2023 International Conference on Frontiers of Robotics and Software Engineering (FRSE) | 979-8-3503-0111-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/FRSE58934.2023.00016

A Method for Automatic Code Comment Generation Based on Different Keyword Sequences

1st Chengcheng Wang*
Beijing University of Technology
Faculty of Information Technology
Beijing
wangchengcheng@emails.bjut.edu.cn

2nd Hang Su
Beijing University of Technology
Faculty of Information Technology
Beijing
suhang@bjut.edu.cn

3rd Hongyu Gao
Beijing University of Technology
Faculty of Information Technology
Beijing
hygao@bjut.edu.cn

4th Libao Zhang
Beijing University of Technology
Faculty of Information Technology
Beijing
zhanglibao@emails.bjut.edu.cn

***Abstract*—Automatic generation of code comments can help alleviate software project development and maintenance difficulties caused by insufficient, missing, or mismatched code comments. Previous works have demonstrated that identifying the intention of code and the corresponding comment categories can enhance the accuracy of automatic code comment generation. However, this method's effectiveness relies on the expertise of programmers and requires a significant amount of human effort. We propose an automatic code comment generation method based on different keyword sequences to address this issue. We first investigate the verbs in code comments of mainstream public datasets and use automated methods to classify the comments into three categories. Then we extract and encode the keyword sequences of the code according to the comment category using different ways. The encoded sequences are combined with three classical comment generation models to generate comments automatically. Additionally, an algorithm for selecting API comments is proposed to address the issue of API comment quantity causing a decrease in the quality of generated comments during the keyword selection process. Experimental results indicate that the performance of the baseline model can be improved in terms of BLEU, ROUGE-L, and BLEU-DC metrics. In addition, compared with two models that combine other information, there is a certain degree of improvement in terms of three metrics, indicating that the proposed method can generate code comments more accurately.**

Automatic comment generation; software maintenance; automatic comment classification; verb; different keyword sequence; select API comment; different models

I. INTRODUCTION

The automatic code comment generation task involves generating a brief natural language description for a given piece of code [1]. Code comments can assist developers become familiar with a code base's core components, understand the code's design principles and behavior, and facilitate software development and maintenance.

In practice, writing code comments often requires additional effort from developers. The lack of professional knowledge or awareness of the importance of comments can lead to missing, insufficient, or mismatched comments, which severely impact software understanding, reuse, and maintenance. High-quality code comments can enhance the efficiency of software development and maintenance, while reducing their costs. Consequently, it is essential to have good automatic comment generation tools, which can improve the overall quality of software products.

Chen et al. [2] investigated the relationship between code blocks and the category of their corresponding comments, and found that this relationship can improve the performance of code comment generation. They classified comments into six categories according to code's intention, and manually labeled 20000 code-comment pairs. However, this classification approach is time-consuming and relies on the expertise of programmers. We propose an automatic classification method to address the issue of manual classification. High quality comments usually start with a verb, and the prediction of the first word will influence the subsequent predictions of the model [3]. We automatically classify comments into three categories based on the verb in the comments: (1) comments containing program semantic verb; which refer to verb with specified semantics in programming languages, such as "return"; (2)comments containing natural language verb; (3) comments that do not contain verbs. Different ways were used to extract corresponding keyword sequences from the code for different categories of comments. Shahbazi et al. [4] extracted Application Programming Interface(API) comments as keyword sequences to improve the results. The study found that when the number of API comments increases, the quality of generated comments has decreases. We propose an algorithm for selecting API comments to address this issue,.

The main contributions in our work are summarized as follows:

- We propose a method for automatically classifying comments based on the verb in the comment to address the issue of manual classification based on the code's intention, which relies on the programmer's expertise.
- We propose an algorithm for selecting API comments to solve the problem of poor comment generation due to the increase of API comments.
- Combine keyword sequences with classical comment generation models. The results show that our method augments existing models and outperforms two models that combine other

information. Additionally, ablation experiments have demonstrated that keyword information can enhance the generation of code comments.

II. Related work

The automatic generation of code comments has been a significant task in software engineering. Since the concept was proposed, many approaches have been developed. First, template-based methods extracted keywords from the code and integrated them into natural language descriptions based on predefined templates, but these methods relied on well-defined identifier naming. Second, retrieval-based methods did not require well-named identifiers or related templates. Instead, they retrieved relevant code segments and used them to generate comments. However, measuring the semantic similarity between code segments became a challenge. Finally, learning-based methods used models to learn syntax and semantic information in code to generate comments.

The template-based method is one of the most significant methods for generating code comments. Hill et al. [5] used Software Word Usage Model(SWUM) to identify the action, theme and secondary arguments from Java method signature and use them to generate intelligible text for the comment. Sridhara et al. [6] first constructed SWUM by combining traditional program analysis and natural language analysis to link language information with structural information. Then they used language information based on themes, second arguments and selected important content based on the structure of the code to generate natural language comments. Sridhara et al. [7] further generated comments for parameters using heuristic rules and integrate them into code comments based on templates.

The retrieval-based methods searched for similar codes from software repository to generate comments. Haiduc et al. [8] used the Vector Space Model(VSM) and Latent Dirichlet Allocation(LSI) models to analyze source code, represent the identifiers of the code as a matrix, extract the most relevant words by setting matrix weights, and consider them as comments. Panichella et al. [9] used heuristic rules and the VSM model to process and analyze developers' communication on method descriptions. They extracted paragraph texts that can be traced back to the source code, calculated their cosine similarity with the target code to select relevant paragraphs, and used them as the method's descriptive information. Wong et al. [10] discovered the most relevant code snippets in a corpus by using code clone detection techniques and use the corresponding descriptions as the comment of the target code.

Recently, some research has attempted to use deep learning methods to generate code comments. Iyer et al. [11] proposed the CODE-NN method, which uses Long Short-Term Memory (LSTM) networks and attention mechanisms to generate natural language comments for C# code snippets and SQL queries. Zheng et al. [12] proposed a Code Attention mechanism. They used gated recurrent unit(GRU) networks to encode code containing important semantics such as loops, and used proposed global attention mechanism to generate code comments more accurately. Hu et al. [13] proposed a traversal method for Abstract Syntax Tree (AST) called SBT (Structure-Based Traversal) to take advantage of the structural information. This method encloses the subtrees contained in a node within a pair of parentheses, allowing the flattened AST to be input into the model. LeClair et al. [13] proposed the ast-attendgru method, which models tokens and AST separately and uses attention mechanisms for each to generate code comments. Alon et al. [15] pointed out that AST is a useful complement to models, but using flattened AST as model inputs is not optimal. Therefore, they proposed the code2seq method, which randomly extracts paths from the AST to represent the structural features of the code. Xu et al. [16] proposed using graph neural networks to model AST to better capture the structural information.

Some studies attempt to use additional information to assist in generating code comments and utilizing information within the code. Hu et al. [17] utilize learned API knowledge to generate code summaries more accurately. Zhang et al. [18] considered method signatures and API calls as keywords. They applied max pooling on all hidden states to obtain the semantic vector, and finally generated code comments based on this vector. Haque et al. [19] attempted to consider other functions within the same file as contextual information for a function to improve the accuracy of generated comments. Bansal et al. [20] modeled other Java files in the same project to generate comments more accurately.

III. Our Approach

We first use the code and the corresponding labeled category to train a classifier. Then, we use the classifier to classify different codes and extract corresponding keyword sequences. Finally, after encoding the keyword sequences based on attention mechanism, we combine them with classical code comment generation model to generate comments more accurately.

A. *Comment Classification*

1) *Categories of Code Comment*

Chen et al. [2] classified comments into six categories according to the intention of the code based on Zhai's[21] work, but this classification is mainly applied to the task of comment propagation. In contrast, the classification methods proposed by Pascarella et al [22]. and Steidl et al [23]. emphasize comments that describe code functionality, which are particularly relevant for code summarization tasks. Therefore, we propose a new classification method that uses verbs to describe the functionality of methods from the perspective of their functional descriptions. For example, "sort the list of connected layer" indicates that the method is used for sorting. In the LeClair dataset[24], 80.60% of comments start with a verb [3]. Moreover, existing deep learning methods are mainly based on encoder-decoder architectures (such as seq2seq, graph2seq). Encoder-decoder models predict one word at a time, they first predict a word and use the predicted first word to predict the second word. The model will require a long recovery time if the first word is incorrect. Thus, the verb in comments is crucial for the model's predictions. We extract the verb in the comments and classify them into three categories based on the verb. According to the term-frequency of verbs in the LeClair dataset [24] and the Hu [17] dataset, we selected 'return' ,'set', 'get', 'test', 'write', 'read', 'if' and 'print' as the program semantic verb set. The comments containing these verbs were classified as the first category. The comments containing other verbs such as 'add' and 'create' were classified as the second category. The comments without any verbs were classified as the third category.

2) Train A Classifier

We process the codes and comments separately based on the code-comment pairs provided by LeClair [24].

For the comments, we first use the WordNet Lemmatizer to perform lemmatization on the words in the comments, such as lemmatizing 'returns' to 'return.' Then use the Stanford NLP Parser [25] to perform part-of-speech tagging on the words in the comments and extract the verb based on the tagging results. Finally, label comments with the corresponding category based on the verb tags.

We extract text features and syntactic features from the code. For text features, we extract term-frequency of the tokens, which is a widely used method in software engineering [26][27]. We extract method names, token numbers, and variable numbers for syntactic features, which are also widely used in software engineering [28].

After labeling the comments and extracting code features, we employ four classification techniques: Random Forest [29], LightGBM [30], Decision Tree [31], and Multinomial Naive Bayes [32], to train a classifier. These techniques have been proven effective in comment classification tasks [33][34].

To access the effectiveness of code comment classification, we employ information retrieval evaluation metrics: Precision, Recall, and F1scores.

TABLE I shows the results of different classification techniques, with the best being the Random Forest classification technique, which achieved an F1 score of 74.73 in ten-fold cross-validation. Therefore, we use the Random Forest classification technique to predict the comment categories on the test set.

TABLE I. CLASSIFICATION RESULTS

Classifier	Precision	Recall	F1
Random Forest	71.95	73.47	74.73
LightGBM	70.56	72.36	71.23
Decision Tree	67.23	67.67	66.85
Naïve Bayes	65.58	66.48	65.30

B. Keyword Extraction

Extracting keyword sequences mainly involves two steps: comment classification and keyword extraction. The method architecture is shown in Figure 1.

1) Comments Containing Programmatic Semantic Verb

Rodeghero et al. [7] tracked the eye movement of several programmers while writing comments and found that programmers focus more on the function signature than on the function call and focus more on the control flow than on the function call. Therefore, we consider the method name, parameters, and return value as keyword sequences for this category of comment.

2) comments containing natural language verb

We consider API comments as the keyword for this type of comment category. Shahbazi et al. [4] extracted API comments as keyword sequences to improve the results. However, the results show that when the number of API comments increases, the quality of generated comments has decreases. We propose an algorithm for selecting API comments for the keyword extraction of this category to address this issue.

We first use srcML [36] to extract API method names from per function. Then we crawl the comment and name of the API method from official documentation(such as Java 8). We match the extracted name and the crawled name to achieve API comments for each function.

ROUGE-L [37] is a recall-based metric that primarily considers the truthfulness of the predicted results. Let $S=\{s_1,s_2,\ldots,s_n\}$ denote a set of API comment statements in a function, where s_i is the *i*th statement in extracted API comments. We first calculate the ROUGE-L score for each statement in S with reference comment. Then we sort the statements based on their score, and in order to select the most relevant statements, we iteratively add one statement with the highest score to the keyword set. We calculate the ROUGE-L score for the keyword set with reference comment, and if it is higher than the previous score, we consider it as an informative statement. The algorithm is described in TABLE II.

TABLE II. Extract informative statements algorithm

Algorithm 1：Extract informative statements algorithm

```
  Input:   S, set of API comments statements,
           c, reference comment
  Output:  L, set of select keyword statements
  1   L= Ø, M= Ø
  2   for s in S
  3       score=RLSummary(s,c)
  4       M++{<s,score>}
  5   end for
  6   sortedByScore(M)
  7   maxScore=0
  8   for <s,score> in M
  9       score=RLSummary(toString(L∪{s}),c)
  10      if score>maxScore then
  11          L++{s}
  12          maxScore=score
  13      end if
  14  end for
  15  return L
```

where function RLSummary() (see line 3 in Algorithm 1) is calculate the ROUGE-L score between two statements.

Let $l=[l_1,l_2,\ldots,l_n]$ denote the labels of the API comment statements, where $l_i \in \{0,1\}$, $l_i=1$ means the

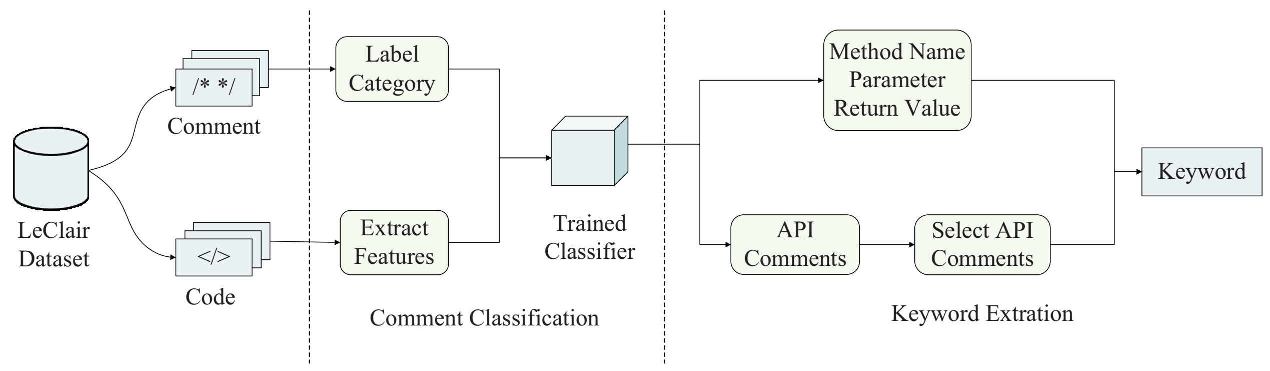

Figure 1. Overall architecture of keyword information extraction

*i*th statement is an informative statement. Therefore, extracting informative statements can be seen as a classification task: given the set of API comment statements S, output a predicted label value l_i for each statement s_i in S. The classification method steps are as follows:

a) Produce Ground-truth Labels: Select informative statements according to algorithm 1, label the selected comment statement as 1, and label the unselected comment statement as 0. Finally, we get the label $\hat{l_i}$ for each statement.

b) Produce Predicted Labels: The prediction model consists of an encoding layer and a classification layer. The encoding layer uses LSTM to encode the statements into vectors, and the classification layer uses softmax to predict the label l_i of the statements.

c) Model Training: During training, we update the parameter θ in the cross-entropy loss function. The equation for the cross-entropy loss function is as follows:

$$l(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\hat{l_i}\log l_i + (1-\hat{l_i})\log(1-l_i) \qquad (1)$$

Where $\hat{l_i}$ is the ground-truth label of the *i*th statement. l_i is the predicted label of the *i*th statement. $l_i \in \{0,1\}$, when l_i=1, the *i*th statement should be included in the keyword sequence.

3) comments that do not contain verbs

The API comments are regarded as the keyword sequences for comments that do not contain verbs, and the same extraction method for comments containing natural language verbs is applied.

C. *Overview*

The method for generating code comments based on different keyword sequences includes three phases: data preprocessing, encoding, and decoding. For each function, we have token input and AST input to the encoder, as well as comment to the decoder. Additionally, we introduce a keyword input extracted from each function and elaborate on how to extract keywords in the previous section. The method architecture is shown in Figure 3.

1) Data Preprocessing

a) Token Sequence: We split the code into tokens using camel case conventions, remove non-alpha characters, and set all tokens to lowercase to achieve the token sequences.

b) AST Sequence: We use two methods to convert an AST into a sequence. One is the method proposed by LeClair [13], which flattens the AST by enclosing the subtrees contained in each node in a pair of parentheses. The other is the method proposed by Alon [15], which represents the structural information of the code by randomly selecting a set of paths from the AST.

c) Keyword Sequence: According to the keyword extraction method proposed in the previous section, we obtain the keyword of the code, which can be split into tokens to achieve the keyword sequences.

2) Encoding

a) Token Encoding: After embedding each token into a 100-dimensional vector, we use a GRU to encode the token sequence and take the last hidden state of the neural network as the context vector h_{token}.

b) AST Encoding: we use two methods to utilize the structural information: one is to use a GRU to encode the AST sequence, and the other is to use a GNN to encode the nodes and edges on the AST. We take the last hidden state of the neural network as the context vector h_{ast}.

c) Keyword Encoding: The extracted keyword sequence includes the static information of the code and API comments. We share the embedding space with token input as it uses the same vocabulary as function comments and can save memory space and computation time. A 100-length embedding vector represents each word. We use GRU to encode the keyword sequences and take the last hidden state of the neural network as the context vector h_{key}.

3) Decoding

a) Vector concatenation: We use attention mechanisms for three types of encoders. During training,

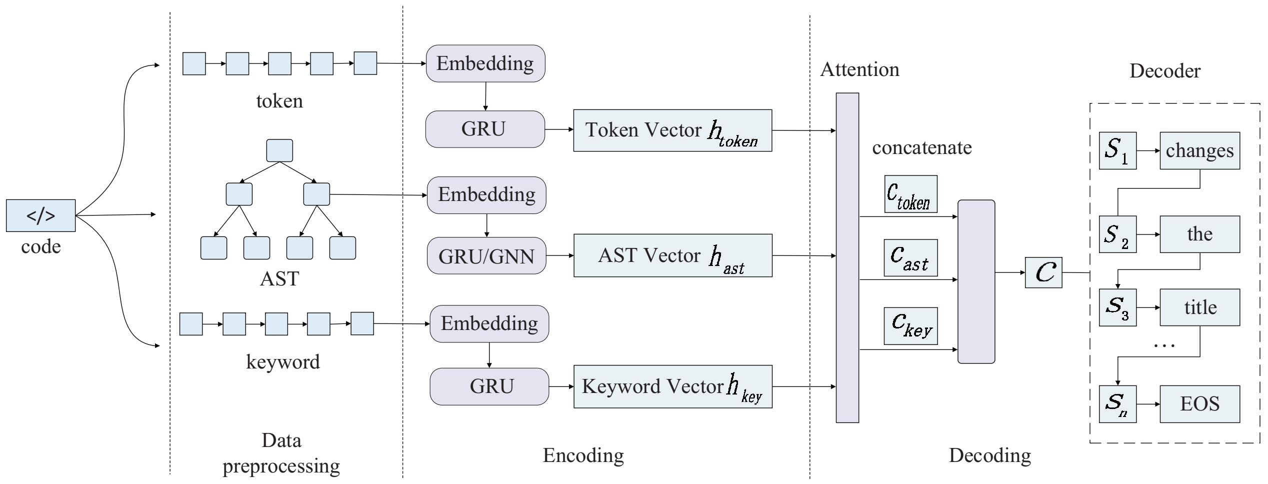

Figure 2. Overall architecture of code comment generation method

the attention vector is updated by computing the hidden states of the encoder at each time step and the hidden state of the reference comment. The equation for the attention vector at time t is as follows:

$$c_t = \sum_{i=1}^{T} \alpha_{ti} h_i \tag{2}$$

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{k=1}^{T} e_{tk}} \tag{3}$$

$$e_{ti} = v_\alpha^{\mathrm{T}} \tanh(W_a[s_{i-1}, h_i]) \tag{4}$$

Where h_i is the hidden state of the encoder at i time, e_{ti} represents the degree of influence that h_i has on the hidden state s_t of the decoder at time step t.

After concatenating the three vectors, we obtain the final attention-based context vector c. The equation for the context vector is as follows:

$$c = [c_{token}, c_{ast}, c_{key}] \tag{5}$$

b) Decoder: The attention-based decoder has three inputs: the hidden state of the previous time step s_{t-1}, the predicted output of the previous time step y_{t-1}, and the context vector of the current time step c_t.

The equation for the probability of the output at time step t is as follows:

$$p(y_t \mid y_1, y_2, \ldots, y_{t-1}) = soft\max(\tanh(W[s_{t-1}, y_{t-1}, c])) \tag{6}$$

We finally find the word corresponding to output y_t at t time by looking up the table.

IV. Experiment and Analysis

A. Dataset

We conduct our study on LeClair's recommended code-comment dataset[24]. This dataset contains 2.1M pairs of Java methods and comments, which were divided into 80% for the training set, 10% for the validation set, and 10% for the test set.

B. Evaluation Metrics

We follow the steps of neural code comment generation methods and compare the effectiveness of comment generation based on standard metrics on a large dataset. We use BLEU [38], ROUGE-L [37], and BELU-DC [39] metrics to evaluate the performance, where metric BLEU is based on precision, metric ROUGE-L is based on recall, and metric BLEU-DC is most correlated to human perception.

C. Baselines

We combine keyword sequences with three classical code comment generation models to validate that they can augment the performance of the baseline models. Moreover, we compare the experimental results with two state-of-the-art methods that combine other information.

ast-attendgru[13]: it combines words from code with flattened AST to input into the model and generate comments.

code2seq[15]: it randomly selects a set of paths from the AST to represent the structure information of the code.

graph2seq[16]: it models the AST nodes and edges through graph neural networks, capturing the structural information of AST more effectively.

ast-attendgru-fc,code2seq-fc,graph2seq-fc[19]: it uses other functions within the same file as contextual information for a function to improve the accuracy of generated comments.

ast-attendgru-pc,code2seq-pc,graph2seq-pc[20]: it combines information from other files in the same software project to enhance the accuracy of generated comments.

We use baseline+keyword to denote combined different models. For example, we use ast-attendgru+keyword to denote we combine keywords with flattened AST to generate comments.

D. Results Comparison

The experimental results show that by combining the keyword sequence with different code comment generation models, the baseline models can generate comments more accurately. The results are shown in TABLE III. The results show that the improvements in BLEU scores are between 1.9% and 9%, ROUGE-L scores between 2.7% and 5.3%, and BLEU-DC scores between 0.9% and 17.8%.

TABLE III. Experimental results combining different models

model	BLEU	ROUGE-L	BLEU-DC
ast-attendgru	19.44	49.91	13.09
ast-attendgru+keyword	**19.80**	**51.27**	**13.22**
code2seq	17.84	48.40	12.35
code2seq+keyword	18.06	50.85	13.25
graph2seq	16.44	46.82	10.68
graph2seq+keyword	17.92	49.33	12.58

We conduct a comparative experiment of the comment generation models that use the same dataset but combine contextual information from the same file and that combine file information from the same software project. The results are shown in TABLE IV. Compared with two models combining other information, the proposed method aggregates BLEU scores increase from 3% to 8.1% and ROUGE-L scores from 1.7% to 8%, and BLEU-DC scores from 3.8% to 19.2%.

TABLE IV. Experimental results combining keyword information and other information

model	BLEU	ROUGE-L	BLEU-DC
ast-attendgru-fc	18.84	48.92	12.73
ast-attendgru-pc	17.91	49.78	11.48
ast-attendgru+keyword	**19.80**	**51.27**	**13.22**
code2seq-fc	16.86	49.67	11.83
code2seq-pc	16.70	50.01	11.11
code2seq+keyword	18.06	50.85	13.25
graph2seq-fc	17.04	47.23	11.76
graph2seq-pc	17.37	49.74	11.47
graph2seq+keyword	17.92	49.33	12.58

E. *Ablation experiment*

To demonstrate the importance of keyword information in the proposed method, this study conducted ablation experiments on the LeClair [24]dataset, using BLEU as the evaluation metric. We conducted two experiments: (1) While keeping other conditions unchanged, we observed the experimental results by using only two types of code information each time. (2) We replaced the GRU model used in this study with Transformer and LSTM models.

TABLE V Results of ablation experiment

Experimental condition	BLEU
No token	16.53
No AST	19.19
No keyword	19.24
LSTM	18.99
Transformer	17.47
Our model	19.80

Experimental results are shown in TABLE V. This study conducted two ablation experiments. Firstly, the impact of keyword information on model performance was investigated. Secondly, the influence of using different models on the results was studied. In the first experiment, while keeping other conditions unchanged, one of the three types of code information was eliminated each time, and the experimental results were observed. The results showed that after eliminating keyword information, the BLEU score was 19.24, which decreased by 2.8%. This indicates that keyword information is significant for the model. The results after eliminating AST information were similar to those after eliminating keyword information, suggesting that code structure information and keyword information have the same important role in the model. After eliminating token information from the code, the result showed a BLEU score of 16.53, with a difference of 16.8% compared to before the ablation. This indicates that token information in the code is crucial for the model. In another ablation experiment, when the newer Transformer model was used to replace the GRU model, it was found that the model's performance dropped by 11.8 percentage points. This suggests that although the Transformer performs well in machine translation tasks, it does not perform well in code summarization tasks, possibly due to its tendency to overfit when dealing with short sequence data, leading to poorer performance on the test set. Additionally, compared to the Transformer model, the GRU model has fewer training parameters and faster computation speed. The experimental results indicate that the GRU-based model outperforms the LSTM-based and Transformer-based models.

F. *Effects of keyword sequence*

We explore the effects of keyword sequence in two ways: First, we provide an overview of the results. Second, we provide concrete cases to illustrate how the model works.

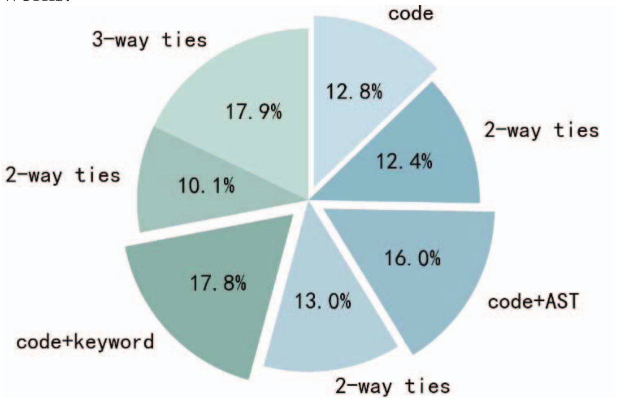

Figure 3. The percentage of methods where the predictions had the highest BLEU1 score for three models

First, consider Figure 3. This chart displays the percentage of functions for which the predictions of three different models had the highest BLEU1 score: attendgru only relies on token information, ast-attendgru with access to structure information, and attendgru+keyword with access to keyword information. We observed that a subset of functions performs best for each model, and not all models perform equally on the same function. The attendGRU model performs the best for 12.8% of the functions, while the ast-attendGRU model performs the best for 16% of the functions. The models with access to keyword information had the highest percentage of the best-performing functions at 17.8%.

We use different methods to extract keyword information from code based on its category to generate comments more accurately. TABLE VI presents three categories of code and the extracted keyword sequences and compares the comments generated by our model and the benchmark model. Through case analysis, we demonstrate that our model can generate comments more accurately.

TABLE VI. Case study of automatically generated code comments

Case 1	
code	public String getPrimaryLsid(String lsid){ TermQuery tq = new TermQuery(new Term(lsid, lsid)); try{ org.apache.lucene.search.TopDocs results = idSearcher.search(tq,1); if(results.totalHits>0) return idSearcher.doc(results.scoreDocs[0].doc).get("reallsid"); }catch(IOException e){} return lsid; }
reference comment	returns the primary lsid for the supplied lsid
ast-attendgru	get the lsid of the given lsid
code2seq	returns the primary term for the given term
graph2seq	returns the vocabulary for the given lsid
category	comments containing programmatic semantic verb
keyword	get primary lsid string lsid return lsid
Our model	return the lsid of the given lsid
Case 2	
code	public void changeTitle(){ String name = getProject().getProjectName(); if(name==null&&saveFile!=null) name=saveFile.getAbsolutePath(); if(name==null) name=Project.UNNAMED_PROJECT; MainFrame.this.setTitle(TITLE_START_TXT+name); }
reference comment	changes the title based on cur project and save file
ast-attendgru	change the project name
code2seq	change the name of the project
graph2seq	change the name of the name
category	comments containing natural language verb
keyword	sets the title of the dialog
Our model	changes the title of the project
Case 3	
code	protected void groupChanged(String name){ //if group switched print the name if (groupSwitched) { System.out.println("Group switched to " + name); this.groupSwitched = false; } else { System.out.println("Group name updated to " + name); } }
reference comment	called when group is switched
ast-attendgru	called when group is changed
code2seq	is called when group is changed
graph2seq	is called when group is changed
category	comments that do not contain verbs
keyword	if group switched print the name
Our model	called when the group has switched

In case 2, the baseline models don't predict the word title. But, our model pays attention to the keyword title, which can make it easier for the model to predict the correct words.

V. CONCLUSION

We propose an automatic comment generation method based on different keyword sequences to help developers better understand code, addressing issues such as incomplete and low-quality comments. Experimental results show that our method can generate comments more accurately, significantly improving the evaluation metrics of the baseline models. Moreover, compared to models that incorporate other information, our model performs better.

REFERENCES

[1] McBurney P W, McMillan C. Automatic source code summarization of context for java methods[J]. IEEE Transactions on Software Engineering, 2015, 42(2): 103-119.

[2] Chen Q, Xia X, Hu H, et al. Why my code summarization model does not work: Code comment improvement with category prediction[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2021, 30(2): 1-29.

[3] Haque S, Bansal A, Wu L, et al. Action word prediction for neural source code summarization[C]//2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2021: 330-341.

[4] Shahbazi R, Sharma R, Fard F H. API2Com: On the Improvement of Automatically Generated Code Comments Using API Documentations[C]//2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC). IEEE, 2021: 411-421.

[5] Hill E, Pollock L, Vijay-Shanker K. Automatically capturing source code context of nl-queries for software maintenance and reuse[C]//2009 IEEE 31st International Conference on Software Engineering. IEEE, 2009: 232-242.

[6] Sridhara G, Hill E, Muppaneni D, et al. Towards automatically generating summary comments for java methods[C]//Proceedings of the IEEE/ACM international conference on Automated software engineering. 2010: 43-52.

[7] Sridhara G, Pollock L, Vijay-Shanker K. Automatically detecting and describing high level actions within methods[C]//Proceedings of the 33rd International Conference on Software Engineering. 2011: 101-110.

[8] Haiduc S, Aponte J, Moreno L, et al. On the use of automated text summarization techniques for summarizing source code[C]//2010 17th Working Conference on Reverse Engineering. IEEE, 2010: 35-44.

[9] Panichella S, Aponte J, Di Penta M, Marcus A, Canfora G. Mining source code descriptions from developer communications. In: Proceedings of the 2012 20th IEEE International Conference on Program Comprehension, 2012. 63-72.

[10] Wong E, Yang J, Tan L. Autocomment: Mining question and answer sites for automatic comment generation[C]//2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2013: 562-567.

[11] Iyer S, Konstas I, Cheung A, et al. Summarizing source code using a neural attention model[C]//Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2016: 2073-2083.

[12] Zheng W, Zhou H Y, Li M, et al. Code attention: Translating code to comments by exploiting domain features[J]. arXiv preprint arXiv:1709.07642, 2017.

[13] Hu X, Li G, Xia X, et al. Deep code comment generation[C]//Proceedings of the 26th conference on program comprehension. 2018: 200-210.

[14] LeClair A, Jiang S, McMillan C. A neural model for generating natural language summaries of program subroutines[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 795-806.

[15] Alon U, Brody S, Levy O, et al. code2seq: Generating sequences from structured representations of code[J]. arXiv preprint arXiv:1808.01400, 2018.

[16] Xu K, Wu L, Wang Z, et al. Graph2seq: Graph to sequence learning with attention-based neural networks[J]. arXiv preprint arXiv:1804.00823, 2018.

[17] HU X, LI G, XIA X, et al. Summarizing source code with transferred API knowledge.(2018)[C]//Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI 2018), Stockholm, Sweden, 2018 July 13. 19: 2269-2275.

[18] ZHANG S S， XIE R,YE W. Keyword based automatic code summarization[J]. Computer Research and Development, 2020.

[19] Haque S, LeClair A, Wu L, et al. Improved automatic summarization of subroutines via attention to file context[C]//Proceedings of the 17th International Conference on Mining Software Repositories. 2020: 300-310.

[20] Bansal A, Haque S, McMillan C. Project-level encoding for neural source code summarization of subroutines[C]//2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC). IEEE, 2021: 253-264.

[21] Zhai J, Xu X, Shi Y, et al. CPC: Automatically classifying and propagating natural language comments via program analysis[C]//Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 2020: 1359-1371.

[22] Pascarella L, Bruntink M, Bacchelli A. Classifying code comments in Java software systems[J]. Empirical Software Engineering, 2019, 24(3): 1499-1537.

[23] Steidl D, Hummel B, Juergens E. Quality analysis of source code comments[C]//2013 21st international conference on program comprehension (icpc). Ieee, 2013: 83-92.

[24] LeClair A, McMillan C. Recommendations for Datasets for Source Code Summarization[C]//Proceedings of NAACL-HLT. 2019: 3931-3937.

[25] Manning C D, Surdeanu M, Bauer J, et al. The Stanford CoreNLP natural language processing toolkit[C]//Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations. 2014: 55-60.

[26] Frantzeskou G, MacDonell S, Stamatatos E, et al. Examining the significance of high-level programming features in source code author classification[J]. Journal of Systems and Software, 2008, 81(3): 447-460.

[27] Van Dam J K, Zaytsev V. Software language identification with natural language classifiers[C]//2016 IEEE 23rd inter-national conference on software analysis, evolution, and reengineering (SANER). IEEE, 2016, 1: 624-628.

[28] Ugurel S, Krovetz R, Giles C L. What's the code? automatic classification of source code archives[C]//Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining. 2002: 632-638.

[29] Breiman L . Random forests, machine learning 45[J]. Journal of Clinical Microbiology, 2001, 2:199-228.

[30] GuolinKe Q M, Finley T, Wang T, et al. Lightgbm: A highly efficient gradient boosting decision tree[J]. Adv. Neural Inf. Process. Syst, 2017, 30: 52.

[31] Loh W Y. Classification and regression trees[J]. Wiley interdisciplinary reviews: data mining and knowledge discovery, 2011, 1(1): 14-23.

[32] Schütze H, Manning C D, Raghavan P. Introduction to information retrieval[M]. Cambridge: Cambridge University Press, 2008.

[33] Pascarella L, Bacchelli A. Classifying code comments in Java open-source software systems[C]//2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). IEEE, 2017: 227-237.

[34] Pascarella L, Bruntink M, Bacchelli A. Classifying code comments in Java software systems[J]. Empirical Software Engineering, 2019, 24(3): 1499-1537.

[35] Rodeghero P, McMillan C, McBurney P W, et al. Improving automated source code summarization via an eye-tracking study of programmers[C]//Proceedings of the 36th international conference on Software engineering. 2014: 390-401.

[36] Collard M L, Decker M J, Maletic J I. Lightweight transformation and fact extraction with the srcML toolkit[C]//2011 IEEE 11th international working conference on source code analysis and manipulation. IEEE, 2011: 173-184.

[37] Lin C Y . ROUGE: A Package for Automatic Evaluation of summaries[C]// In Proceedings of the Workshop on Text Summarization Branches Out (WAS 2004). 2004.

[38] Papineni K, Roukos S, Ward T, et al. Bleu: a method for automatic evaluation of machine translation[C] //Proceedings of the 40th annual meeting of the Association for Computational Linguistics. 2002: 311-318.

[39] Shi E, Wang Y, Du L, et al. On the evaluation of neural code summarization[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 1597-1608.