedded in the function body.

There are 1139 pieces of data embedded with the same function name as the original. We selected the three most frequent words and calculated their proportion in their category as shown in the Table XIII below.

TABLE XIII. THE HIGHEST HIT RATIO OF CONSISTENT WORD

Origin_name	Prediction_name & Embedded function name	Percentage
Sort	**sort**	**11%**
Tostring	tostring	9%
IsEmpty	isEmpty	9%
Sqrt	sqrt	5%

There are a total of 478 extreme code summaries that are the same as the embedded function names in the function body, but different from the original function names. It can be seen that the model concentrates on learning the local program semantics of some embedded functions while ignoring other semantics in function body, which leads to the false negative result of the model. We selected three kinds of verbs with the highest frequency for statistical analysis as shown in the Table XIV below.

TABLE XIV. THE HIGHEST HIT RATIO OF INCONSISTENT WORD

Origin_name	Prediction_name & Embedded function name	Percentage
copy	**write**	**14%**
delete	remove	13%
reset	clear	8%

Problem analysis: From the verification experiment results, we can conclude that due to the fact that many semantic information other than embedded function name in the function body was not captured by neural model, resulting in false negative in generated results.

C. Ablation Study

Based on the statistical study of these five aspects, to further explore the impact of various aspects on the model's ability to capture semantics hidden in source code, we conducted the ablation experiment, in which we respectively improve the preprocessor statement of data sets in terms of function name format, function naming style, semantic level similarity, the differences in hitting rate of functions and the correlation between function name and function body, then we evaluate the ablation experimental results by using Bleu, Rouge and Meteor metrics, as shown in Table XV.

1) False negative aspect: In terms of the three aspects that produced false negative results, we performed the following ablation experiments.

a) The differences in Hitting Rate of Functions: We filter four types of high-frequency verbs in low hit level category.

b) The Correlation between Function Name and Function Body: We filter the function name data set which omits nouns in the parameter list from the data set whose embedded function name is inconsistent with the original function name.

2) False positive aspect: In terms of the two aspects that produced false negative results, we performed the following ablation experiments:

a) The Format of the Function Name: We filter the function name data set with omitted parameters in the function name data set with shrinkage scenario.

b) The Naming Style of Function Name: We filter out "is" in the function name data set of representing judgment class; We filter out the pre-verbs in the data set of the function name consisting of verb and noun; We don't deal with the conjunctions.

3) Ablation result: Ablation experiment results (Table XV) show that semantic similarity in program has a stronger influence on false negative in results, the format of function name has a stronger influence on false positive in results.

D. Insights Gained From Experiments

Based on the analysis of experimental results and further validation of ablation experiments on the above research questions, we can make some improvements to the model for executing the task of extreme code summarization in terms of preprocessing filtering enhancement, external data source enhancement, and attention mechanism enhancement. The specific optimization steps are outlined in red dashed lines in Fig 10.

1) Preprocessing filtering enhancement: For the cases of different types of naming styles of function names in section B-b and function names with high correlation with function bodies in section B-e, we will seek optimization from the perspective of data preprocessing. We plan to filter out common prefixes of data words with specific naming styles and embedding function names during the preprocessing process to reduce the occurrence of false positives in the generated results.

2) External data source enhancement: For the situation that the synonym group representing the same program semantics in section B-c cannot be recognized by the program, we plan to import the constantly improving program semantic synonym library as an external data source and integrate it with summary information during the model training process, so that the neural model can gain data enhancement in the process of learning the text of summary to avoid false negatives in the generated results.

TABLE XV. ABLATION EXPERIMENTAL RESULT

Aspect		Code2Vec			Code2Seq			Code-Transformer		
		BLEU	ROUGE-L	METEOR	BLEU	ROUGE-L	METEOR	BLEU	ROUGE-L	METEOR
False Negative	**Semantic Similarity**	**48.85**	**63.86**	**31.79**	**51.81**	**63.84**	**32.77**	**52.88**	**64.89**	**29.83**
	Classification Hit Rate	39.79	54.81	25.74	40.76	53.82	24.75	42.84	53.87	21.78
	Func_Body_Correlation	37.98	52.80	22.70	39.74	51.81	23.71	40.82	50.83	20.76
False Positive	**Naming Format**	**47.54**	**58.51**	**30.49**	**49.52**	**60.50**	**29.46**	**50.59**	**59.61**	**32.53**
	Naming Style	37.58	52.55	21.54	38.54	49.54	18.52	41.62	51.63	23.56

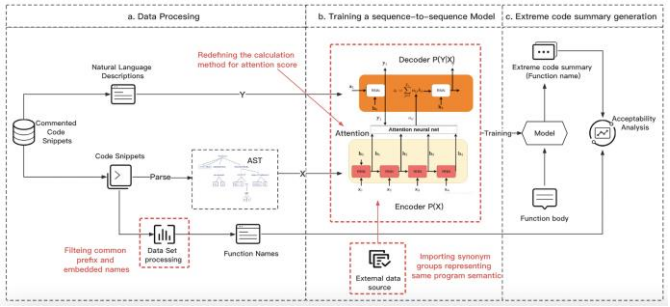

Fig. 10. The Diagram of model architecture for extreme code summarization.

3) Enhancement of attention mechanism: For the cases where the important program semantics in section B-a and the noun semantics in section B-d are omitted during the model learning process, we will seek optimization from the perspective of attention mechanism. In the attention mechanism, each piece of information is assigned a different attention score. If the attention score of important semantic information is low, it may cause the output sequence information to lose this part of semantics. Therefore, we can try to innovate in the calculation methods of attention score, such as dot product, multiplication, addition, or other more complex calculation formulas, then assign new attention score to each information, so as to improve the attention score of important semantic information and avoid false positives in the results as much as possible.

V. Threat to Validity

1) Model re-implementation: In the process of reimplementing the three models, the word length of some data sets exceeds the limit. For example, the length of words in the function body of code-transformer cannot exceed 1200, and the length of code2vec and code2seq exceeding 900 will also cause model parsing failure. In order to avoid this situation, we need to filter out relevant nonconforming data in the process of data set preprocessing.

2) Selected dataset: Java dataset has a total of 8714 pieces of data. Although the sample data is of high quality and representative, and the domain knowledge is perfect, the overall scale is small. If we want to retrain the data set of the model in the future, we should inject a larger data set.

3) Model comparison: We choose the three represents extreme code summary generation model. In our experiment, we use the same data set, run all models in the same hardware environment, and adopt the same data preprocessing process to reduce this threat.

VI. Conclusion

Many studies show that the quality of code summary generation algorithms based on deep learning is not ideal because it does not take full advantage of relevant program semantic information. In this paper, in order to observe the utilization of program semantics in automatic code summarization, we conducted an experimental study by analyzing the acceptability of the code summaries generated from neural models. To focus on the observation of the semantics, we re-implement the neural models from three selected studies as extreme code summarization solutions. Fig. 10 shows the diagram of model architecture for extreme code summarization. After an intuitive observation and exploration of the generated summaries with the models trained from a Java dataset, we identify five acceptability aspects: (1) function name format; (2) function naming style; (3) semantic level similarity; (4) the differences in hitting rate of representative words; (5) the correlation between extreme code summaries with function body. Experimental analysis shows that false negative is common in the results if only evaluated with classic metrics, and aspects (3)(4)(5) bring the major influence. We also observed that false positives related to aspects (1)(2) also commonly appeared in the result, which suggests that the current models also fail to filter the noise from the raw source code to a reasonable extent.

We put forward hypotheses for these above five aspects, for example, the semantics of the function body may not have been fully learned by neural model. Then we designed and completed relevant verification experiments to prove whether our hypotheses are correct. The verification experiment confirmed that aspects (2)(5) is caused by insufficient preprocessing of the data set, aspects (1)(3)(4) are caused by the semantics of function body have not been fully extracted and utilized by neural model.

To further explore the influence of the above five aspects on the quality of extreme code summaries, we conducted ablation experiments which indicated that aspect (3) had a stronger influence on false negative in extreme code summarization results than the other aspects (4)(5), The aspect (1) has a stronger influence on false positive in extreme code summarization results than aspect (2). The results of ablation experiment illustrate prove the significance and potential of utilizing the program semantics explicitly in code summarization.

Therefore, based on the experimental results and findings, in the future study, we plan to improve the model in performing code summarization tasks from three aspects of preprocessing filtering enhancement, external data source enhancement and attention mechanism enhancement, which have been mentioned in section IV-D. Let's wish all these findings promote the progress in the field of code summarization.

References

[1] Allamanis M, Barr ET, Devanbu P, et al. Code Generation as a Dual Task of Code Summarization [J]. ACM Computing Surveys (CSUR), 2020, 51(4): 1-37.

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2017. A Survey of Machine Learning for Big Code and Naturalness. arXiv preprint arXiv:1709.06182 (2017).

[3] Raychev V, Vechev M, Krause A. Predicting program properties from" big code"[J]. ACM SIGPLAN Notices, 2015, 50(1): 111-124.

[4] Parisotto E, Mohamed A, Singh R, et al. Neurol-symbolic program synthesis[J]. arXiv preprint arXiv:1611.01855 (2016).

[5] Liu S, Chen Y, Xie X. A Convolutional Attention Network for Extreme Summarization of Source Code [J]. arXiv preprint arXiv:1802.03691 (2017).

[6] Wang K, Singh R, Su Z. Reassessing Automatic Evaluation Metrics for Code Summarization Tasks [J]. arXiv preprint arXiv:1711.07163(2020).

[7] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E. Gonzalez, and Ion Stoica. 2020. Contrastive Code Representation Learning. *CoRR* abs/2007.04973 (2020).

[8] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*. Association for Computational Linguistics, Barcelona, Spain, 74–81.

[9] Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2021. Retrieval-Augmented Generation for Code Summarization via Hybrid GNN. In 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021.

[10] Monperrus M. Summarizing Source Code using a Neural Attention Model [J]. ACM Computing Surveys (CSUR), 2018, 51(1): 1-24.

[11] Hellendoorn V J, Sutton C, Singh R, et al. Deep code comment generation with hybrid lexical and syntactical information [C]//International conference on learning representations (2018).

[12] Vasic M, Kanade A, Maniatis P, et al. Sequence to Sequence Learning with Neural Networks [J]. arXiv preprint arXiv:1904.01720 (2016).

[13] Guo D, Ren S, Lu S, et al. Graphcodebert: Pre-training code representations with data flow[J]. arXiv preprint arXiv:2009.08366 (2021).

[14] Chen X, Liu C, Song D, et al. Multi-modal attention network learning for semantic source code retrieval[J]. arXiv preprint arXiv:1909.13516 (2019).

[15] Alon, Uri, Meital Zilberstein, Omer Levy, and Eran Yahav. "code2vec: Learning distributed representations of code." Proceedings of the ACM on Programming Languages 3, no. POPL (2019): 1-29.

[16] Uri Alon, Shaked Brody, Omer Levy, Eran Yahav: code2seq: Generating Sequences from Structured Representations of Code. ICLR,2019.

[17] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann, "Language-agnostic representation learning of source code from structure and context". ICLR, 2021.

[18] Kishore Papineni, Salim Roukos, Todd Ward,et al. BLEU: a Method for Automatic Evaluation of Machine Translation[J]. ACL, Philadelphia, July 2002, pp. 311.

[19] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer Normalization. *CoRR* abs/1607.06450 (2016).

[20] Zi Gong, Cuiyun Gao, Yasheng Wang, Wenchao Gu, Yun Peng, and Zenglin Xu. 2022. Source Code Summarization with Structural Relative Position Guided Transformer. *CoRR* abs/2202.06521 (2022).

[21] Xing Hu, Xin Xia, David Lo, Zhiyuan Wan, Qiuyuan Chen, and Tom Zimmermann. 2022. Practitioners' Expectations on Automated Code Comment Generation. In ICSE '22: Proceedings of the 44th ACM/IEEE International Conference on Software Engineering.

[22] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. 2019. Automatic Generation of Pull Request Descriptions. In *34th IEEE/ACM* International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019. IEEE, 176–188.

[23] Paul W. McBurney and Collin McMillan. 2016. Automatic Source Code Summarization of Context for Java Methods. *IEEE Trans. Software Eng.* 42, 2(2016), 103–119.

[24] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. In *SANER*. IEEE, 261–271.

[25] Lun Yiu Nie, Cuiyun Gao, Zhicong Zhong, Wai Lam, Yang Liu, and Zenglin Xu. 2020. Contextualized Code Representation Learning for Commit Message Generation. *CoRR* abs/2007.06934 (2020).