A Naming Pattern Based Approach for Method Name Recommendation

Yanping Yang[1,2], Ling Xu[1,2*], Meng Yan[1,2], Zhou Xu[1,2], Zhongyang Deng[1,2]
[1]Key Laboratory of Dependable Service Computing in Cyber Physical Society (Chongqing University), Ministry of Education, China
[2]School of Big Data and Software Engineering, Chongqing University, Chongqing, China
Email:{ypyang, xuling, mengy, zhouxullx, zy.deng}@cqu.edu.cn

***Abstract*—Method names in software projects are significant for developers to understand the method functionality. Existing state-of-the-art automated approaches tend to explore tokens composing method names from method contexts. However, the method name is not a simple combination of tokens, as it is structured and contains many repetitive naming patterns (e.g. *"get____"*, *"create____"*). Through a large-scale empirical analysis on 15M methods from 14K real software projects developed with Java codes, we found repetitive naming patterns in method names. In addition, the names of two function-similar methods usually have the same naming pattern.**

Based on our empirical study, we propose a naming pattern-based approach for method name recommendation, named NamPat. Specifically, for a target method, NamPat first retrieve the most similar method from the training data by estimating their body code similarity. Then, the name of the most similar method is used as the pattern guider to provide the naming pattern, and NamPat combines it with the context information of the target method to perform method name recommendation. To verify the effectiveness of the proposed approach, we conducted experiments on 17M methods from a widely used Java dataset. Experimental results show that compared with Code2vec, Code2seq, MNire, and Cognac, NamPat improves the state-of-the-art approaches in precision (5.8%-27.1%), recall (11.1%-60.1%), and F-score (8.5%-43.9%), which proves the effectiveness of our proposed approach.

***Index Terms*—method name recommendation, naming pattern, deep learning**

I. Introduction

Meaningful and succinct method names usually reflect the functionality of the method, which help developers understand the program behavior [1]–[3]. In contrast, misleading method names may confuse developers and cause API misuses, leading to software defects [4]–[6]. Therefore, it is crucial to find a suitable name for the method. Actually, constructing a high-quality method name is a challenging and time-consuming task [7].

To obtain meaningful method names, researchers have proposed various automated approaches to recommend meaningful method names. Early studies mainly employed Information Retrieval (IR) approaches [8], [9] such as Liu et al. [9] deemed that two methods with similar bodies should have similar names. They retrieved the similar methods and reused their method names. However, in many cases, two methods with similar bodies are given different names because they might have different semantics or for various tasks. Besides, the IR-based approaches cannot recommend new method names that are unseen in the training data.

With the rapid development of deep learning, researchers use deep learning technologies to recommend appropriate method names [10]–[13]. Based on the idea that programming languages can greatly benefit from representations that leverage the structured nature of their syntax [14]–[16]. Code2vec [11] and Code2seq [10] used a set of paths between two leaf nodes in the Abstract Syntax Tree (AST) to represent the method body for predicting the method's name. Nguyen et al. [12] noted that for method naming, the naturalness factor, i.e., the names of program entities, are more important than structured representation. Therefore, they proposed MNire, which used the program entity tokens of three contexts (i.e., implementation context, interface context, and enclosing context) to generate the method name. Wang et al. [13] found that MNire still has limitations in dealing with the methods having little content, so they developed Cognac, which introduced the caller/callee context of the method as prior knowledge to complete the recommendation of method names.

The above state-of-the-art approaches mainly focus on introducing more context information which may contain the target method name tokens for recommending method names. Their approaches usually generate the method names by composing the tokens extracted from context information. However, we find that method names contain structured naming patterns, and such combinations may lead to poor readability of method names. We define that if a method name starts with a verb or preposition, the method name is regarded as having a naming pattern. For example, a method named *"createItemForm"*, its naming pattern is *"create____"*. This naming pattern is helpful in understanding the method's intention and improving the readability of the method name. Therefore, this finding motivates us to investigate whether the naming pattern can be utilized to infer meaningful method names better. We also observe that two methods with similar body code have the same naming pattern. For example, two methods named *"getImage"* and *"getText"* have token-level similarity in the body code and have the same naming pattern *"get____"*. Therefore, to obtain the naming pattern of the target method name, we use the name of the most similar method as the pattern guider to provide the naming pattern. Then, we combined pattern guider

*Corresponding author.

with the context semantic information of the target method, to recommend meaningful method names.

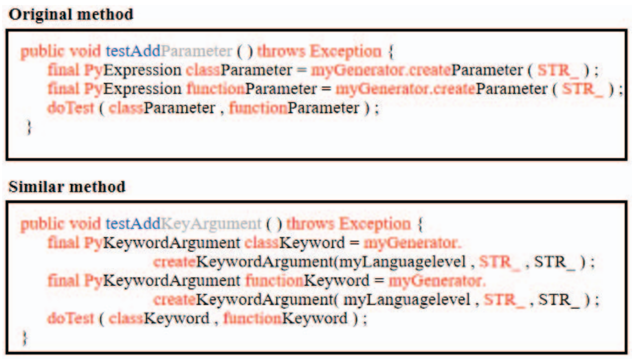
To verify our observations and motivation, we extract 15M methods from 14K high-quality projects from the GitHub repository and conduct a large-scale empirical research. We find that there are a lot of repetitive naming patterns in method names, and similar methods usually have the same naming pattern. In detail, among all the methods studied, in 58.53% of the methods, their method name is named by statistical naming patterns, such as *"test____"*, *"get____"* and *"set____"*. And the proportion that the method names of two methods with similar body code have the same naming pattern is 66.15%. We also find that the tokens of the target method name can be found in the name of its most similar method for 75.5% of the total cases. These results show that the name of the most similar method can provide additional information to help infer method names.

Supported by the results of the above empirical research, we propose NamPat, an approach that uses the name of the most similar method as the pattern guider to provide a naming pattern, and then combines the contexts of the method to generate method names. First, NamPat retrieves the most similar method for the target method from training data by estimating their body code similarity through Lucene [17], a widely used text search engine. And NamPat uses the name of the most similar method as the pattern guider. Then, NamPat extracts context token sequence (implementation context, interface context, and enclosing context) from the target method to represent the semantics of the target method. It is better to use the token sequence of program entities in the context for method name recommendation [12]. Finally, NamPat combines the naming pattern in the pattern guider with the semantic information provided by the context token sequence to recommend method names.

We conduct experiments on a previously widely used dataset [12] with 10K GitHub projects to evaluate the performance of NamPat with Code2vec [11], Code2seq [10], MNire [12] and Cognac [13]. The results show that NamPat improves the state-of-the-art approaches in precision (5.8%-27.1%), recall (11.1%-60.1%), and F-score (8.5%-43.9%).

In summary, our main contributions are outlined as follows:

- **Empirical Study:** Our research finds that method names have many repetitive naming patterns, and methods with similar body code usually have the same naming pattern.
- **Method name recommended method:** We propose a novel approach NamPat for method name recommendation. It uses the name of the most similar method as the pattern guider to provide the naming pattern and combines the contexts of the method to recommend method names.
- **Performance evaluation:** We conducted experiments on a large-scale Java dataset to evaluate the performance of NamPat. The experimental results show that NamPat achieves overall better performance than the state-of-the-art approaches.



```
public void testAddParameter ( ) throws Exception {
    final PyExpression classParameter = myGenerator.createParameter ( STR_ ) ;
    final PyExpression functionParameter = myGenerator.createParameter ( STR_ ) ;
    doTest ( classParameter , functionParameter ) ;
 }
```

```
public void testAddKeyArgument ( ) throws Exception {
    final PyKeywordArgument classKeyword = myGenerator.
                createKeywordArgument(myLanguagelevel , STR_ , STR_ ) ;
    final PyKeywordArgument functionKeyword = myGenerator.
                createKeywordArgument( myLanguagelevel , STR_ , STR_ ) ;
    doTest ( classKeyword , functionKeyword ) ;
}
```

Fig. 1. An example of the original method and a similar method retrieved

The rest of the paper is organized as follows: Section II describes motivating examples. Section III details the empirical research on the name of the method. Section IV introduces the details of our proposed approach NamPat. Section V and Section VI describe the experimental setup and evaluation results. Section VII and Section VIII discuss some details and describe the related work, respectively. Finally, Section IX concludes the paper.

II. Motivating Examples

In this section, we present an example of obtaining the naming pattern from the name of the most similar method to help generate method names. Figure 1 shows an original method, named *"testAddParameter"*, having a naming pattern *"test____"*. we retrieve its most similar method through the Lucene engine according to the code tokens of *"testAddParameter"* (the implementation details are described in Section IV). Then we gain the most similar method *"testAddKeyArgument"* from the code base. We observe that the most similar method's name have the same naming pattern *"test____"* as the name of original method. Further, in this example, we can get a detailed naming pattern *"testAdd____"*. From the detailed naming pattern, we can find that the similar function of both methods is to complete the test addition. This motivates us to retrieve the most similar method for the original method and extract their method names to capture the naming pattern.

Besides observing the name pattern from the most similar method's name, we also find the tokens that make up the original method's name can be observed in the name of the most similar method. For example, the tokens *"test"* and *"add"* from the original method name *"testAddParameter"*, which also appear in the name *"testAddKeyArgument"* of its most similar method. While the token *"add"* does not appear in the contexts of the original method *testAddParameter*. This discovery prompts us to enrich the semantics of a method name by incorporating its most similar method name.

Given the above observations, we address that the naming pattern of the most similar method can be used to help method name recommendation. In this paper, we propose an approach to generate a proper method name by combining the name pattern in its most similar method's name and semantic context information of the method.

III. Empirical Study

Based on our observations, we conducted an empirical study to ensure that our observations are universal across large open-source projects. Specifically, this empirical study aims to answer the following questions:

RQ1: Whether method names have repetitive naming patterns?

This RQ aims to explore whether there are naming patterns for method names in a large-scale dataset. In addition, whether two methods with similar body code have a same naming pattern.

RQ2: Can the tokens composing the name of a target method be frequently observed in the name of its similar method?

This RQ aims to investigate the frequency that similar method names contain the target method name token and whether the similar method names can help us better predict the method names.

A. Concepts

To simplify the representation, we give a brief definition of implementation context, interface context, enclosing context, and pattern guider.

Implementation context: For a method, the implementation context [12] is the names of all program entities in the method body. This shows the implementation of the method.

Interface context: For a method, the interface context [12] is the type of method input parameter and method return type. This represents the input and output of the method.

Enclosing context: For a method, the enclosing context [12] is the name of the class that defines the method. This represents the general task purpose information of the class of the method.

Pattern guider: For a method, the pattern guider is the name of its most similar method. This represents the naming pattern of the method name.

B. Data collection and processing

To complete the empirical research, we use a dataset of 14,317 top-ranked and well-maintained java projects on GitHub, which was used in previous work [12]. We processe 15,294,489 Java methods from this dataset with the latest, stable project version, ensuring that method names have stable status. For each method processed, we collect its contexts, including the types of the parameters, the return type, the enclosing name, and the body of the method. We split the collected names into tokens using the camel case and underscore naming convention, converting the obtained tokens to lowercase. Then, we use the Lucene search engine to retrieve

TABLE I
The naming patterns and scale of the method names

Method Name Pattern	Number	Proportion	Same Pattern Proportion
get____	3707168	0.2424	0.5674
set____	1534798	0.1003	0.7063
test____	1002558	0.0656	0.9275
is____	485527	0.0317	0.5159
create____	463080	0.0303	0.7257
add____	317731	0.0208	0.6485
to____	227229	0.0149	0.5669
on____	205359	0.0135	0.8226
remove____	134576	0.0088	0.5842
write____	124636	0.0082	0.8219
update____	114821	0.0075	0.6207
read____	109325	0.0071	0.7702
do____	103154	0.0067	0.7008
find____	93031	0.0061	0.7169
visit____	90968	0.0060	0.8665
check____	81328	0.0053	0.6443
run____	77877	0.0051	0.6352
parse____	76504	0.0050	0.7442
Total	8949670	0.5853	0.6615

the most similar method for each method and extract its method name. As a result, each method sample includes the target method context sequence, the target method name, and the name of the most similar method.

C. Statistical Analysis

RQ1:Naming patterns in method names. Based on observation, we count 18 naming patterns in the dataset. The results are illustrated in Table I, about 58.5% (8,949,670/15,294,489) of the methods have naming patterns. For example, 24.2% (37,007,168/15,294,489) of methods have the pattern *"get____"*, 10.0% (1,534,798/15,294,489) of methods have the pattern *"set____"*. These numbers indicate that there are many naming patterns in method names, and many method names use the same naming pattern.

Then, we further study the above methods with naming patterns. The results are shown in Table I. In 66.1% of cases, when a method has the above naming pattern, we can get the same naming pattern from its most similar methods. For example, in all the methods using the *"get____"* naming pattern, their most similar methods also use the *"get____"* naming pattern, accounting for 56.7%. These results motivate us to use the name of the most similar method to obtain the naming pattern of the target method.

Finding-1: *Nearly 58.5% of methods have naming patterns, and the method names of two similar methods usually have the same naming pattern, which implies that we use naming patterns to generate method names.*

RQ2: Frequencies of tokens can be observed in the name of the most similar method. We investigate the frequency that the tokens composing the target method name are found in the name of the most similar method. From 15,294,489 method names, we extract 41,119,385 tokens, which means that each method name is composed of three tokens on average. There-

TABLE II
THE FREQUENCY OF METHOD NAMES BE FOUND IN SIMILAR METHOD NAMES.

	Number	Frequency
# **The number of method name samples**	15294489	-
# Not in method contexts	3390396	22.2%
# In similar method name	11551101	75.5%
# In similar method name but not in method contexts	1794625	11.7%
# **The number of method name tokens**	41119385	-
# Not in method contexts	23489730	42.9%
# In similar method name	26177105	63.7%
# In similar method name but not in method contexts	8879199	21.6%

fore, we analyze the frequency from two perspectives: method names and tokens.

As shown in Table II, we count the frequency of method name tokens that are not observed in the method context. The results showed that 22.2% of method names, any constituent tokens of them could not be observed in their context. And almost 42.9% of tokens in method names cannot be observed in their context. These results imply to us that it is not enough to rely on context information to predict method names.

Then, we count the frequency of method name tokens that are observed in the name of the most similar method. The results showed that 75.5% of method names could be observed with any constituent tokens in the name of the most similar method. Almost 63.7% of tokens in method names can be observed in the name of the most similar method. This means that the name of the most similar method can provide rich prediction information for the target method name. After that, for 11.7% of the methods, the tokens composed of method names can only be observed in the name of the most similar method. Nearly 21.6% of the method names constitute tokens, which can only be observed in the name of the most similar method. These results indicate that the name of the most similar method can provide unique prediction information for the target method name.

Finding-2: *For the tokens constituting the method name, 63.7% of the tokens can be observed in the name of similar methods, of which 21.6% can only be observed in the name of similar methods. This implies that we can use the method names of similar methods to provide prediction information for the target method names.*

IV. METHODOLOGY

In this work, we propose NamPat, an approach based on naming patterns for method name recommendation. The overview of NamPat is shown in Figure 2. It mainly includes two modules: naming pattern extraction and method name generation. In the naming pattern extraction module, a retrieval base is built based on the training dataset. For a method, we use the open-source search engine Lucene to retrieve the most similar method according to the input method tokens. Then, the name of the most similar method is extracted as the pattern guider with the naming pattern. Meanwhile, the context sequence is extracted for the input method to represent the semantics of the input method. In the method name generation module, we employ a seq2seq framework to generate the method name. Specifically, we use two encoders to encode the pattern guider and the contexts into vector representations, respectively. The decoder generates the target method name sequentially by predicting the probability of the tokens in the method name.

A. Naming Pattern Extraction Module

The results of statistical analysis in Section III show that the naming pattern can be obtained from the name of the most similar method to help predict method names. So, the naming pattern extraction module aims to extract the context sequence and get the name of the retrieved most similar method.

Inspired by previous studies [18], [19], we build a tokens-level similarity-based retrieval module. Specifically, we use BM25 [20] as the similarity evaluation metric to estimate the relevance of methods. BM25 is a bag-of-words retrieval function that is widely used. Given a method without the method name, the BM25 function based on TF-IDF [21] calculates the term frequency in the document of each token in the method and multiplies it by the term's inverse document frequency. The more relevant the two methods are, the higher the BM25 score is. We use the open-source search engine Lucene [17] to build the retrieve component and the training set as the retrieval corpus.

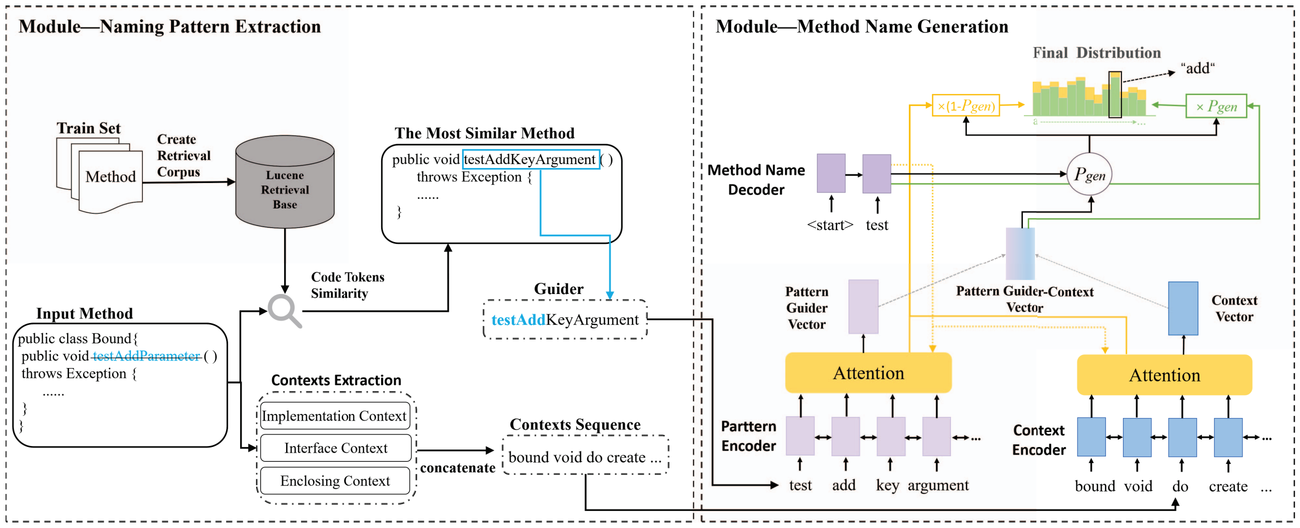
As depicted in Figure 2, input a method without the method name, the Lucene-based retrieval component returns its most similar method according to the similarity of code tokens. Then, we extract the method name from the most similar method and process it into a token level sequence, which is a pattern guider with the naming pattern. On the other hand, for the input method, we will extract three contexts in the method, including implementation context, interface context, and enclosing context, then process them into token level sequence and connect them. Thus, for the input method, we get the pattern guider sequence with naming pattern and context sequence for the subsequent method name generation module.

B. Method Name Generation Module

As shown in the right part of Figure 2, this module describes the seq2seq attention model architecture used in NamPat, which is a novel pointer generator network [22]. The reason why we adopt this network is to enable the model to generate tokens that do not exist in the vocabulary to alleviate the out-of-vocabulary (OOV) problem in the method name recommendation.

The key step of this module is to learn the pattern in the pattern guider and combine it with the context information of the method to generate a new method name. Specifically, we utilize the encoder to get the vector representation of pattern guider and contexts, respectively. Lastly, the method name decoder is used to generate a new method name conditioning on the guider and contexts representation.

1) Pattern And Contexts Encoder. We input the context token sequence and the pattern guider token sequence obtained in



```
public void testAddKeyArgument( )
    throws Exception {
    ......
}
```

```
public class Bound{
 public void ~~testAddParameter~~( )
 throws Exception {
     ......
 }
}
```

Fig. 2. Overview of NamPat

the previous module into the model. The encoder embeds the input context tokens into a vector $x = (x_1, x_2 ... x_m)$, and the pattern guider tokens into a vector $x^{'} = (x^{'}_1, x^{'}_2 ... x^{'}_n)$. Then, the two embedded vector sequences are processed using a single-layer bidirectional long short-term memory (Bi-LSTM) [23] unit respectively. The hidden state h_i of Bi-LSTM can be represented as:

$$\vec{h}_i = LSTM(\vec{h}_{i-1}, x_i); \overleftarrow{h}_i = LSTM(\overleftarrow{h}_{i+1}, x_i) \quad (1)$$

$$h_i = [\vec{h}_i \oplus \overleftarrow{h}_i] \quad (2)$$

where $\oplus$ is a concatenation operation. After encoding, two sequences of encoder hidden states will be obtained: context hidden states $h = (h_1, h_2, ..., h_m)$ and pattern guider hidden states $h' = (h^{'}_1, h^{'}_2, ..., h^{'}_n)$. Note that in our experiments, we used two separate LSTMs, one for encoding context sequence and one for encoding pattern guider sequence.

2) *Method Name Decoder:* The decoder is a single-layer unidirectional LSTM. For the decoding stage, at the time step t, the hidden state of the decoder is calculated by:

$$s_t = LSTM(s_{t-1}, y_{t-1}) \quad (3)$$

where s_{t1} means the previous hidden state of the decoder, y_{t-1} is the embedding of a word (while training, this is the (t-1)-th word embedding of ground-truth; while test, it is the previous word emitted by the decoder).

Then, at the t-th time step, NamPat focus and place more attention on the relevant tokens of the context sequence and the pattern guider sequence as needed. It is useful to introduce the attention mechanism into the method name generation model. We think different tokens of the method name are related to different parts of the method. For example, when the model generates the token "get" because it notices the token "return" in the method. Similarly, NamPat also can focus on some parts of the pattern guider when generating the method name. So, we calculate the weight of the tokens α in the context sequence and the pattern guider sequence, respectively according to the attention mechanism:

$$e^t_i = v^T(W_h h_i + W_s s_t) \quad (4)$$

$$a^t = softmat(e^t) \quad (5)$$

where v, W_h and W_s are learnable parameters. s_t is the output of the decoder at t-th time step. We calculate the attention weight a^t and $a^{t'}$ in the context hidden state sequence and the pattern guider hidden state sequence, respectively. Next, we use the attention weight distribution to calculate the weighted sum of the hidden states of the two encoders, and call them context vector h^*_t and pattern guider vector $h^{*'}_t$ respectively, and connect them together as pattern guider-context vector h^{gc}_t:

$$h^*_t = \sum\nolimits_i a^t_i h_i \quad (6)$$

$$h^{*'}_t = \sum\nolimits_i a^{t'}_i h^{'}_i \quad (7)$$

$$h^{gc}_t = [h^*_t \oplus h^{*'}_t] \quad (8)$$

where pattern guider-context vector h^{gc}_t can be seen as a fusion representation of the context information in the method and the naming pattern of the method name in the pattern guider.

Because NamPat is allowed to copy tokens from the input sequences and generate tokens from a fixed vocabulary. In addition, the generation probability $p_{gen} \in$ [0,1] at the time step t is calculated based on the pattern guider-context vector h^{gc}_t, the state s_t of the decoder and the input z_t of the decoder:

$$P_{gen} = \sigma(w^T_{h^gc} h^{gc}_t + w^T_s s_t + w^T_z z_t + b_{ptr}) \quad (9)$$

where $w_{h^{gc}}$, w_s, w_z and b_{ptr} are learnable parameters and is the sigmoid function. p_{gen} represents the probability that the model generates tokens from a fixed vocabulary, which are tokens observed from the training corpus. On the other hand, 1-p_{gen} indicates the probability of directly copying the token from the input sequences as the output. Finally, at this time step t, the probability of outputting token y_t is calculated as:

$$P(y) = p_{gen}P_{vocab}(y) + (1 - p_{gen}) \sum_{i:y_i=y} a_i^t \tag{10}$$

where the first part represents the probability of generating y from the fixed vocabulary, and the second part represents the probability of copying y from the input sequences. We want to replicate not only from the input method context sequence but also from our pattern guider token sequence. Therefore, the attention distribution here includes two parts: context token sequence and pattern guider token sequence. If y is an OOV token, the probability of token y generated from the fixed vocabulary is 0, then $p_{vocab}(y)$ is 0. That is, the model copies words based on the attention distribution of the input token sequences. This will enable the model to generate OOV tokens.

3)Loss Function: During training, the overall loss of the whole sequence is calculated as the average loss of each step, which is the negative log-likelihood of the target word y_t^* for that timestep:

$$loss = \frac{1}{T}\sum_{t=0}^{T}(-logP(y_t^*)) \tag{11}$$

V. EXPERIMENTAL SETUP

A. Dataset

To evaluate the performance of NamPat on the method name recommendation task, we tested it on the widely used Java dataset built by Nguyen et al. [12]. This dataset contains more than 10K Java projects, which are collected from the top-ranked public projects on GitHub. Following the same settings of Nguyen et al. [12], we randomly split the dataset into 9772 training and 450 testing projects, containing 17M training and 580K testing methods. In addition, we shuffle the corpus according to items rather than files, which avoids the unfair performance improvement caused by file-based shuffling [24].

B. Evaluation Metrics

To evaluate the quality of the method names generated by our model, we use the same metrics as previous works [11]–[13], which measured *Precision*, *Recall*, and *F-score* over case-insensitive tokens. Specifically, for a target method name t and a recommended method name r, its $precision(t, r)$, $recall(t, r)$, and $F - score(t, r)$ are calculated as:

$$precision(t, r) = \frac{|token(t) \cap token(r)|}{|token(r)|} \tag{12}$$

$$recall(t, r) = \frac{|token(t) \cap token(r)|}{|token(t)|} \tag{13}$$

$$F - score(t, r) = \frac{2 \times precision(t, r) \times recall(t, r)}{precision(t, r) + recall(t, r)} \tag{14}$$

where $token(n)$ return the tokens in the name n. The overall performances are computed as the average values of all samples in the dataset. In addition, we also use the additional metric *Exact Match Accuracy (EM Acc)*, in which the order of the tokens is also taken into consideration.

C. Implementation Details

NamPat is implemented based on PyTorch framework. We set the embedded dimensions of words to 150, LSTM hidden states dimensions to 400, and the batch size to 128. We use gradient clipping with a maximum gradient norm of 2 and use javalang to parse java codes and extract context information. In our experiment, we limit the sequence constituting the method context to 400 tokens, the pattern guider sequence to 6 tokens, and the length of the target method name to 6. In the testing phase, the method name is produced using beam search with beam size of 4. We use Adagrad with the learning rate of 0.15 and the initial accumulator value of 0.1 to train the model for about 1.3 million iterations (10 epochs). All experiments of our model are performed on an NVIDIA Titan V GPU with 12 GB memory.

D. Research Questions

To evaluate the effectiveness of our proposed approach, we conducted experiments to investigate the following research questions:

RQ3: How effective is NamPat compared with the state-of-the-art baselines?

To evaluate the performance of our proposed approach, we compare NamPat with the state-of-the-art approaches on the method name recommendation task.

RQ4: How does the pattern guider affect the NamPat performance?

We regard the name of the most similar method as a pattern guider, which is used to guide NamPat to generate a more appropriate method name. To analyze the impact of the pattern guider on NamPat effectiveness, we only use three local contexts of the method as prediction information to explore whether the performance of NamPat will be effected without the help of the pattern guider.

RQ5: How effective is NamPat on method names of different lengths?

We explored how many tokens developers usually used for naming methods. Then, to measure the effectiveness of NamPat on different lengths of method name, we perform NamPat under different lengths settings.

VI. Evaluation

A. RQ3: How effective is NamPat compared with the state-of-the-art baselines?

To answer this research question, we compared NamPat with the following state-of-the-art models of method name recommendation:

code2vec [11]: An attention-based neural model decomposes the code into a collection of paths in its AST and aggregates all of the path vectors into a single vector by the attention mechanism. The method name recommendation task is regarded as a classification task, and the representation vector of the method is used to predict a method name.

code2seq [10]: An extended approach of code2vec, which utilizes the seq2seq framework to represent a code snippet as a path set in its AST by using LSTMs, and uses the attention mechanism to select the relevant path when generating the method name subtokens.

MNire [12]: An RNN-based seq2seq approach uses the program entity names in the method body and the enclosing class name to recommend method names.

Cognac [13]: A context-guided method name recommendation approach recommends method names by using the local context of the method, the global context of the interactive method, and the probability of tokens in different contexts composing the method name.

As shown in Table III, the performance of NamPat outperforms all baseline models on all metrics. On the exact match accuracy, NamPat reaches 50.4%, which is relatively increase by 16.9%∼46.1% compared with the baselines. The higher exact match accuracy indicates the more method names recommended by NamPat completely matched the ground truth. The method names recommended by all baseline models, sometimes have a high F-score, but they do not achieve an exact match because their models usually can't learn the naming pattern of method names written by developers. For example, the ground truth is *"prepare update item statement"*, and the recommended name is *"prepare delete item statement"*. In this case, it gets a good F-score, but the exact match doesn't occur. In NamPat, the name of the most similar method is treated as the pattern guider, which is written by people with the naming pattern. Pattern guider will guide NamPat to recommend the method name according to the same naming pattern so that the recommended method name can better match the ground truth.

We note that both code2vec and code2seq represent methods by using the AST path. However, the structures of ASTs are usually large and deep due to the complexity of the methods. This complex representation leads to lower performance on all metrics than MNire, which uses the name of the program entity rather than the AST path for generating the method name [12]. These results prove that it is correct for our model to use the name of the program entity. Compared with all baselines, NamPat has the highest recall and precision. Specifically, NamPat improves the precision by 5.8%∼27.1%, the recall by 11.1%∼60.1%, and F-score by 8.5%∼43.9%, which means that the tokens in the method names generated by our model can cover much more target tokens than the other baselines. Through analysis, pattern guider plays an important role in NamPat, which has a strong correlation with the target method name and introduces more tokens that make up the target method name.

TABLE III
Method Name Recommendation Comparison

Approaches	Precision	Recall	F-score	EM Acc
code2vec	58.4%	44.6%	50.6%	38.4%
code2seq	67.4%	57.5%	61.9%	41.3%
MNire	70.1%	64.3%	67.1%	43.1%
Cognac	69.8%	55.8%	62.1%	34.5%
NamPat	**74.2%**	**71.4%**	**72.8%**	**50.4%**

TABLE IV
Impact of pattern guider on NamPat

Model	Precision	Recall	F-score	EM Acc
-Pattern Guider	67.6%	54.7%	60.5%	34.1%
NamPat	74.2%	71.4%	72.8%	50.4%

The performance of NamPat is higher than state-of-the-art approaches in all Metrics. Therefore, our model can sufficiently recommend method names that precisely describe the functionality of the method body.

B. RQ4: How does the pattern guider affect the NamPat performance?

In our model, we use three kinds of context information of methods and the name of the most similar methods as a pattern guider to generate method names. To explore the effectiveness of the pattern guider, we remove it from NamPat and only use the three context information as input to recommend the method name.

The experimental results are shown in Table IV. When we use the pattern guider, the performance of NamPat is greatly improved. Specifically, it increases precision from 67.6% to 74.2%, recall from 54.7% to 71.4%, F-score from 60.5% to 78.8%, and exact match accuracy from 34.1% to 50.4%. Among them, the larger improvement is achieved on recall and exact match accuracy. These results mean that it is difficult to generate method names with correct naming patterns only relying on the three local context information of the method (implementation context, interface context, and enclosing context). When NamPat uses the pattern guider, it can generate more accurate matching method names according to the naming pattern of the pattern guider and cover more target tokens.

C. RQ5: How effective is NamPat on method names of different lengths?

To measure the performance of NamPat on different settings of method name lengths, we first statistically analyze the

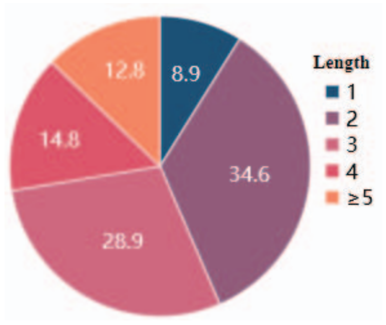


Fig. 3. Distribution of Method Name Length

TABLE V
PERFORMANCE OF NAMPAT ON DIFFERENT LENGTHS METHOD NAMES

	Len 1	Len 2	Len 3	Len 4	Len≥ 5
Precision	40.3%	68.1%	79.1%	83.1%	84.5%
Recall	70.0%	75.5%	76.2%	73.3%	61.3%
F-score	51.2%	71.6%	77.6%	77.1%	71.0%
EM Acc	48.1%	56.1%	61.3%	43.5%	17.1%

number of methods with different method name lengths in the test corpus. As shown in Figure 3, we put together the methods whose length is greater than or equal to five, which accounts for about 12.7% of the total methods in the test corpus. Meanwhile, we can also find that the method names with a length of one account for the minor proportion of 8.9%. Maybe it is difficult to describe the functionality of the method when the length of the method name is too short. The method names with the length of two and three have the largest number, accounting for 64.1% together, and almost two of every three methods have a name with a length of two or three. These results indicate that developers tend to use two or three words for method names.

As shown in Table V, we tested the performance of NamPat on method names of different lengths. NamPat has obtained the best performance on the method names with a length of three, which achieved an F-score of 77.6%, and exact match accuracy of 61.3%. As for the method names with a length of two, NamPat also achieved excellent performance, which reached an F-score of 71.6% and exact match accuracy of 56.1%. These results indicate that the method names generated by our model fit well with the naming habit of developers. When the method name length is one, the F-score of NamPat is only 51.2%, which proves that NamPat prefers to recommend detailed names rather than simple names. For method names with a length of greater than or equal to five, NamPat still has 71.0% F-score because the pattern guider provides a unique naming pattern and many predictable tokens. But exact match accuracy of NamPat only is 17.1%, which shows that generating the method name exactly matching the ground truth will become more difficult with the increase of the method name length.

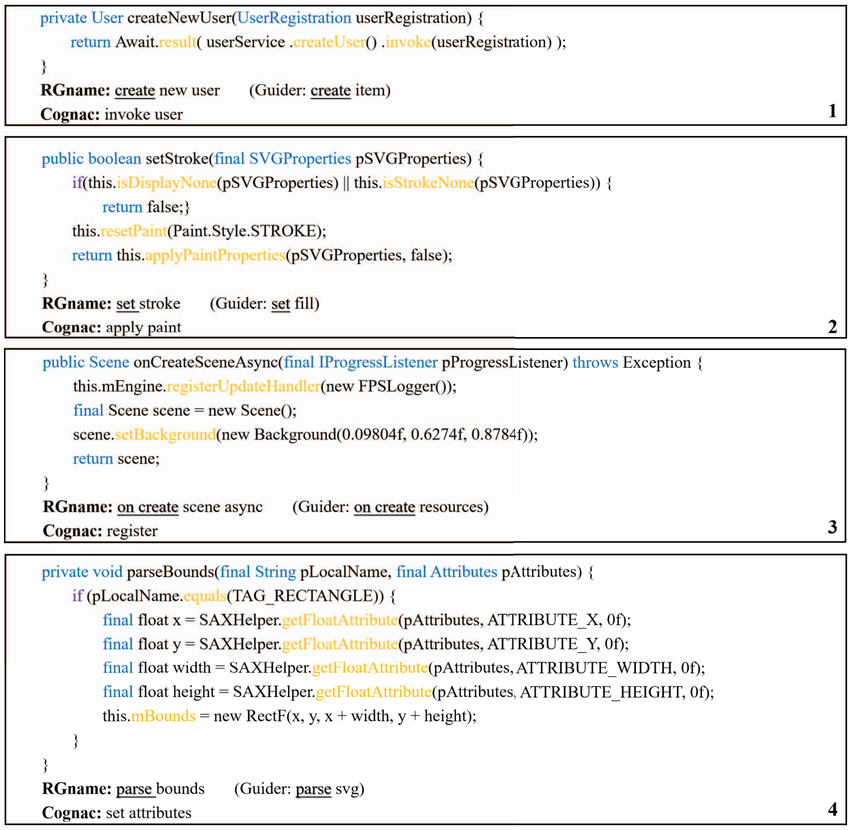
VII. DISCUSSION

A. Qualitative Analysis

As shown in Figure 4, we present four examples generated by NamPat and the state-of-the-art baseline Cognac from the test set. In cases 1 and 2, Cognac recommends the method names *"invoke user"* and *"apply paint"*, in which the tokens can be observed in the context of the methods, but the recommended names do not match the real semantics of the methods. In NamPat, with the help of the pattern guider, we get the naming pattern of the target method names *"create____"* and *"set____"* which help NamPat generate more meaningful method names. In addition, we can observe in case 3 that the naming pattern from the pattern guider is *"on create____"*. The naming pattern provides particular prediction information, that is, the tokens *"on"* and *"create"* appear in the pattern guider instead of being observed in the method contexts. In case 4, Cognac recommends the method name *"set attributes"*, which has the naming pattern *"set____"*. This does not conform to the semantics of the method. However, with the help of the pattern guider, NamPat can get the correct naming pattern *"parse____"*. The above examples show that the method names recommended by NamPat are close to the correct method names. Previous models tend to generate tokens observed in contexts, which sometimes fail to recommend meaningful method names.

B. Performance Enhancements from the Replication Mechanism

Previous studies such as the seq2seq model-based MNire can only generate tokens from their own fixed vocabularies but cannot copy tokens from the input sequences, which makes the models encounter OOV problems. So, the replication mechanism is used in our model to replicate tokens from the input sequences, which enables our model to process OOV tokens. However, this is not the main reason for the superior performance of our model. To verify this viewpoint, we remove the replication mechanism from NamPat by setting p_{gen} in the model to 1, so that our model can only generate tokens from the fixed vocabulary and not from the input sequence. We perform a performance evaluation on the same dataset, and the experimental results show that the performance of the model removing the replication mechanism reaches 72.1% (F-score), which is slightly lower than NamPat (72.8%) and much higher than MNire (67.1%) and Cognac (62.1%). In other words, the performance of NamPat is superior mainly from the information provided by the names of similar methods rather than the replication mechanism. Although we experiment in a large training set, the OOV problems always appear in our tasks. So, it is reasonable for us to add the replication mechanism to deal with OOV tokens. The above results show that introducing the name of the most similar method into our model NamPat makes it more effective than state-of-the-art approaches, and the replication mechanism makes the performance of NamPat better.



```
private User createNewUser(UserRegistration userRegistration) {
    return Await.result( userService .createUser() .invoke(userRegistration) );
}
RGname: create new user     (Guider: create item)
Cognac: invoke user                                                          1
```

```
public boolean setStroke(final SVGProperties pSVGProperties) {
    if(this.isDisplayNone(pSVGProperties) || this.isStrokeNone(pSVGProperties)) {
        return false;}
    this.resetPaint(Paint.Style.STROKE);
    return this.applyPaintProperties(pSVGProperties, false);
}
RGname: set stroke     (Guider: set fill)
Cognac: apply paint                                                          2
```

```
public Scene onCreateSceneAsync(final IProgressListener pProgressListener) throws Exception {
    this.mEngine.registerUpdateHandler(new FPSLogger());
    final Scene scene = new Scene();
    scene.setBackground(new Background(0.09804f, 0.6274f, 0.8784f));
    return scene;
}
RGname: on create scene async     (Guider: on create resources)
Cognac: register                                                             3
```

```
private void parseBounds(final String pLocalName, final Attributes pAttributes) {
    if (pLocalName.equals(TAG_RECTANGLE)) {
        final float x = SAXHelper.getFloatAttribute(pAttributes, ATTRIBUTE_X, 0f);
        final float y = SAXHelper.getFloatAttribute(pAttributes, ATTRIBUTE_Y, 0f);
        final float width = SAXHelper.getFloatAttribute(pAttributes, ATTRIBUTE_WIDTH, 0f);
        final float height = SAXHelper.getFloatAttribute(pAttributes, ATTRIBUTE_HEIGHT, 0f);
        this.mBounds = new RectF(x, y, x + width, y + height);
    }
}
RGname: parse bounds     (Guider: parse svg)
Cognac: set attributes                                                       4
```

Fig. 4. Examples of generated method names

C. Application Scenario

The inputs of our approach NamPat are the name of the most similar method and the contexts of methods. The first application scenario is that developers want to determine an appropriate name for an implemented method. Given an implemented method, NamPat can directly work to retrieve its naming pattern and combine it with context information to recommend an appropriate method name. This can be seen as a just-in-time method name recommendation. The second scenario is that developers determine whether the existing name of a method is consistent. In this scenario, NamPat can recommend an appropriate method name and compare it with the existing name to check the consistency.

D. Threats to Validity

One threat to validity is that we have evaluated our approach on only one dataset, which is the benchmark dataset for method name recommendation in previous work [11]–[13]. All projects in the dataset are high-quality and well-maintained projects selected from GitHub. Therefore, we consider that most method names are consistent with methods. In addition, we only focus on the Java programming language, and our large-scale empirical research is also done in the Java programming language. However, according to the principle of NamPat, it uses the name of the most similar method and the context information of the method, which is not limited to a specific programming language, and we will extend NamPat to other languages in future work.

Another threat to validity relates to the statistical metrics. We choose the statistical metrics (precision, recall, F-score, and exact matching) used in the previous works [11]–[13] to evaluate the performance of our approach. These evaluation metrics maybe not be truly reflected whether the recommended method names are helpful to developers in the actual development. Therefore, more manual verification is needed to confirm the usefulness of our approach.

VIII. Related Work

The method name recommendation task aims to recommend meaningful names that summarizes the key functionality of the method. This is similar to the code summarization task of generating good descriptions for the functionality of a method. Therefore, we introduce the related works in three aspects: method name recommendation, code summarization, and code representation.

A. Method Name Recommendation

There are two categories of approaches for method name recommendation. The first one is Information Retrieval (IR). Liu et al. [9] relies on the principle that methods with similar bodies have similar names. They retrieved similar methods and reused their method names for the recommendation. Jiang et al. [8] searches for methods with similar return types and parameters and derive method names based on heuristics. The main disadvantage of these IR-based methods is that they cannot generate new names that have not been seen in the training set by searching in the existing method names collection.

The other category is based on machine learning. Allamanis et al. [25] considered method names as the extreme summarization of source code. They proposed a convolution attention neural network, which convoluted the tokens sequence of the method body to obtain the structure information of the source code. Alamanis et al. [1] projected all the names in the method body and the tokens in the method name into the same vector space. Their model combined nearby tokens to compose a new method name. To use the structure information in the source code, Alon et al. [11] proposed Code2vec, which aggregated the paths from different leaf nodes to leaf nodes in the source code AST and used the attention mechanism to generate a code vector. Then Code2vec used the code vector to recommend the method name. Later, Alon et al. [10] proposed code2seq, which used the seq2seq model with attention to represent the component distribution vector of the source code and generate a series of words. MNire [12] used the seq2seq model with attention to recommending method names and explored the sub-tokens of method names appearing in the implementation context, interface context, and enclosing context of the target method. Wang et al. [13] found that MNire still has limitations in dealing with the methods having little content, So they introduced the caller and callee context for method name recommendation. They also used the frequency with which the method name token appears in different contexts as additional information for method name recommendation. Li et al. [26] also considered using more contexts, including the internal context, the caller and callee contexts, sibling context, enclosing context, and learning context representation to recommend method names.

B. Code Summarization

The code summarization task aims to generate brief descriptions of code automatically. High-quality code summaries also help developers understand the code. Most code summarization generation approaches used the seq2seq framework. Iyer et al. [27] used a Recurrent Neural Network (RNN) [28] with an attention mechanism to generate code summaries and achieved good results. Ahmad et al. [29] proposed a transformer-based approach to generate code summarization.

In addition, many works used the structure information of the code [30]–[32]. Hu et al. [30] proposed a neural model named DeepCom to utilize the structural information of code. Then, they converted the AST into a token sequence to generate summarization for Java methods. Fernandes et al. [31] construct a graph representation of code using AST and lexical information. Then, Liu et al. [32] further use more structural information. They combine diverse representations of the source code, including AST, Control Flow Graph(CFG), and Program Dependency Graph (PDG), into a joint code property graph to generate summarization.

C. Code Representation

Code representation learning is a hot research topic in software engineering. Existing code representation efforts represent code in three categories: (1) source code token sequence; (2) AST; and (3) Graph. Based on the token representation, the source code is tokenized as a token sequence, and each token is represented as a vector. Based on AST representation, AST is usually into a series of nodes by traversing. Based on the representation of a Graph, some work is represented as a Graph by adding edges to extending ASTs. And other works use CFG and DFG to represent source code. These learned code vectors then can be used for various SE tasks, such as method name recommendation [11], [12], code clone detection [33], [34], code search [35], code summarization [30], [36], etc.

IX. Conclusion

In this paper, we conduct a large-scale empirical analysis on 15M Jave methods, and the key findings are that: (1) method names contain many repetitive naming patterns, and the names of the two methods with similar body code usually have the same naming pattern; (2) the tokens composing the target method name can be observed in the name of the most similar method. The name of the most similar method can be utilized for better inferring the target method name. Motivated by our finding, we propose a naming pattern-based approach for method name recommendation, named NamPat. NamPat contains two modules. The naming pattern extraction module not only extracts the context information of the target method but also retrieves the most similar method and extracts its method name as a pattern guider. The method name generation module is a pointer generator network that learns the semantics of naming patterns and context information and combines them to recommend method names. We conducted extensive experiments on a widely-used Java dataset. The experimental results show that NamPat outperforms the state-of-the-art baselines. All the source code and data of this study can be found at: https://github.com/cqu-isse/NamPat.

Acknowledgment

This work was supported in part by the General Program of Chongqing Natural Science Foundation(cstc2021jcyj-msxmX0309), the National Key Research and Development Project (No.2018YFB2101200), the Chongqing Science and Technology Plan Projects (No. cstc2018jszx-cyztzxX0037), Key Project of Technology Innovation and Application Development of Chongqing (No. cstc2019jscx-mbdxX0020), the National Natural Science Foundation of China (No. 62102054), and the Open Foundation of Key Laboratory of Dependable Service Computing in Cyber Physical Society, Ministry of Education of China (CPSDSC202004).

References

[1] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 38–49.

[2] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. Adesope, "The effect of poor source code lexicon and readability on developers' cognitive load," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 286–28 610.

[3] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, and M. Beigl, "Descriptive compound identifier names improve source code comprehension," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 31–3109.

[4] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 31–35.

[5] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The effect of lexicon bad smells on concept location in source code," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. Ieee, 2011, pp. 125–134.

[6] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2018.

[7] E. W. Høst and B. M. Østvold, "Debugging method names," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 294–317.

[8] L. Jiang, H. Liu, and H. Jiang, "Machine learning based automated method name recommendation: How far are we," in *Proceedings of the 34th ACM/IEEE International Conference on Automated Software Engineering (ASE'19). IEEE CS*, 2019.

[9] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon, "Learning to spot and refactor inconsistent method names," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1–12.

[10] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[11] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[12] S. Nguyen, H. Phan, T. Le, and T. N. Nguyen, "Suggesting natural method names to check name consistencies," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1372–1384.

[13] S. Wang, M. Wen, B. Lin, and X. Mao, "Lightweight global and local contexts guided method name recommendation with prior knowledge," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 741–753.

[14] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 404–419, 2018.

[15] P. Bielik, V. Raychev, and M. Vechev, "Phog: probabilistic model for code," in *International Conference on Machine Learning*. PMLR, 2016, pp. 2933–2942.

[16] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from" big code"," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 111–124, 2015.

[17] E. Hatcher and O. Gospodnetic, *Lucene in action (in action series)*. Manning Publications Co., 2004.

[18] J. Li, Y. Li, G. Li, X. Hu, X. Xia, and Z. Jin, "Editsum: A retrieve-and-edit framework for source code summarization," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 155–166.

[19] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1385–1397.

[20] S. Robertson and H. Zaragoza, *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc, 2009.

[21] S. Qaiser and R. Ali, "Text mining: use of tf-idf to examine the relevance of words to documents," *International Journal of Computer Applications*, vol. 181, no. 1, pp. 25–29, 2018.

[22] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 1073–1083.

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

[24] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018.

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

[26] Y. Li, S. Wang, and T. N. Nguyen, "A context-based automated approach for method name consistency checking and suggestion," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 574–586.

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

[28] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model." in *Interspeech*, vol. 2, no. 3. Makuhari, 2010, pp. 1045–1048.

[29] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.

[30] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010.

[31] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=H1ersoRqtm

[32] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Automatic code summarization via multi-dimensional semantic fusing in gnn," *arXiv preprint arXiv:2006.05405*, 2020.

[33] K. W. Nafi, T. S. Kar, B. Roy, C. K. Roy, and K. A. Schneider, "Clcdsa: cross language code clone detection using syntactical features and api documentation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1026–1037.

[34] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[35] L. Xu, H. Yang, C. Liu, J. Shuai, M. Yan, Y. Lei, and Z. Xu, "Two-stage attention-based model for code search with textual and structural features," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 342–353.

[36] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," *Advances in neural information processing systems*, vol. 32, 2019.