Learning to Recommend Method Names with Global Context

Fang Liu
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
liufang816@pku.edu.cn

Ge Li*
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
lige@pku.edu.cn

Zhiyi Fu
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
ypfzy@pku.edu.cn

Shuai Lu
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
lushuai96@pku.edu.cn

Yiyang Hao
Silicon Heart Tech Co., Ltd
Beijing, China
haoyiyang@nnthink.com

Zhi Jin*
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
zhijin@pku.edu.cn

ABSTRACT

In programming, the names for the program entities, especially for the methods, are the intuitive characteristic for understanding the functionality of the code. To ensure the readability and maintainability of the programs, method names should be named properly. Specifically, the names should be meaningful and consistent with other names used in related contexts in their codebase. In recent years, many automated approaches are proposed to suggest consistent names for methods, among which neural machine translation (NMT) based models are widely used and have achieved state-of-the-art results. However, these NMT-based models mainly focus on extracting the code-specific features from the method body or the surrounding methods, the project-specific context and documentation of the target method are ignored. We conduct a statistical analysis to explore the relationship between the method names and their contexts. Based on the statistical results, we propose GTNM, a Global Transformer-based Neural Model for method name suggestion, which considers the local context, the project-specific context, and the documentation of the method simultaneously. Experimental results on java methods show that our model can outperform the state-of-the-art results by a large margin on method name suggestion, demonstrating the effectiveness of our proposed model.

CCS CONCEPTS

• **Software and its engineering**; • **Computing methodologies** → **Artificial intelligence**;

KEYWORDS

method name recommendation, global context, deep learning

*Corresponding authors.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510154

ACM Reference Format:
Fang Liu, Ge Li, Zhiyi Fu, Shuai Lu, Yiyang Hao, and Zhi Jin. 2022. Learning to Recommend Method Names with Global Context. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510154

1 INTRODUCTION

During programming, developers must name variables, functions, parameters, *etc.* The appropriateness of a name changes over time during the software evolution. For example, a good function name can degrade into a poor one when the semantics of the function change or the function is used in a new context. Poor names make programs harder to understand and maintain [9, 10, 17, 22, 25, 39], leading to misuses and defects [1, 2, 8, 13]. Finding consistent names for program constructs has always been a cynosure in the software industry.

Methods are the most minor named units for indicating the program behavior in most programming languages [18], thus they are particularly important [12, 30, 32]. Meaningful and conventional method names are vital for developers to understand the behavior of programs or APIs. Once the name of a method is decided, it is laborious to change, especially when used for an API [4]. The results from an investigation in Liu et al. [28] indicate that among the change history in projects, developers usually change the method names without any change to the corresponding body code in many cases, which suggests that programmers strive to choose meaningful and appropriate method names, i.e., more consistent with other names in the same project or the codebase. Especially when collaborating, they need to obey a project's coding conventions.

In recent years, researchers have proposed automated approaches for suggesting consistent names for those methods. Based on the intuition that two methods implemented with similar code in their body code are likely to be named similarly, Liu et al. [28] proposed an IR-based approach to detect and rename inconsistent method names. They identify the inconsistent method names by comparing the names retrieved from the method body vector space with those retrieved from the method name vector space. For the inconsistent names, their model recommends the potentially consistent names by referring to the names of similarly implemented methods. However, in many cases, even the methods with similar body code can be named differently because they might belong to different projects

and have different semantics. Besides, by retrieving names from similar methods, the model cannot suggest neologisms. Allamanis et al. [5] proposed a convolutional attentional network to extract local time-invariant and long-range topical attention features in the method body to suggest names for methods. To leverage the syntactic structure of programming languages, Code2vec [7] and Code2seq [6] represent the method body as a set of compositional abstract syntax tree (AST) paths and use the path representation to predict the method's name. Nguyen et al. [34] proposed MNire, a simple but effective approach to recommend a method name and detect method name inconsistencies. They treated the method name generation task as an abstractive summarization of the tokens of the program entities' names in the method body and the enclosing class name. Li et al. [24] developed DeepName, a context-based approach for method name consistency checking and suggestion. They extract the features from four contexts: the internal context, the caller and callee contexts, sibling context, and enclosing context.

The above state-of-the-art models mainly focus on exploiting code-specific features from the method body or the surrounding methods in the same program file, which can be considered as local contexts of a method. However, the information of the whole project (global context) is ignored in these models. For example, the documentation of the method can describe the method's functionality and the role it plays in the project. Besides, there also exist nested scopes for project, where a source code file can have references to other files of the same projects. Thus, the contexts from other program files which are imported by the file where the target method in are also helpful in understanding the methods. Intuitively, these contexts are of great importance for method name recommendation, especially for the methods which have little content in the body, but with sufficient global contexts. A method does not exist in isolation, a large number of associations can be found among the project-specific contexts and the documentation: (1) The functionality and naming convention of a method can be better understood when more contextual features are provided. (2) There might be many possible names that can match the semantic of the method. By referring to the global contextual information, the solution space of the method names can be narrowed. Thus, when recommending a method name, it is necessary to refer to the global contexts. It can help in following situations: when the method is first created, existing global context can be accessed for suggesting a proper name for it; during the code refinement, the global context can be used to suggest an alternative name if the current name is inconsistent.

To verify our intuition, we first conducted a statistical analysis to learn the relation between the method names and their contexts of different levels. Based on the statistical analysis results, we propose **GTNM**, a novel **G**lobal **T**ransformer-based **N**eural **M**odel for method name suggestion, aiming at generating meaningful and consistent names for methods. We treat the method name suggesting task as the abstractive text summarization, where *the tokens from the contexts of different levels* are considered as input, and *the sub-tokens in the method's name* is considered as the target summary of input sequences. We use the attention mechanism to allow the model to attend to different contexts during the decoding process.

The main contribution of our model can be summarized as follows:

- We conduct a statistical analysis to explore the relationship between the method names and their contexts of different levels.
- We propose a novel global approach for method name suggestion, which considers the local context, the project-level context, and the documentation of the method simultaneously.
- We conduct extensive experiments to evaluate our approach on the large-scale datasets of Java methods. The experimental results show that our model substantially improves the performance of the previous approaches on suggesting method names.

2 MOTIVATING EXAMPLE AND STATISTICAL ANALYSIS

According to Nguyen et al. [34], the principle of naturalness of software [16] also holds for the tokens composing the names of program entities. Specifically, tokens are repetitive and occur in regularity, where the repetitiveness can be captured by statistical models trained on a large code corpus. Therefore, the tokens composing the names of program entities can reflect the semantic and functionality of the code snippets. Based on this evidence, most previous work mainly considers the associations among the tokens of the method names and the tokens in the method body (local context). However, only considering the local context is not sufficient. We assume that the project-specific context can better reflect the role that the target method plays in the whole project. For example, the methods in the same file with the target method (we call them in-file contextual methods) and the methods in other program files of the same project that are imported by the file where the target method locates (we call them cross-file contextual methods). Besides, the documentation of the method also plays an important role in recommending the method names. We present several java method examples to illustrate the associations among method names and the project-specific and documentation contexts in Section 2.2, appearing as the token overlapping. Based on those observations, we conduct a statistical analysis to explore the relationship between the method names and the contexts of different levels in Section 2.3, i.e., local context, project-specific context, and documentation context.

2.1 Definitions

Firstly, we give a brief definition of tokens, local context, project-specific context, and documentation context.

Definition of Tokens. For programs, we parse the program to AST and extract entities (method names, identifiers, parameters, return-types) from AST. Then we split the entities following camelcase and underscore naming conventions, and lowercase the entities to get tokens. For documentation, we extract the first sentence in Javadoc by deleting the punctuations. Then we split the sentence with space to get words and lowercase the words to get tokens.

Definition of Local Context. Local-context contains the program entities in the method signature and body, including parameters, return type, and identifiers.

Definition of Project-specific Context. Project-specific context is supposed to reflect the target method's role in the whole project

and the naming styles. We argue that the methods in the same file with the target method (we call them in-file contextual methods) and the methods in other program files of the same project that are imported by the file where the target method (we call them cross-file contextual methods) in can provide the above information. We consider the name of the contextual methods as the project-specific context.

Definition of Documentation Context. The first sentence of the code documentation is informative, and many code summarization approaches use it as a code summary [19, 23, 43]. Following them, we use the tokens in the first sentence as the documentation context.

2.2 Motivating Example

1. The project-specific context might contain the entities that can provide semantic information for the target method name recommendation. In Code 1, the names of the third method (getMaxValue) do not describe the functionality of the methods well. When changing it into a more precise name that contains the project-related entity names (getMaximumResourceCapability), only referring to the method body is not enough. If the (in-file) project-level contextual information, i.e., other methods in the same file, can be accessed, we can easily realize that the method is related to the *resource capability* and make correct revisions.

```
public Resource getClusterResource() {
    return clusterResource;
}
public Resource getMinimumResourceCapability() {
    return minimumAllocation;
}
// consistent name: getMaximumResourceCapability
public Resource getMaxValue() {
    return maximumAllocation;
}
```

Code 1: Project-specific context contains the entities that can provide semantic information

2. The project-specific contextual information can imply the logic and the functionality of the project, which will reflect the role the target method plays in the project. In Code2, these methods are related to the window events, including the keypress events or trackpad touch events. By accessing the (in-file) project-level context, the functionality of the whole project and the role of the target method can be better understood, thus offering more knowledge for recommending meaningful method name.

```
public boolean touchDown (InputEvent event, float x, float y, int
      pointer, int button) {
  ...
}
public void touchUp (InputEvent event, float x, float y, int
      pointer, int button) {
  ...
}
public boolean keyDown (InputEvent event, int keycode) {
  return isModal;
}
public boolean keyUp (InputEvent event, int keycode) {
  return isModal;
}
```

Code 2: project-specific contexts imply the logic and the functionality of the project.

3. There might be many semantically consistent names that can reflect the function of a specific method. We can narrow the solution space and suggest a consistent and conventional method name by referring to the project-specific contextual information. Both of the two methods in Code3 indicate that some errors are encountered. However, different verbs are used in the names ("Encountered" and "Occured"), and they are synonyms. Although these two names are both semantically correct, they are not consistent. When refactoring the second method name "serverErrorOccured" into a name that is consistent with the contextual methods, we can use the verb "Encountered" to replace "Occured" by referring to the previous method name "clientErrorEncountered". This suggests that with the help of the project-level context, we can choose the candidate names from a smaller and specific solution space.

```
public void clientErrorEncountered() {
  clientErrors.incr();
}
// consistent name: serverErrorEncountered
public void serverErrorOccured() {
  serverErrors.incr();
}
```

Code 3: Semantically consistent names

4. Cross-file project-specific context can provide extra information when the in-file context is less informative. In code4, the AccountActivity class inherits from BaseActivity class, thus the methods of the parent class BaseActivity might be overridden in AccountActivity class, for example, getLayoutRes(), onCreateActivity(), etc. The program file where the BaseActivity class defined is imported at the beginning of the file. Thus, we can extract the methods defined in the BaseActivity class by considering the cross-file project-specific contexts. Thus, when predicting the method name for the methods in AccountActivity class, the methods defined in its parent class can be accessed, which are helpful for the cases where the in-file context is less informative for inferring the method name.

```
[AccountActivity.java]
...
import com.github.airsaid.accountbook.base.BaseActivity;
...
public class AccountActivity extends BaseActivity {

    @Override
    public int getLayoutRes() {
        return R.layout.activity_account;
    }
    @Override
    public void onCreateActivity(@Nullable Bundle
          savedInstanceState) {
        Account account = getIntent().getParcelableExtra(
              AppConstants.EXTRA_DATA);
        ...
    }
    ...
}
-----------------------------------------------------------------
[BaseActivity.java]
public abstract class BaseActivity extends SlideBackActivity {
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
    ...
    }
    ...
    public abstract int getLayoutRes();
    public abstract void onCreateActivity(@Nullable Bundle
          savedInstanceState);
}
```

Code 4: Cross-file project-specific context can provide extra information when the in-file context is less informative

5. The documentation can also provide rich information about the methods, which will help for suggesting method names. In Code5, the body code of these methods looks similar, and all of them cannot offer enough information for suggesting the method name. The

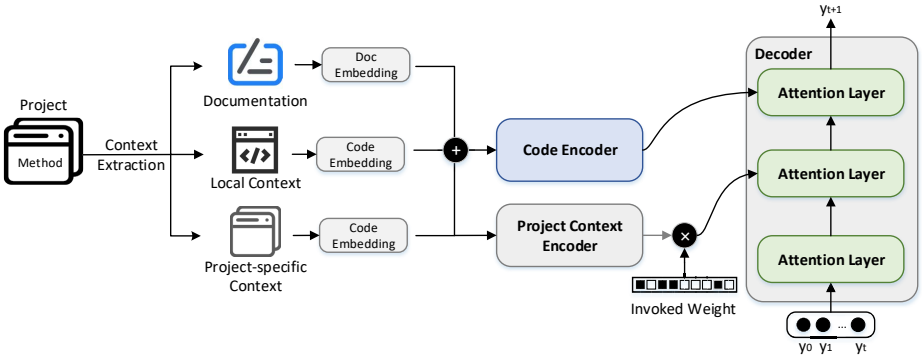

Figure 1: The overall framework of GTNM.

documentation of the methods contains useful information that can reflect the functionality of the methods, thus being helpful for the method name recommendation. When predicting the name for the first method, the documentation can provide a useful indication.

```
/**
 * Used to retrieve the plugin tool's descriptive name. */
// consistent name: getDescriptiveName
@Override
public String getName() {
    return "Remove Spurs (prunning)";
}
/**
 * Used to retrieve a short description of what the plugin tool
      does. */
@Override
public String getToolDescription() {
    return "Removes the spurs (prunning operation) from a Boolean
        image.";
}
```

Code 5: The documentation can provide rich information about the methods.

2.3 Statistical Analysis

Based on the above observations, we conduct a statistical analysis to explore the relationships between the method names and their contexts by computing the percentage of their token sharing. For the analysis, we used the java programs in the Java-small dataset used in Alon et al. [6]. The dataset contains 11 high quality open-source java projects, which is a benchmark dataset for method name suggetstion task. It contains about 700K Java method examples. Thus, we use this dataset to conduct the statistical analysis to explore the relationships between the method names and their contexts. The statistical results in this analysis can be expected in a good project where most of the names are consistent.

For local context, we found that the tokens of 67.47% of the method names can be found in the identifiers, and 35.64% can be found in the return type and parameters. For Project-specific context, we found that the tokens of 85.98% of the method names can be found in the names of its in-file contextual methods, and the tokens of 53.83% of the method names can be found in the names of its cross-file contextual methods. For the documentation context, we found that the tokens of 55.98% of the method names can be found in its documentation. There exists overlapping among different contexts, for example, the subtokens of the method name can appear in both local and documentation contexts. Thus, the sum of these numbers is not 100%. Besides, 10.87% of the method names cannot found in the body, but occur in the names of its project-specific context (in- and cross-file contextual methods). These results demonstrate that developers always refer to the project-specific context when naming the methods. Thus, project-specific context also contains essential information for method name recommendation, which should be carefully considered.

3 PROPOSED MODEL

3.1 Overview

In this work, we propose GTNM, a global Transformer-based Neural Model for method name recommendation aiming at generating meaningful and consistent method names. The overall architecture of our approach is shown in Figure 1. To fully utilize the contextual information of a method, we firstly extract context from three different levels given the target method and the project, including the local context, project-specific context, and documentation context. We employ a transformer-based seq2seq framework [41] to generate the method name. Specifically, we build corresponding encoders to encode the contexts into vector representations. The decoder generates the target method name by sequentially predicting the probability of the subtokens y_{t+1} in the method name based on the contextual representations produced by the encoders, and the previous predicted subtokens $y1, y2, ..., y_t$. We use the attention mechanism to allow the model to attend to different contexts during the decoding process.

3.2 Context Extraction

We extract the contexts of three different levels for generating meaningful and consistent names for the method, including local context, project-specific context, and documentation. Figure 2 shows an example of the contexts for the Java method "getElement".
Local Context Extraction According to the results of our statistical analysis and to represent the method body succinctly, we extract the following code entities as the local contexts for the method: (1) identifiers; (2) parameters; (3) return type. We tokenized each of the names from the local contexts following camelcase and underscore naming conventions, then normalized the tokens to lowercase. Finally, all the subtokens are concatenated in the order that they occurred in the source code to form the sequential representation of the local feature.

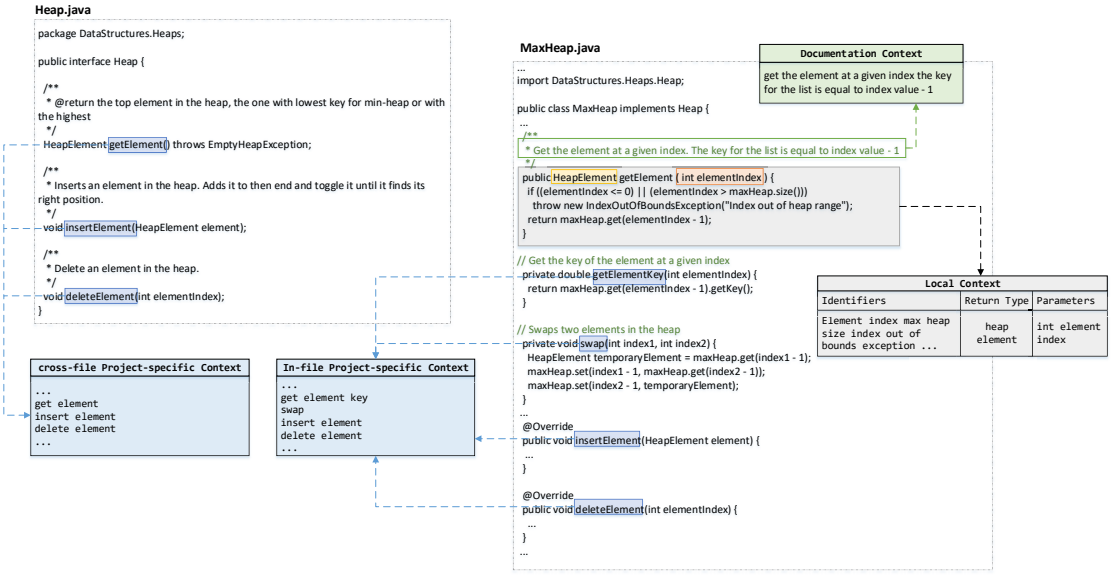

Figure 2: Different levels of contexts for method name suggestion.

Project-specific Context Extraction We define the project-specific context of one method as its in-file methods (other methods in the same file with the target method) and cross-file contextual methods (methods in the files imported by the file containing the target method). For simplicity and efficiency, we extract the name of the contextual methods as the project-specific context. Then we perform a similar process to these names as to local context. The concatenation of the lower-cased subtokens serves as the representation of the project-specific feature.

Documentation Context Extraction For each method with a comment, to get its documentation context, we extract the first sentence that appeared in its Javadoc description since it typically describes the functionalities of the method[1]. Then we delete the punctuations and split the sentence with space to get words and lowercase the words. All the words are concatenated to form the documentation context.

3.3 Global Transformer-based Neural Model

We use a transformer-based model to generate the method name, which leverages the self-attention mechanism and can capture rich semantic dependencies. The Transformer consists of stacked self-attention and point-wise, fully connected layers. The multi-head attention mechanism is performed in the self-attention layers. In each attention head, given the input vectors $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_n)$, the output vectors $\boldsymbol{o} = (\boldsymbol{o}_1, \boldsymbol{o}_2, ..., \boldsymbol{o}_n)$ is computed as:

$$\begin{aligned} \boldsymbol{o}_i &= \sum_{j=1}^{n} \alpha_{ij}(\boldsymbol{x}_j \boldsymbol{W}^V) \\ \alpha_{ij} &= \frac{\exp(e_{ij})}{\sum_{k=1}^{n} \exp(e_{ik})} \\ e_{ij} &= \frac{\boldsymbol{x}_i \boldsymbol{W}^Q (\boldsymbol{x}_j \boldsymbol{W}^K)^T}{\sqrt{d_k}} \end{aligned} \tag{1}$$

where $\boldsymbol{W}^Q, \boldsymbol{W}^K \in \mathbb{R}^{d_{model} \times d_k}, \boldsymbol{W}^V \in \mathbb{R}^{d_{model} \times d_v}$ are the trainable parameters that are unique per layer and per attention head. Then the outputs of all the heads are concatenated to produce the final output of the self-attention layer.

After the attention layers of both encoder and decoder, a fully connected feed-forward network is employed:

$$FFN(\boldsymbol{x}) = max(0, \boldsymbol{x}\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2 \tag{2}$$

where $\boldsymbol{W}_1 \in \mathbb{R}^{d_{model} \times 4d_{model}}, \boldsymbol{W}_2 \in \mathbb{R}^{4d_{model} \times d_{model}}, \boldsymbol{b}_1 \in \mathbb{R}^{4d_{model}}$, $\boldsymbol{b}_2 \in \mathbb{R}^{d_{model}}$ are the trainable parameters.

Encoders. We build a Code Encoder to encode the whole context x including the local context, project-specific context, and documentation for the method name generation, and build an extra Project Context Encoder to encode the project context x_{pro} for enhancing the attention to the project-specific context.

i) Code Encoder. The local context, project-specific context and the documentation context are first embedded into vectors $\boldsymbol{x}_{loc}$, $\boldsymbol{x}_{pro}$, $\boldsymbol{x}_{doc}$, then these vectors are concatenated to form the representation of the whole contexts $\boldsymbol{x} = concat(\boldsymbol{x}_{loc}, \boldsymbol{x}_{pro}, \boldsymbol{x}_{doc})$, where $|\boldsymbol{x}| = |\boldsymbol{x}_{loc}| + |\boldsymbol{x}_{pro}| + |\boldsymbol{x}_{doc}|$. Then we employ transformer-based encoder to encode $\boldsymbol{x}$ into hidden representation $\boldsymbol{h} = (\boldsymbol{h}_1, \boldsymbol{h}_2, ..., \boldsymbol{h}_{|\boldsymbol{x}|})$.

ii) Project-specific Encoder. To increase the attention for the project-specific context, especially for the method names where the target method invoked, we build a Project-specific Encoder to encode the project-specific context $\boldsymbol{x}_{pro}$ into hidden representation $\boldsymbol{h}_{pro} = (\boldsymbol{h}_1^{pro}, \boldsymbol{h}_2^{pro}, ..., \boldsymbol{h}_{|\boldsymbol{x}_{pro}|}^{pro})$. We use a mask vector $\boldsymbol{M} \in \mathbb{R}^{|\boldsymbol{x}_{pro}|}$ to record the methods that are invoked by the local context. M_i is 1 if the i-th method in the project-specific context is invoked by the local context else is 0.

Intuitively, the methods in the project-specific context invoked by the local context are more important and relative to the target method. Thus we give these methods more attention by multiplying the invoked weight $\boldsymbol{w}$ on the project-specific hidden vector $\boldsymbol{h}_{\boldsymbol{pro}}$ to

[1] http://www.oracle.com/technetwork/articles/java/index-137868.html

Table 1: Statistics of the datasets.

	Train	Validation	Test
Files	1,700,000	393,327	61,000
Methods	18,230,509	4,283,580	636,816
Methods with doc	4,264,852	964,078	143,913

produce the final project-specific hidden vector $\tilde{\boldsymbol{h}}_{pro}$:

$$\begin{aligned} \boldsymbol{w} &= softmax(1 + \boldsymbol{M}) \\ \tilde{\boldsymbol{h}}_{pro} &= \boldsymbol{w} \otimes \boldsymbol{h}_{pro} \end{aligned} \tag{3}$$

where $\otimes$ is the element-wise production operation.

Decoder. The decoder aims to generate the target method name by sequentially predicting the subtoken y_{t+1} conditioned on the context vectors $\boldsymbol{h}$ and $\tilde{\boldsymbol{h}}_{pro}$, and the previous generated subtokens $\boldsymbol{y}_{1:t}$:

$$\begin{aligned} p(y_{t+1}) &= \text{softmax}(FFN(\boldsymbol{dec}_2)) \\ \boldsymbol{dec}_2 &= \text{Attention-Layer3}(\boldsymbol{h}, \boldsymbol{dec}_1) \\ \boldsymbol{dec}_1 &= \text{Attention-Layer2}(\tilde{\boldsymbol{h}}_{pro}, \boldsymbol{dec}) \\ \boldsymbol{dec} &= \text{Attention-Layer1}(\boldsymbol{y}_{1:t}) \end{aligned} \tag{4}$$

where the first attention layer performs multi-head attention over the decoder input $y_{1:t}$ to produce the hidden representation $\boldsymbol{dec}$. Then the second attention layer performs multi-head attention over the weighted project-specific hidden vector $\tilde{\boldsymbol{h}}_{pro}$ to produce the hidden representation $\boldsymbol{dec}_1$, which models the dependency between the decoder input and the project-specific context. The last attention layer performs multi-head attention over the whole context hidden vector $\boldsymbol{h}$ to produce the final hidden representation $\boldsymbol{dec}_2$, which models the dependency between the decoder input, project-specific context, and the whole context. Then the final hidden representation is fed into a fully connected feed-forward network and softmax layer to produce the probability of the next subtoken y_{t+1} for the target method name.

Training. To train the network, we adopt cross-entropy loss between the predicted distribution $\boldsymbol{q}$ and the "true" distribution $\boldsymbol{p}$, which is computed as:

$$H(\boldsymbol{p}||\boldsymbol{q}) = -\sum_{y \in Y} p(y) \log q(y) = -\log q(y_{true}) \tag{5}$$

where y_{true} is the target name. Since p will assign value of 1 to the actual label in the training example and 0 otherwise, the cross-entropy loss for a example is equivalent to the negative log-likelihood of the true label. As $q(y_{true})$ tends to 1, the loss approaches zero. The smaller $q(y_{true})$ goes, the greater the loss becomes. Thus, minimizing this loss is equivalent to maximizing the log-likelihood that the model assigns to the true labels.

4 EXPERIMENTAL SETUP

4.1 Datasets

We train and evaluate GTNM on Java programs following MNire [34] and Code2vec [7]. Nguyen et al. [34] provide the list of java repositories, which contains 10K top-ranked, public Java projects on GitHub. They used the same setting as in code2vec to shuffle files in all the projects and split them into 1.7M training and 61K testing files. Following their setting, we download the repositories they provide and follow the same way to build the dataset. After data processing, the detailed data information is shown in Table 1.

Table 2: Statistics of contexts and target name lengths.

	Avg	Med
In-file Contextual Method	1399	68
Cross-file Contextual Method	197	80
Variables	23	7
Parameter and return type	3	3
Target Names	3	2

4.2 Metrics

To evaluate the quality of the generated method name, we adopted the metrics used by previous works [6, 7, 34], which measured *Precision*, *Recall*, and *F-score* over sub-tokens. Specifically, for the pair of the target method name t and the predicted name p, the $precision(t, p)$, $recall(t, p)$, and $F1(t, p)$ score are computed as:

$$\begin{aligned} precision(t, p) &= \frac{|\text{subtoken(t)}| \cap |\text{subtoken(p)}|}{|\text{subtoken(p)}|} \\ recall(t, p) &= \frac{|\text{subtoken(t)}| \cap |\text{subtoken(p)}|}{|\text{subtoken(t)}|} \\ F1(t, p) &= \frac{2 \times precision(t, p) \times recall(t, p)}{precision(t, p) + recall(t, p)} \end{aligned} \tag{6}$$

where subtoken(n) return the subtokens in the name n. Precision, Recall, and F-score of the set of the suggested names are defined as the average ones on all samples. Besides, we also measure the *Exact Match Accuracy (EM Acc)*, in which the order of the subtokens are also taken into consideration.

4.3 Implementation Details

We use Transformer with 6 layers, hidden size 512, and 8 attention heads for both encoders and decoders. The inner hidden size of the feed-forward layer is 2048. We use javalang[2] to parse the java code to extract the contexts. The details of different contexts and target names (subtoken) lengths are shown in Table 2.

In our experiments, we set the in-file project-specific context length to 30, the cross-file project-specific context length to 30, the local context length to 55 (variable length (50) + parameter and return type length (5)), the documentation context length to 10. And the maximum target name length is set to 5 [3]. We use the same vocabulary for the input source code and the target method name and build another vocabulary for the documentation context. The vocabulary size for the source code is set to 20,000, and the vocabulary size for documentation is set to 10,000. The out-of-vocabulary tokens are replaced by ⟨UNK⟩. To demonstrate the effectiveness of the cross-file project-specific context, we conduct experiments under the cross-project setting where the programs used in the training and test process are from different projects. Since more contexts can be accessed, we assume that we can use fewer programs to train the model. To verify the assumption, we train the

[2] https://github.com/c2nes/javalang

[3] we examined model's performance with different context length settings, the setting that can achieve the best results were used for the final training

Table 3: Method name recommendation comparison results.

Model	Precision	Recall	F1	EM Acc
code2vec[7]	51.93%	39.85%	45.10%	35.59%
code2seq[6]	68.41%	60.75%	64.36%	41.50%
MNire[34]	70.10%	64.30%	67.10%	43.10%
DeepName[24]	73.60%	71.90%	72.70%	44.30%
GTNM	**77.01%**	**74.15%**	**75.60%**	**62.01%**

model using a subset of the whole training dataset and compare it with the results without using the cross-file project-specific context. The detailed results are presented in 5.3.

We use Adam with the learning rate of 3e-4, linear learning rate warmup schedule over the first 4,000 steps to train the model for 20 epochs. We use a dropout probability of 0.3 on all layers. Our model is trained on one Tesla V100 GPU with 16GB memory.

5 RESEARCH QUESTIONS AND RESULTS

To evaluate our proposed approach, in this section, we conduct experiments to investigate the following research questions:

5.1 RQ1: Comparison against state-of-the-art models

We compare GTNM with the following state-of-the-art method name suggestion models:

1) code2vec [7]: an attention-based neural model, which performs attention mechanism over AST paths and aggregates all of the path vector representations into a single vector. They considered the method name prediction as a classification problem and predicted a method's name from the vector representation of its body.

2) code2seq [6]: an extended approach of code2vec, which employs seq2seq framework to represent AST paths of the method body node-by-node using LSTMs and then attend to them while generating the target subtokens of the method name.

3) MNire [34]: an RNN-based seq2seq model approach to suggest a method name based on the program entities' names in the method body and the enclosing class name.

4) DeepName [24]: an RNN-based approach for method name consistency checking and suggestion, using both internal and interaction contexts for method name consistency checking and suggestion, which achieves the state-of-the-art results on java method name suggestion task.

The first three baselines do not use the cross-file project-specific context for the method name suggestion. To make the comparison fair, we do not use the cross-file project context in this experiment. We use the same dataset as MNire and DeepName to train our model. For code2vec and code2seq, we download their publicly available source code and train their model on the same datasets. The results are shown in Table 3. Among these baselines, code2vec and code2seq only use the context in the method body to predict the method names. MNire utilizes the enclosing class (where the method is in) contexts, and DeepName further considers the interaction context and sibling context, which might appear in other program files.

The results show that GTNM outperforms all the baseline models on all the metrics by a large margin, especially on the exact match accuracy. The higher exact match accuracy indicates the generated name is more close to the ground truth. Table 4 shows two examples where the exact-match didn't occur but F1 was good. In the first case, the semantics of two names are reverse although they shared most of the sub-tokens with a high F1 score. Thus, exact match accuracy can evaluate the generated name more precisely, which plays a crucial role in method name suggestion. There are 32% of the test methods where exact match is not satisfied but F1 $\geq$ 0.5. Among these cases, only 2.32% of methods have the same subtoken set between generated names and target name.

Table 4: Examples where the exact match did not occur but F1 was good.

Prediction	Ground Truth
'before', 'attach', 'primary', 'storage' 'reset', 'buffer'	'before', 'detach', 'primary', 'storage' 'reset'

Table 5: Performance of using different contexts.

Model	Precision	Recall	F1	EM Acc
Token seq	70.25%	64.75%	67.39%	49.44%
Local cxt	69.60%	64.38%	66.89%	50.95%
+ In-file Project cxt	75.16%	71.83%	73.46%	59.51%
+ Documentation cxt	**77.01%**	**74.15%**	**75.60%**	**62.01%**

Although MNire and DeepName also consider the contexts beyond the method body, the contexts extracted by their approaches are different from ours. They only consider the contexts directly interacting with the target method, such as the sibling methods, callers methods, and callees methods. However, the methods which have no explicit interaction with the target methods can also provide essential information for understanding the functionality of the target method. For example, the methods appeared in the imported files, as shown in our previous motivation examples. Besides, MNire and DeepName use an RNN-based model to learn the relationship among the entities in the context. In our model, we extract contexts from a larger set of program entity candidates and employ a powerful backbone model to model the contexts, which is based on the self-attention mechanism. Besides, we also give the project-specific contexts more attention weights by applying invoked weight matrix. When generating the names of the target method, different decoder layers are utilized to focus on the contexts of different levels. Thus, our model can achieve better performance than baseline models.

Among these metrics, the exact match accuracy is much more strict than the other three metrics, which calculates the percentage of the predicted method names that are exactly the same as the ground truth. The other three metrics are based on the subtoken overlapping between the predicted names and the target names, where the order of the subtokens is ignored. The results show that our model obtains larger improvements on exact match accuracy and recall, which further demonstrates that the subtokens in the predicted names generated by our model can cover much more target subtokens than the other baselines. Therefore our model can fully and precisely describe the functionality of the method body.

Table 6: The results on the extracted documented methods.

Model	Precision	Recall	F1	EM Acc
GTNM	85.36%	82.54%	83.93%	70.60%
- doc	80.31%	76.65%	78.44%	64.14%

5.2 RQ2: The contributions of contexts in the same file

In the previous experiment, we consider contexts of the same file (i.e., local context, in-file project-specific context, and documentation context) for generating the method name. To answer this research question, we conduct experiments using different context combinations. As shown in Table 5, the first row shows the results of only taking the source code token sequence in the method body as input. The second row presents the result of using the local context (i.e., the entities' names of the method signature and variables) as input to suggest the method name. The third row shows the results of using both the in-file project-specific context and local context. The last row gives the results of using all three contexts: local, in-file project-specific, and documentation context.

As seen from Table 5, comparing the results of using local context (sequence length is 55) with the results of using source code token sequence (sequence length is 200), the performance is comparable, and using the local context can achieve higher exact match accuracy. However, the length of the local context is much shorter than the source code token sequence, which demonstrates that the local context extracted by our model contains enough information about the functionality of the method body, and the shorter context can improve the computational efficiency of the model. When we further incorporating the project-specific context information, the performance is improved by a large margin. Specifically, the F1 score and exact match accuracy significantly increase from 66.89% and 50.95% to 73.46% and 59.51%. The substantial improvement shows that the project-specific context, which can offer knowledge about the project information, is essential and efficient for improving the performance of method name recommendation.

When the documentation information is added, the performance is further improved. However, in our whole dataset, only about 20% of the methods have the document information. Thus, for most of the methods, the documentation context information is missing. To directly illustrate the contribution of the documentation context information, we extract those documented methods from the whole dataset and present the results on the extracted dataset. As shown in Table 6, the first row shows the results of our full model on the extracted dataset, and the second row shows the results of removing the documentation context from the input. When removing the documentation context, the performance is decreased by 5.1 in precision, 5.9 in recall, 5.5 in F1, and 6.5 in exact match accuracy, respectively. The results demonstrate that the documentation context can provide useful information for the method name suggestion.

5.3 RQ3: The contribution of cross-file context

When considering the cross-file project-specific context, we need to preserve the project structure of the programs in the dataset. Since more contextual information can be accessed, we assume that the model can be trained in a low-resource setting, that is, fewer programs are needed for training the model. Thus, we only use a subset of the whole training dataset in this experiment. Specifically, we sample 4000 projects from the big training set as a small training set and extract the cross-file project-specific context for the programs in the sampled projects. We compare with the results of our model setting without using project-specific context. To further demonstrate the effectiveness of the cross-file project-specific context, we conduct the experiment under the cross-project setting. That is, we split the corpus based on the projects instead of files or the methods. The cross-project setting is challenging and reflects better the real-world usage of the method name recommendation where the model is trained on the set of existing projects and used to check for a new project.

Table 7: Performance of using cross-file project-specific context under cross-project and low-resource setting.

Model	Precision	Recall	F1	EM Acc
w/o cross-file cxt	67.25%	64.66%	65.93%	49.71%
w/ cross-file cxt	73.52%	70.65%	72.06%	60.69%

The results are shown in Table 7. As seen from the results, with the help of cross-file project-specific context, our model can achieve comparable results with the results of the previous model setting, where the training set is bigger and in-project split, only using less than 50% of the whole training set and under the challenging cross-project experimental setting. When removing the cross-file project-specific context, the performance of the model drops a lot, which further demonstrates the importance of the cross-file project-specific context.

6 DISCUSSION

6.1 Qualitative Analysis

We perform qualitative analysis on the human-written method names and method names which are automatically generated by GTNM. In most cases, the names generated by GTNM are exactly the same as the human-written names. To figure out in what cases our model generates different names with human, we randomly sample 200 cases where the names generated by our model are different from the ground truth from the test to analyze the results.

Following McBurney and McMillan [31] and Hu et al. [20], we performed qualitative analysis to obtain opinions from participants on the quality of the generated-name, aiming at getting the feedback on our approach and directions for future-work. We invited 8 volunteers with 3-5 years of Java development experience to evaluate the generated names of the sampled 200 cases in the form of a questionnaire. Each participant is asked to answer several questions, including whether the human-written-names or generated-names are good, what are the differences between two names, etc. According to the questionnaire results, we summarize top-4 representative situations (The proportion of each situation is 19.4%/43.6%/6.6%/11.9%) as shown in Table 8.

Contain More Detailed Information. As shown in method 1, human just names the method as "add". What and when to add is not given. The human-written method name is very short and cannot reflect the detailed role of the target method. In cases like

Table 8: Examples of generated summaries given Java methods.

Examples	
Method 1	`/** * Adds a path (but not the leaf folder) if it does not already exist. */ protected void ____ (List<String> path, int depth) { int parentSize = path.size() - 1; String name = path.get(depth); Folder child = getChild(name); if (child == null) { child = new Folder(name); ... }`
Human-written GTNM	"add" "add", "path", "if", "not", "exists"
Method 2	`/** * Append the longs in the array to the selection, each separated by a comma */ private void ____ ( long[] objects ) { for ( int i = 0; i < objects.length; i++ ) { selection.append( objects[i] ); if ( i != objects.length - 1 ) { selection.append( ',' ); } } }`
Human-written GTNM	"join", "in", "selection" "append", "selection"
Method 3	`/** * Calculates the DefinitionUseCoverage fitness for the given DUPair on the given ExecutionResult */ public double ____ () { if (isSpecialDefinition(goalDefinition)) return calculateUseFitnessForCompleteTrace(); double defFitness = calculateDefFitnessForCompleteTrace(); if (defFitness != 0) return 1 + defFitness; return calculateFitnessForObjects(); }`
Human-written GTNM	"calculate", "d", "u", "fitness" "calculate", "fitness", "for"
Method 4	`/** * Validate removal of invalid entries. */ public void ____ () { RightThreadedBinaryTree<Integer> bt = new RightThreadedBinaryTree<Integer>(); assertFalse (bt.remove(99)); bt = buildComplete(4); assertFalse (bt.remove(99)); assertFalse (bt.remove(-2)); }`
Human-written GTNM	"test", "invalid", "removals" "test", "remove", "invalid"

this, GTNM tends to generate a longer name that contains more information about the method's functionality. In this example, GTNM suggests a more detailed name "add path if not exists", which indicates that the object and the usage scenario of the target method. Our model can learn this detailed information from the documentation, parameters, and the method body. In the whole test set, 25% of the wrong cases belong to this situation.

Synonyms. As shown in method 2, the human-written name and the name generated by our model have the same meaning, and the verbs used in these two names are synonyms ("join in" and "append"). Since "join in" is not as often used as "append" in the method names, and the contexts (including the project-specific context, local context, and the documentation context) also do not offer the relevant information about it. Thus, GTNM cannot correctly suggest the subtokens "join in". However, the name generated by our model can also precisely describe the functionality of the target method, which is also semantic consistent and acceptable.

Acronym. In method 3, the human-written name contains an acronym for the specific entities, i.e., "du" for "definition use", which our model cannot correctly infer. Based on the given contexts,

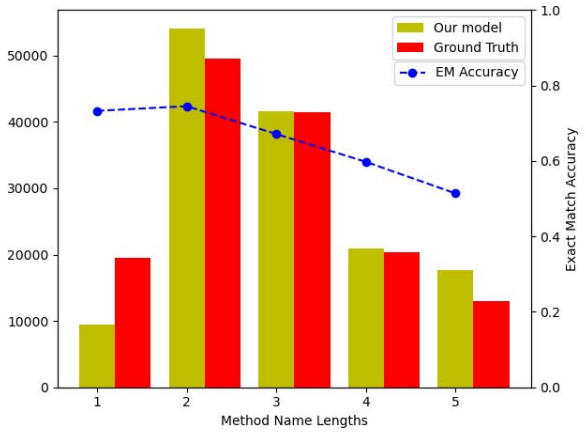

Figure 3: The method name length distribution and the exact match accuracy of different name lengths

GTNM suggests a name that has a similar style with the project-specific context, but fails to suggest the acronym for specific entity names.

Different Word Orders. As shown in method 4, the subtokens in the human-written name and GTNM suggested name are almost the same (except for "removals" and "remove"), but the subtoken orders are different. In this example, the different orders do not affect the semantic of the method name, and both of the two names express the same meaning. However, in other cases, the semantic of the names with different subtoken orders might be different. 0.7% of the wrong cases belong to this situation.

6.2 Length analysis

We further analyze the generated name length distribution and the performance of GTNM for different name lengths. As shown in Figure 3, the lengths of the method names (the number of subtokens in the method name) mainly range from 2 to 3. Our model generates fewer names of length 1, and generated more names with lengths 4 and 5. Among all the methods, only 13.78% of the names generated by our model are shorter than the ground truth. We apply the Wilcoxon Rank Sum Test (WRST) [44] to test whether the increase in the method name length is statistically significant, and all the p-values are less than 1e-5, which indicates a significant increase. We also use Cliff's Delta [29] to measure the effect size, and the values are non-negligible. Thus, our model tends to suggest more detailed names for the method. Besides, we also give the exact match accuracy of different lengths. As the length increase, the method naming task becomes harder. Even though our model can still achieve more than 50% accuracy for the names of length 5.

6.3 Explainability Analysis

Lack of explainability is an important concern in many complex AI/ML models in SE [35, 40]. It is crucial to ensure that the model is learned correctly and the logic behind the model is reasonable, which is also important for method name recommendation task. In this section, we analyze the explainability of GTNM. We employ model's confidence about its prediction to decide whether to accept the model's recommendation. Prediction Confidence Score (PCS) [47] which depicts the probability difference between the two classes with the highest probabilities is a measure for evaluating model's confidence. In our model, the Pearson Correlation Score between PCS and F1-score of the generated names is 0.612 and p-value <0.05, demonstrating that the correctness of the generated name is closely related to the model's confidence about its prediction. Thus, users can decide whether to accept the generated names depending on the case's error tolerance and the model's confidence.

6.4 Threats to Validity

Threats to external validity relate to the quality of the dataset we used and the generalizability of our results. We evaluate our approach on the Java dataset, which is a benchmark dataset for method name suggestion, and has been used in previous work [6, 7, 34]. All of the programs in the dataset are collected from top-ranked and popular GitHub repositories. Thus, most-of-the-names are expected consistent. However, there still exist a few cases that the name is inconsistent as shown in section 6.1. Besides, further studies are also needed to validate and generalize our findings to other programming languages. Furthermore, our case study is on a small scale. More user evaluation is needed to confirm and improve the usefulness of our model.

Threats to internal validity include the influence of the model architectural choices and the hyper-parameters used in our model. The hyper-parameters and architectural choices were obtained by a mix of small-range random grid search and manual selection. Thus, there is little threat to the hyper-parameter choosing, and there might be room for further improvement. However, current settings have achieved a considerable performance increase.

Threats to construct validity relate to the suitability of our evaluation measure. We adopted the measure used by the previous method name recommendation work [5–7, 34], which measured precision, recall, and F1 score over subtokens, and exact match accuracy. This is based on the idea that the quality of the generated method name is mostly dependant on the sub-words that were used to compose it.

7 RELATED WORK

7.1 Code Representation

Code representation is a hot research topic in both software engineering and machine learning fields. Different neural network-based approaches have been proposed for representing programs as vectors, which can be divided into the following categories: (1) source code token (subtoken) sequence - Using the source code token sequence as input. (2) AST node sequence - Using the flattened AST node sequence as input. (3) AST paths - Using a path through the AST as input. (4) Graph - Extending ASTs through adding edges to build the graph as input. (5) Program entities - Using tokens in program entities' names. These learned program vectors then can be used for various SE tasks, such as code summarization [19, 43], method name recommendation [7, 34], code clone detection [33, 46], code completion [21, 26, 27], etc. These different approaches model the program from different aspects, for example, ASTs can represent the structure and the syntax of the source code better, while the graphs focus more on the data flow and the semantic of the programs. For method name recommendation, existing research mainly focuses on modeling the method

body as token sequence [5, 34] or AST paths [6, 7], and then built an RNN-based encode-decoder framework to generate the subtokens of the method name.

7.2 Neural Machine Translation

Neural Machine Translation (NMT) [45] is an end-to-end learning approach for automated translation. In recent years work of NMT is largely based on encoder-decoder architecture [11], where the encoder maps an input sequence of words $x = (x_1, ..., x_n)$ to a sequence of continuous representations $z = (z_1, ..., z_n)$. Given z, the decoder then generates a sequence of output words $y = (y_1, ..., y_m)$ one token at a time, hence modeling the conditional probability: $p(y_1, ..., y_m|x_1, ..., x_n)$. The encoder-decoder architecture has been applied across many SE seq2seq tasks, including code summarization [5, 19], method name recommendation [7, 34], code generation [37, 43], program translation [14], etc. Different neural networks can be used in the encoder and decoder. Code2seq [6] employs a bi-directional LSTM to encode the AST paths then averages the representations of all the paths as the final representation of the program encoder, and employs another LSTM as the decoder to generate the output (method name or code summarization). Hu et al. [19] use RNN for both encoder and decoder for code comment generation task. Allamanis et al. [5] employ CNN to encode the code snippet and use GRU as decoder to generate the tokens of the method name. Fernandes et al. [15] employ GNN as the encoder and LSTM as the decoder for a range of summarization tasks. Ahmad et al. [3] use transformer network for both the encoder and decoder in code summarization task.

7.3 Method Name Recommendation

Recommending meaningful and consistent method names is important for ensuring readability and maintainability of programs. Many approaches have been introduced to suggest succinct names for methods [5, 7, 34], where different model architectures and method contexts are considered. In this section, we summarize related work on method name recommendation from the following two aspects.

7.3.1 Models. Suzuki et al. [38] proposed an N-gram based approach to evaluate the comprehensibility of method names and suggest comprehensible method names. Liu et al. [28] follow an information retrieval (IR) method with the motivation that two methods with similar bodies should have similar names. They use paragraph Vector and Convolutional Neural Networks to produce the vector representations of method names and bodies, respectively. They compared the similarity of the names retrieved from the method body vector space and the method name vector space to identify the inconsistent method names. For the inconsistent names, they use the names of methods whose bodies are similar to the body of the input method to suggest the new method name. However, methods with the same bodies can still have different names since they are in different projects and are under different contexts. Besides, the IR-based approach cannot generate a new name that it has not seen before. Another kind of researches based on NMT models, where encoder-decoder framework is used to encode the method bodies and generate the method names [5, 7, 34]. Allamanis et al. [5] built a convolutional attentional network to extract local features of the subtoken sequence from the method body, and then use these features to suggest names for methods. Alon et al. [7] design attention-based neural network to encode the AST paths into vectors, and based on the path representation to make predictions on the method's name. Zügner et al. [48] proposed Code Transformer, a Transformer-based language-agnostic code representation model. They combined distances computed on structure and context in the self-attention operation, which can learn jointly from the structure and context of programs relying on language-agnostic features. They applied their representations to the task of method name suggestion. Nguyen et al. [34] proposed an RNN-based seq2seq approach to recommend method names and to detect method name inconsistencies. They take the program entities in the method body and enclosing class name as the input. Li et al. [24] also developed an RNN-based seq2seq approach DeepName for method name consistency checking and suggestion, which extended the contexts by considering the internal context, the caller and callee contexts, sibling context, and enclosing context.

7.3.2 Method Contexts. Different method contexts are taken into account for method name recommendation. Most of the research only focused on exploiting the features from the method body, where the token sequences or ASTs of the method body are taken as the inputs. Allamanis et al. [5] considered the token sequence from the method body and built a convolutional attentional network to extract the features from the context. Alon et al. [7], Alon et al. [6], Zügner et al. [48], and Peng et al. [36] considered the AST paths extracted from the method body as the context, and made predictions on the method's name based on the path representation. In addition to the data from the method body, many research began to include the information from a wide range of contexts. Nguyen et al. [34] took the program entities in the method body and enclosing class name as the input. Wang et al. [42] also considered other methods in the project that have call relations with the target method. Li et al. [24] further extended the contexts by considering the internal context, the caller and callee contexts, sibling context, and enclosing context. Inspired by these approaches, we further considered the nested scopes of the project and the documentation of the method by extracting the project-specific and documentation context, which can help for suggesting accurate method names.

8 CONCLUSION

In this paper, we propose GTNM, a global method name suggestion approach, which considers contexts of different levels, including local context, project-specific context, and the documentation of the target method. We employ a transformer-based seq2seq framework to generate the method names, which uses the attention mechanism to allow the model attending to different level contexts when generating the names. The experimental results on Java methods show that our model has a substantial improvement over baseline models.

ACKNOWLEDGMENTS

This research is supported by the National Key R&D Program of China under Grant No. 2020AAA0109400, and the National Natural Science Foundation of China under Grant Nos. 62072007, 62192733.

REFERENCES

[1] Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus. 2011. 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, 125–134.

[2] Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus. 2011. The Effect of Lexicon Bad Smells on Concept Location in Source Code. In *11th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM 2011, Williamsburg, VA, USA, September 25-26, 2011*. IEEE Computer Society, 125–134. https://doi.org/10.1109/SCAM.2011.18

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 4998–5007. https://doi.org/10.18653/v1/2020.acl-main.449

[4] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 38–49. https://doi.org/10.1145/2786805.2786849

[5] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings, Vol. 48)*, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.). JMLR.org, 2091–2100. http://proceedings.mlr.press/v48/allamanis16.html

[6] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. 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. https://openreview.net/forum?id=H1gKYo09tX

[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[8] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Trans. Software Eng.* 45, 12 (2019), 1170–1188. https://doi.org/10.1109/TSE.2018.2827384

[9] Venera Arnaoudova, Laleh Mousavi Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2014. REPENT: Analyzing the Nature of Identifier Renamings. *IEEE Trans. Software Eng.* 40, 5 (2014), 502–532. https://doi.org/10.1109/TSE.2014.2312942

[10] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2016. Linguistic antipatterns: what they are and how developers perceive them. *Empir. Softw. Eng.* 21, 1 (2016), 104–158. https://doi.org/10.1007/s10664-014-9350-8

[11] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.0473

[12] Kent Beck. 2007. *Implementation patterns*. Pearson Education.

[13] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2009. Relating Identifier Naming Flaws and Code Quality: An Empirical Study. In *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*, Andy Zaidman, Giuliano Antoniol, and Stéphane Ducasse (Eds.). IEEE Computer Society, 31–35. https://doi.org/10.1109/WCRE.2009.50

[14] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree Neural Networks for Program Translation. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 2552–2562. https://proceedings.neurips.cc/paper/2018/hash/d759175de8ea5b1d9a2660e45554894f-Abstract.html

[15] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=H1ersoRqtm

[16] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[17] Johannes C. Hofmeister, Janet Siegmund, and Daniel V. Holt. 2017. Shorter identifier names take longer to comprehend. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, Martin Pinzger, Gabriele Bavota, and Andrian Marcus (Eds.). IEEE Computer Society, 217–227. https://doi.org/10.1109/SANER.2017.7884623

[18] Einar W. Høst and Bjarte M. Østvold. 2009. Debugging Method Names. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5653)*, Sophia Drossopoulou (Ed.). Springer, 294–317. https://doi.org/10.1007/978-3-642-03013-0_14

[19] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, Foutse Khomh, Chanchal K. Roy, and Janet Siegmund (Eds.). ACM, 200–210. https://doi.org/10.1145/3196321.3196334

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

[21] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code != big vocabulary: open-vocabulary models for source code. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1073–1085. https://doi.org/10.1145/3377811.3380342

[22] Dawn J. Lawrie, Christopher Morrell, Henry Feild, and David W. Binkley. 2006. What's in a Name? A Study of Identifiers. In *14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*. IEEE Computer Society, 3–12. https://doi.org/10.1109/ICPC.2006.51

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

[24] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. A Context-based Automated Approach for Method Name Consistency Checking and Suggestion. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 574–586.

[25] Ben Liblit, Andrew Begel, and Eve Sweetser. 2006. Cognitive Perspectives on the Role of Naming in Computer Programs. In *Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2006, Brighton, UK, September 7-8, 2006*. Psychology of Programming Interest Group, 11. http://ppig.org/library/paper/cognitive-perspectives-role-naming-computer-programs

[26] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. 2020. A Self-Attentional Neural Architecture for Code Completion with Multi-Task Learning. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 37–47. https://doi.org/10.1145/3387904.3389261

[27] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task Learning based Pre-trained Language Model for Code Completion. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 473–485. https://doi.org/10.1145/3324884.3416591

[28] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Tae-young Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1–12. https://doi.org/10.1109/ICSE.2019.00019

[29] Guillermo Macbeth, Eugenia Razumiejczyk, and Rubén Daniel Ledesma. 2011. Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica* 10, 2 (2011), 545–555.

[30] Robert C. Martin. 2009. *Clean Code - a Handbook of Agile Software Craftsmanship*. Prentice Hall. http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0132350882,00.html

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

[32] Steve McConnell. 2004. *Code complete - a practical handbook of software construction, 2nd Edition*. Microsoft Press. https://www.worldcat.org/oclc/249645389

[33] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. 2019. CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API Documentation. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 1026–1037. https://doi.org/10.1109/ASE.2019.00099

[34] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting natural method names to check name consistencies. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1372–1384. https://doi.org/10.1145/3377811.3380926

[35] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. 1–18.

[36] Han Peng, Ge Li, Wenhan Wang, Yunfei Zhao, and Zhi Jin. 2021. Integrating Tree Path in Transformer for Code Representation. *Advances in Neural Information Processing Systems* 34 (2021).

[37] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A Grammar-Based Structural CNN Decoder for Code Generation. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 7055–7062.

https://doi.org/10.1609/aaai.v33i01.33017055

[38] Takayuki Suzuki, Kazunori Sakamoto, Fuyuki Ishikawa, and Shinichi Honiden. 2014. An approach for evaluating and suggesting method names using n-gram models. In *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, Chanchal K. Roy, Andrew Begel, and Leon Moonen (Eds.). ACM, 271–274. https://doi.org/10.1145/2597008.2597797

[39] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. 1996. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Program. Lang.* 4, 3 (1996), 143–167. http://compscinet.dcs.kcl.ac.uk/JP/jp040302.abs.html

[40] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[42] Shangwen Wang, Ming Wen, Bo Lin, and Xiaoguang Mao. 2021. 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*. 741–753.

[43] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code Generation as a Dual Task of Code Summarization. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 6559–6569. https://proceedings.neurips.cc/paper/2019/hash/e52ad5c9f751f599492b4f087ed7ecfc-Abstract.html

[44] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.

[45] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR* abs/1609.08144 (2016). arXiv:1609.08144 http://arxiv.org/abs/1609.08144

[46] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 783–794. https://doi.org/10.1109/ICSE.2019.00086

[47] Xiyue Zhang, Xiaofei Xie, Lei Ma, Xiaoning Du, Qiang Hu, Yang Liu, Jianjun Zhao, and Meng Sun. 2020. Towards characterizing adversarial defects of deep learning software from the lens of uncertainty. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 739–751.

[48] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-Agnostic Representation Learning of Source Code from Structure and Context. In *ICLR (Poster)*. OpenReview.net.