An Intra-Class Relation Guided Approach for Code Comment Generation

Zhenni Wang†, Xiaohan Yu†, Yansong Feng∗, Dongyan Zhao

Wangxuan Institute of Computer Technology, Peking University, China
The MOE Key Laboratory of Computational Linguistics, Peking University, China
{wangzhenni,yuxiaohan,fengyansong,zhaodongyan}@pku.edu.cn

Abstract

Code comments are critical for maintaining and comprehending software programs, but they are often missing, mismatched, or outdated in practice. Code comment generation task aims to automatically produce descriptive comments for code snippets. Recently, methods based on the neural encoder-decoder architecture have achieved impressive performance. These methods assume that all the information required to generate comments is encoded in the target function itself, yet in most realistic situations, it is hard to understand a function in isolation from the surrounding context. Furthermore, the global context may contain redundant information that should not be introduced. To address the above issues, we present a novel graph-based learning framework to capture various relations among functions in a class file. Our approach is based on a common real-world scenario in which only a few functions in the source file have human-written comments. Guided by intra-class function relations, our model incorporates contextual information extracted from both the source code and available comments to generate missing comments. We conduct experiments on a Java dataset collected from real-world projects. Experimental results show that the proposed method outperforms competitive baseline models on all automatic and human evaluation metrics.

1 Introduction

Code comment generation is the task of automatically producing natural language descriptions for given code snippets. Appropriate and sufficient comments are essential for software maintenance and understanding (Xia et al., 2018). They allow developers to grasp the purpose of source code quickly and accurately. However, in real-life software projects, comments are often missing, incomplete or outdated (Briand, 2003). Existing comments will also need to be adjusted as the associated programs are updated, which could cause large time and labor costs. Hence, there is a significant need for automatic generation technologies that can effectively produce high-quality comments.

Recent works in code comment generation take the neural encoder-decoder architecture as their cornerstone (Hu et al., 2018a; Alon et al., 2019; LeClair et al., 2020; Zhang et al., 2020; Wei et al., 2020). However, these works only utilize the information provided by the target function itself. In object-oriented programming, classes are the building blocks that express algorithmic intentions and they encapsulate the interaction between functions. Therefore, the class-level contextual information should not be ignored when we attempt to generate code comments. There are some existing studies that attempt to fill this gap. Haque et al. (2020) encode all functions in a source file using GRU (Cho et al., 2014) and apply an attention mechanism to learn associations between the encoding results to words in the generated comment. Yu et al. (2020) construct a class graph that connects the

```
1  private void firePropertyChange (String propName,
2          Object oldValue, Object newValue) {
3          PropertyChangeEvent evt = new PropertyChangeEvent();
4          ...
5        }
6  /* Removes a time series from the map and
7     fires a TS_REMOVED PropertyChangeEvent */
8  public removeTS (String name) {
9          boolean fireChanged = false;
10         ...
11         if (fireChanged)
12          firePropertyChange(TS_REMOVED, name, name);
13  }
14  /* Removes all time series from the map and
15     fires an ALL_TS_REMOVED PropertyChangeEvent */
16  public removeAllTS() {
17    ...
18    firePropertyChange(ALL_TS_REMOVED, null, null);
```

Table 1: Example illustrating the importance of utilizing class-level contextual information.

1Equal contribution.
2Corresponding author.
target function to all other functions in the same class to aggregate contextual information. Bansal et al. (2021) present a project-level encoder to augment existing models by introducing contextual information.

Although the above methods have shown promising performance, the way they introduce contextual information is somewhat crude. Since not all surrounding functions are closely related to the target function, indiscriminately utilizing the whole context may introduce noise, which would hurt the model performance. We propose that considering function relations is a better way to leverage the contextual information. For example, Table 1 presents three functions in a Java class. Within this class, we can observe two types of function relations. First, the function removeTS calls firePropertyChange in its function body. As we can see, the word "PropertyChangeEvent" in the human-written comment appears not in the target function, but in the callee function. Second, removeTS and removeAllTS perform very similar operations, and their comments are almost identical, with the exception of a few noun subjects. This example illustrates that the information required to generate a comment may be located outside the boundary of the target code snippet and within the related functions.

Motivated by the above observation, we define two types of relations between a function pair: extractive relation and inductive relation. The extractive relation captures connections between source code snippets at two levels: call dependencies and semantic similarity, allowing us to derive external knowledge directly from the relevant code snippets. The inductive relation captures common programming patterns within a class. We observe that developers usually create similar comments for functions that conform to a specific programming pattern. Therefore, comments of functions that have inductive relation to the target function can be used as a template to guide the target comment generation.

In this paper, we propose a graph-based encoder-decoder learning framework for code comment generation. Our approach is based on a common scenario where only a few functions in the class file are documented. We construct a heterogeneous graph to model both the extractive relation and inductive relation among functions within a class file. In the encoding stage, we encode all functions and available comments using bi-GRU. Then, we design an intra-class relational GAT encoder to aggregate information and perform a fusion of both types of relations via a cross-graph mechanism. In the decoding stage, we employ a GRU decoder with a by-pointer mechanism to generate a comment utilizing the encoding results.

To evaluate the performance of our approach, we gather a Java dataset that preserves the class structure. We conduct experiments on this dataset and perform evaluation using automatic and human evaluation metrics. The experimental results show that our model outperforms prior methods by a significant margin, which demonstrate the effectiveness of our proposed framework.

2 Related Works

Early efforts on code comment generation are template-based or information retrieval (IR) based approaches (Sridhara et al., 2010; Haiduc et al., 2010a,b; Eddy et al., 2013; Rodeghero et al., 2014; McBurney and McMillan, 2014). In recent years, the neural encoder-decoder architecture is employed to the code comment generation field, which was designed for neural machine translation (NMT) task originally. CodeNN (Iyer et al., 2016) is an early work that attempts to adopt the encoder-decoder architecture for generating code comments. Followed works develop a variety of models by introducing the Abstract Syntax Tree (AST) to extract structural information of the source code (Hu et al., 2018a; Alon et al., 2019; Allamanis et al., 2018; Liang and Zhu, 2018; LeClair et al., 2019a). More recently, novel code representations are learned via well-designed encoders, such as GNN-based encoders (LeClair et al., 2020; Zhang et al., 2022) and pre-training encoders (Ahmad et al., 2020; Zügner et al., 2021; Guo et al., 2022).

Hybrid methods that integrate the IR-based and neural-based techniques proved to perform well on the code comment generation task. Zhang et al. (2020) retrieve two similar code snippets of the target function at syntax and semantics levels, then utilize their encoding information to generate comments in the decoding stage. Wei et al. (2020) retrieve the most similar code snippet and the corresponding comment to assist the generation process. Liu et al. (2021) retrieve the most similar code-comment pair and add it as auxiliary information to their proposed Hybrid GNN framework.

However, these works rarely utilize contextual information that is external to the target function.
Some of the most recent works make efforts to bridge this gap (Haque et al., 2020; Yu et al., 2020; Bansal et al., 2021). In contrast to these existing approaches, our method explores function relationships within a class and only incorporates related functions. This has the advantage of avoiding noise caused by irrelevant functions and focusing on the valuable contextual information.

3 Approach

This section introduces our proposed framework for code comment generation. Figure 1 illustrates an overview of our approach.

3.1 Relation Extraction

To focus on valuable class-level contextual information, we need to develop extraction rules for function relations. We define the extractive relation as (1) the call dependency; (2) the TF-IDF cosine similarity. Call dependencies can be extracted using the java-callgraph toolkit, and the TF-IDF cosine similarity between functions \( X^i \) and \( X^j \) is calculated as:

\[
s_{ij} = \frac{\text{tfidf}(X^i) \cdot \text{tfidf}(X^j)}{||\text{tfidf}(X^i)|| \cdot ||\text{tfidf}(X^j)||} \in [0, 1]
\]

If the similarity score \( s_{ij} > \alpha \), we consider an extractive relation between \( X^i \) and \( X^j \), where \( \alpha \) is a pre-defined threshold.

Towards the inductive relation, we summarize some common programming patterns and organize them into five heuristic rules based on extensive observations of open-source software projects. Formally, we consider an inductive relation between two functions if:

(R1) the verbs in function names are antonyms, with the same or no object entities;

(R2) the verbs in function names are the same, with overlapping object entities;

(R3) both functions have the same parameters as well as the same verbs in their names;

(R4) both functions have the same parameters as well as the same return type;

(R5) the return type of one function corresponds to the parameter type of another.

In (R1)-(R3), we conduct part-of-speech tagging on function names using the toolkit Stanford CoreNLP to identify verbs and noun entities. And we use the NLTK interface of WordNet to get antonyms of verbs in (R1). Appendix A provides several examples that correspond to the preceding rules.

3.2 Graph Construction

For each class, we build a graph structure that consists of two subgraphs, called function graph and comment graph.
comment graph. The node of the function graph represents each function, while the node of the comment graph represents the corresponding comment. Since only a small fraction of comments in the class are known, we use function names to replace unknown comments. According to the previously defined extraction rules, we add edges between the corresponding function nodes if a pair of functions fulfill the extractive relation, and between comment nodes if they satisfy the inductive relation. Formally, we define a graph

\[ G = \{ (v_i, r_{fun}, v_j) \cup (u_i, r_{com}, u_j) \}, \]

where \( v \in \mathcal{V}_f \) is the node of function, \( u \in \mathcal{V}_c \) is the node of comment or function name, \( \mathcal{V}_f, \mathcal{V}_c \) are the sets of function nodes and comment nodes, and \( r_{fun}, r_{com} \) denote edges that represent the extractive relation between functions, and the inductive relation between comments, respectively.

### 3.3 Local Encoder

Our model contains two local encoders, a function encoder and a comment encoder. They extract features from functions and comments separately. The function encoder employs a bi-GRU (Cho et al., 2014) to convert the source code sequence \( \{x_1, \ldots, x_n\} \) into numerical vectors \( Z = \{z_1, \ldots, z_n\} \), where \( z_i = [\overline{z_i} \mid \overline{z_{n-i}}] \) is the concatenation of the hidden states from both directions. We take \( Z \) as the representation of the input function. The comment encoder also apply a bi-GRU to the comment sequence \( \{w_1, \ldots, w_m\} \), and produce hidden states \( \{r_1, \ldots, r_m\} \). The last hidden state \( r_m \) is considered as the comment representation.

### 3.4 Intra-class Relational GAT

We propose an intra-class relational graph attention network that performs on the previously constructed graph.

**Node Initialization** For function nodes, we first apply the function encoder to obtain their representations \( \{Z_1, \ldots, Z_K\} \), where \( Z_i = \{z_{i1}, \ldots, z_{in}\} \) represents the \( i \)-th function. Then, we use a target-aware attention mechanism to focus on information in other functions that is beneficial to the target function. We take the last hidden state as the representation of the target function \( Z_t \), which can be denoted as \( z_t \), and use it to compute the attention weight as:

\[
\alpha_{t,ij} = \frac{\exp(z_t^T W_t z_{ij})}{\sum_{k=1}^{m_t} \exp(z_t^T W_t z_{ik})},
\]

where \( W_t \) is a learnable parameter. The attention score \( \alpha_{t,ij} \) measures the similarity between the target function and the \( j \)-th token in the code sequence of the \( i \)-th function. Then, we compute the weighted sum of all elements in \( Z_i \):

\[
f_t^0 = \sum_{j=1}^{n_i} \alpha_{t,ij} z_{ij},
\]

and take it as the initial representation of the function node \( v_i \). Finally, we obtain a set of target-aware initial vectors representing function nodes, which is denoted as \( \{f_t^0 \mid i : v_i \in \mathcal{V}_f\} \). For comment nodes, the last hidden states are used directly as the initial node representations, and we denote them as \( \{c_t^0 \mid i : u_i \in \mathcal{V}_c\} \).

**Cross-Graph Attention** We employ two separate GAT modules, one for the function graph and the other for the comment graph, which have the same structure. In order to interact information between these two graphs, we design a cross-graph attention mechanism that is applied to each layer of the GATs. Fig. 2 illustrates an example of this mechanism. Specifically, the \( l+1 \)-th layer of each GAT receives a set of messages \( \{f_{l+1}^i \mid i : v_i \in \mathcal{V}_f\} \) or \( \{c_{l+1}^i \mid i : u_i \in \mathcal{V}_c\} \) from the previous layer. Then, we obtain two output vectors \( f_{l+1}^{g,i}, c_{l+1}^{g,i} \) and two attention distributions calculated as:

\[
e_{ij}^f = \text{LeakyReLU}(a^T [W_a f_i^l \| W_a f_j^l]),
\]

\[
\beta_{ij} = \text{Softmax}(e_{ij}^f) = \frac{\exp(e_{ij}^f)}{\sum_k \exp(e_{ik}^f)},
\]

\[
e_{ij}^c = \text{LeakyReLU}(b^T [W_b c_i^l \| W_b c_j^l]),
\]

\[
\beta_{ij} = \text{Softmax}(e_{ij}^c) = \frac{\exp(e_{ij}^c)}{\sum_k \exp(e_{ik}^c)},
\]

where \( a, W_a, b, W_b \) are trainable parameters. Next, we exchange the attention weights and com-
pute two cross-graph vectors as:
\[ f_{\text{cross},i}^{l+1} = \sum_{j: u_j \in N_c(u_i)} \beta_{ij}^c W_d f_j^l, \]  
\[ c_{\text{cross},i}^{l+1} = \sum_{j: v_j \in N_f(v_i)} \beta_{ij}^f W_c c_j^l, \]  
where \( u_i \) is the corresponding comment node of function node \( v_i \), \( N_f(v_i) \) is a set of the neighboring function nodes of \( v_i \), \( N_c(u_i) \) is a set of the neighboring comment nodes of \( u_i \), and \( W_d, W_c \) are trainable parameters. Finally, we aggregate the cross-graph vector and the original GAT output vector to obtain an integral context vector that takes into account both extractive and inductive relations:
\[ f_{\text{aggr},i}^{l+1} = \tanh(W_f [f_{\text{cross},i}^{l+1}, f_i^{l+1}]). \]  
\[ c_{\text{aggr},i}^{l+1} = \tanh(W_c [c_{\text{cross},i}^{l+1}, c_i^{l+1}]), \]  
where \( W_f, W_c \) are trainable weights.

**Update Gate** Motivated by (Cho et al., 2014), we introduce an update gate to control how much information from the previous representation should be transferred to the current representation:
\[ g = \text{Sigmoid}(W_g f_{\text{aggr},i}^l || f_i^l), \]  
\[ f_i^{l+1} = g \ast f_{\text{aggr},i}^{l+1} + (1 - g) \ast f_i^l. \]  
We also obtain \( c_i^{l+1} \) by performing the same operation. These two representations are the outputs of the \( l + 1 \)-th layer. We repeat the above process \( L \) times and get the final node representations, denoted as \( f_i^L \) and \( c_i^L \). Finally, we concatenate the representation of the function node and its corresponding comment node as the output of our GAT encoder, which is denoted as \( g_i = f_i^L \| c_i^L \).

### 3.5 Decoder

The decoder employs a GRU to generate comment for the target function \( Z_t \). The initial hidden state is a concatenation of the last hidden state \( z_t \) from the function encoder and the final output \( g_t \) from the GAT encoder.

**Attention** We consider multiple context vectors: \( c_z t \) toward the output from the function encoder, \( c_r t \) toward the output from the comment encoder, and \( c_g t \) toward the output from the GAT encoder, which can be calculated as follows:
\[ \gamma_{tj} = \frac{\exp(h_t^T W_s \eta_j)}{\sum_k \exp(h_t^T W_s \eta_k)}, \]  
where \( W_s \) is a learnable parameter, \( h_t \) is the current decoder hidden state, and \( \eta_j \) represents the function encoder output \( z_j \), the comment encoder output \( r_j \), and the GAT encoder output \( g_j \), respectively.

**By-Pointer** Since both code snippets and known comments may contain words that are not in the vocabulary, a portion of the predicted tokens could be copied directly from them. Motivated by (See et al., 2017) and (Sun et al., 2018), we design a by-pointer mechanism to solve this problem. In the \( t \)-th time step, the decoder takes embedding \( y_t \) as input, and the copy distribution is formulated as:
\[ \lambda = \text{Sigmoid}(W_t [c_r t \| h_t \| y_t]), \]  
\[ P_{\text{copy}} = \lambda \ast \gamma + (1 - \lambda) \ast \gamma_f, \]  
where \( W_t \) is the trainable parameter. The \( \gamma_f \) is the attention distribution between the current hidden state \( h_t \) and source codes, \( \gamma_c \) is the attention distribution between \( h_t \) and known comments, both calculated by Eq (12). Additionally, the generative distribution over all vocabulary tokens is calculated based on \( h_t \) and three context vectors:
\[ P_{\text{gen}} = \text{Softmax}(W_v [h_t \| c_z t \| c_r t \| c_g t] + b_v), \]  
where \( W_v, b_v \) are trainable parameters. Finally, we obtain the prediction distribution as follows:
\[ \mu = \text{Sigmoid}(W_m [c_z t \| c_r t \| c_g t \| h_t \| y_t]), \]  
\[ P(w) = \mu \ast P_{\text{gen}} + (1 - \mu) \ast P_{\text{copy}} \]  
where \( W_m \) is the trainable parameter. This mechanism allows our model to both generate tokens from the vocabulary and copy tokens from two sources during inference.

### 4 Experimental Setup

#### 4.1 Dataset

Due to most public datasets only consist of independent code snippets, we collect a dataset from Google Code Archive that preserves class-level information. With the help of Sourcerer (Bajracharya et al., 2014), we are able to trace and recover the entire architecture of 1,000 real-world JAVA projects. We assume that only 10% of functions or at least one function have comments in each class. To determine which functions will be treated as commented, we use two different sampling settings. (1)
Random Sampling: we randomly sample 10% of functions in each class as commented based on the assumption that commenting is a stochastic behavior for developers; (2) Degree Sampling: since functions that connect to others more frequently often play a key role in programming, we calculate function degrees in the class graph and rank them in descending order. Then we sample the top 10% of functions as commented. After sampling, we split our dataset by projects according to a ratio of 8:1:1. The more detail of our dataset is provided in the Appendix B.

4.2 Baselines

Retrieval-based Models RandomCopy randomly copies comments from a known comment set. MaxCopy computes the ROUGE-L score between the golden comment and known comments, then copies the comment with the highest score. NNGen (Liu et al., 2018) is a IR-based method for generating commit messages that can also be used in the code comment generation task.

Generation-based Models Seq2Seq (Sutskever et al., 2014) is a bi-GRU with an attention mechanism. ASTGNN (LeClair et al., 2020) applys a GRU encoder for the source code sequence, a GCN (Kipf and Welling, 2017) encoder for the AST and a GRU decoder for generation. Rencos (Zhang et al., 2020) retrieves two similar functions from a code retrieval base to enhance the neural generation. ClassGAT (Yu et al., 2020) employs a local bi-GRU encoder and a global GNN encoder to obtain two different levels of function representation. The encoder outputs are fed into a GRU decoder with an attention and copy mechanism. CodeBERT (Feng et al., 2020) is a pre-trained model that can be adapted to a variety of NL-PL applications. GypSum (Wang et al., 2022) learns representations from source codes and ASTs using a pre-trained encoder and a GAT encoder. The encoding results are fused in a Transformer decoder to generate comments.

with Known Comments The baseline models mentioned above only work on the source code itself, whereas our approach incorporates known comments additionally. To explore the influence of known comments, we perform a modification that introduces them into multiple baselines. For Seq2Seq, we produce a comment by combining the target function representation and the weighted sum of known comment representations. Towards ClassGAT, we take the initial node representation as (i) a concatenation or (ii) a weighted sum of the function representations and their corresponding comment representations, then report the best performance. As for CodeBERT, we set its input as a concatenation of the target function and known comments within the class.

with CodeBERT We also incorporate CodeBERT into our model for verifying whether function relations still provide benefits when employing a strong pre-training model. Specifically, we use the CodeBERT and a transformer decoder to replace the bi-GRU encoder and the GRU decoder, respectively.

4.3 Implementation Details

The value of threshold $\alpha$ is set to 0.7. Word embeddings are randomly initialized, the size of embeddings and hidden states are set to 256. Both the encoder and decoder GRUs have a single layer and the GAT has 3 layers. We use Adam (Kingma and Ba, 2015) optimizer to train our model with the weight decay rate being 1e-6. We set the learning rate to 1e-4 and the dropout (Srivastava et al., 2014) rate is 0.3. There is also a scheduler that reduce learning rate when the BLEU on the validation set stops improving for 3 epochs, and the learning rate will not be less than 1e-6. All our experiments were trained on Nvidia A40 GPUs.

4.4 Evaluation Metrics

We evaluate the quality of generated comments based on BLEU (Papineni et al., 2002) and ROUGE-1, -2, -L (Lin, 2004). We also report 1,2,3,4-gram precisions to determine how many n-grams in the generated text overlap with the reference text. For human evaluation, we invited three experienced raters to score fifty samples randomly selected from the test dataset. For each generated comment, raters assign scores in three aspects: (i) Fluency, which measures comment quality in terms of grammaticality and readability; (ii) Relevance, which examines whether the generated comment accurately summarizes the functionality of the code snippet; (iii) Informativeness, which evaluates whether the comment offers concrete information that is free of redundancy or repetition. These human evaluation metrics have a scale of 0 to 2 (where 2 indicates highly satisfied and 0 means highly unsatisfied).
Table 2: Comparison between our model and baselines. "KC" refers to the known comments.

<table>
<thead>
<tr>
<th>Model</th>
<th>BLEU</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
<th>ROUGE-1</th>
<th>ROUGE-2</th>
<th>ROUGE-L</th>
</tr>
</thead>
<tbody>
<tr>
<td>RandomCopy</td>
<td>13.08</td>
<td>30.0</td>
<td>14.4</td>
<td>9.2</td>
<td>7.4</td>
<td>30.72</td>
<td>14.04</td>
<td>36.05</td>
</tr>
<tr>
<td>MaxCopy</td>
<td>14.65</td>
<td>32.2</td>
<td>16.2</td>
<td>10.4</td>
<td>8.4</td>
<td>33.60</td>
<td>16.20</td>
<td>32.26</td>
</tr>
<tr>
<td>NNGen</td>
<td>16.11</td>
<td>29.1</td>
<td>16.4</td>
<td>13.1</td>
<td>12.1</td>
<td>30.29</td>
<td>17.68</td>
<td>29.48</td>
</tr>
<tr>
<td>Seq2Seq</td>
<td>15.10</td>
<td>37.1</td>
<td>18.1</td>
<td>11.9</td>
<td>9.9</td>
<td>37.49</td>
<td>18.47</td>
<td>36.05</td>
</tr>
<tr>
<td>ASTGCN</td>
<td>16.05</td>
<td>39.5</td>
<td>18.9</td>
<td>12.2</td>
<td>9.6</td>
<td>41.76</td>
<td>20.65</td>
<td>39.71</td>
</tr>
<tr>
<td>Rencos</td>
<td>16.08</td>
<td>39.1</td>
<td>20.9</td>
<td>14.4</td>
<td>12.3</td>
<td>37.83</td>
<td>20.06</td>
<td>36.63</td>
</tr>
<tr>
<td>ClassGAT</td>
<td>17.38</td>
<td>40.7</td>
<td>20.5</td>
<td>13.6</td>
<td>11.1</td>
<td>42.70</td>
<td>21.65</td>
<td>40.51</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>18.29</td>
<td>46.9</td>
<td>25.2</td>
<td>16.8</td>
<td>13.5</td>
<td>45.00</td>
<td>24.01</td>
<td>43.25</td>
</tr>
<tr>
<td>GypSum</td>
<td>18.95</td>
<td>44.6</td>
<td>23.9</td>
<td>15.5</td>
<td>12.0</td>
<td>45.44</td>
<td>24.22</td>
<td>43.60</td>
</tr>
<tr>
<td>Seq2Seq+KC</td>
<td>16.51</td>
<td>39.4</td>
<td>20.4</td>
<td>12.9</td>
<td>10.6</td>
<td>39.82</td>
<td>20.76</td>
<td>38.37</td>
</tr>
<tr>
<td>ClassGAT+KC</td>
<td>18.52</td>
<td>42.8</td>
<td>22.1</td>
<td>14.9</td>
<td>12.6</td>
<td>43.34</td>
<td>22.67</td>
<td>41.05</td>
</tr>
<tr>
<td>CodeBERT+KC</td>
<td>19.64</td>
<td>51.9</td>
<td>29.0</td>
<td>18.3</td>
<td>13.6</td>
<td>49.95</td>
<td>27.54</td>
<td>47.87</td>
</tr>
<tr>
<td>Ours</td>
<td>21.39</td>
<td>47.3</td>
<td>26.6</td>
<td>18.6</td>
<td>15.9</td>
<td>46.78</td>
<td>25.72</td>
<td>44.87</td>
</tr>
<tr>
<td>+ CodeBERT</td>
<td>25.60</td>
<td>51.9</td>
<td>31.8</td>
<td>22.8</td>
<td>18.8</td>
<td>51.87</td>
<td>31.54</td>
<td>49.89</td>
</tr>
</tbody>
</table>

Table 3: Results of human evaluation (standard deviation in parentheses).

<table>
<thead>
<tr>
<th>Model</th>
<th>Fluency</th>
<th>Relevance</th>
<th>Informativeness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq2Seq</td>
<td>1.19 (±0.86)</td>
<td>0.72 (±0.75)</td>
<td>0.93 (±0.79)</td>
</tr>
<tr>
<td>ClassGAT</td>
<td>1.27 (±0.82)</td>
<td>0.81 (±0.75)</td>
<td>1.01 (±0.76)</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>1.39 (±0.78)</td>
<td>1.13 (±0.77)</td>
<td>1.31 (±0.75)</td>
</tr>
<tr>
<td>Ours</td>
<td>1.54 (±0.77)</td>
<td>1.21 (±0.87)</td>
<td>1.36 (±0.72)</td>
</tr>
</tbody>
</table>

5 Results and Analysis

5.1 Automatic Evaluation

The comparative results are summarized in Table 3. Overall, our model outperforms all baselines by a large margin. Retrieval-based models have relatively poor results since they do not adequately exploit the semantic information of the source code. In comparison, generation-based models perform better. ASTGCN surpasses Seq2Seq by incorporating structural information from the AST. Rencos and ClassGAT improve their performance with the assistance of external information. CodeBERT and GypSum exceed other baselines by utilizing their extensive pre-training knowledge. After aggregating related contextual information, our model outperforms all baseline models. This suggests that considering function relations is an effective way to enhance the comprehension of the target function.

We discover that introducing known comments can improve the performance of some baselines. As shown in Table 2, all of the methods achieve an improvement on BLEU and ROUGE. Due to the vast knowledge gained during the pre-training process, CodeBERT significantly improves ROUGE-L from 43.25 to 47.87 (+ 4.62%). This result suggests that known comments from the class context can help with code comment generation. We also analyze the effect of known comments on our model in the Appendix C.

Although these models present competitive performance with the incorporation of known comments, our model equipped with CodeBERT still manages to make a further improvement and achieves the best BLEU and ROUGE scores among all the involved models. This shows that combining our framework with the pre-trained model can effectively absorb both the contextual information and the pre-training knowledge, which allows our approach to collaborate with more advanced pre-trained models in the future.

5.2 Human Evaluation

We further conduct human evaluation to assess the quality of comments generated by different models, as shown in Table 3. Our model surpasses the baseline models on all metrics. The Seq2Seq model has a much lower score than others, because it only utilizes local information contained in the source code, whereas other models incorporate contextual information or pre-training knowledge as well. Since the by-pointer mechanism enables our model to copy tokens from both the source code and known comments, it significantly improves the fluency of generated comments. Besides, the highest relevance and informativeness score indicates that our model can effectively summarize the behavior of a given function.

5.3 Ablation Study

To examine the contribution of components in our framework, we evaluate the performance after removing each of them, as shown in Table 4. We discover that removing any of the modules has a neg-
Table 4: Ablation study results of our approach.

<table>
<thead>
<tr>
<th>Model</th>
<th>BLEU</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>ROUGE-1</th>
<th>ROUGE-2</th>
<th>ROUGE-L</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours</td>
<td>21.39</td>
<td>47.73</td>
<td>26.6</td>
<td>18.6</td>
<td>15.9</td>
<td>46.78</td>
<td>25.72</td>
<td>44.87</td>
</tr>
<tr>
<td>w/o function encoder</td>
<td>16.39</td>
<td>42.2</td>
<td>21.4</td>
<td>13.1</td>
<td>10.0</td>
<td>43.45</td>
<td>22.44</td>
<td>41.40</td>
</tr>
<tr>
<td>w/o GAT encoder</td>
<td>16.75</td>
<td>44.3</td>
<td>22.0</td>
<td>13.4</td>
<td>10.1</td>
<td>45.09</td>
<td>22.20</td>
<td>42.84</td>
</tr>
<tr>
<td>w/o target-aware attention</td>
<td>19.35</td>
<td>46.3</td>
<td>24.7</td>
<td>16.2</td>
<td>12.9</td>
<td>46.41</td>
<td>24.72</td>
<td>44.24</td>
</tr>
<tr>
<td>w/o cross-graph attention</td>
<td>18.81</td>
<td>45.0</td>
<td>24.1</td>
<td>15.8</td>
<td>12.6</td>
<td>45.14</td>
<td>23.74</td>
<td>43.06</td>
</tr>
<tr>
<td>w/o by-pointer mechanism</td>
<td>20.27</td>
<td>46.9</td>
<td>25.7</td>
<td>17.4</td>
<td>14.7</td>
<td>46.58</td>
<td>25.20</td>
<td>44.53</td>
</tr>
</tbody>
</table>

Table 5: Impact of different types of edges.

<table>
<thead>
<tr>
<th>Model</th>
<th>BLEU</th>
<th>ROUGE-1</th>
<th>ROUGE-2</th>
<th>ROUGE-L</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours</td>
<td>21.39</td>
<td>46.78</td>
<td>25.72</td>
<td>44.87</td>
</tr>
<tr>
<td>w/o TF-IDFSim</td>
<td>19.21</td>
<td>46.11</td>
<td>24.64</td>
<td>43.95</td>
</tr>
<tr>
<td>w/o R1</td>
<td>19.70</td>
<td>45.86</td>
<td>24.09</td>
<td>43.81</td>
</tr>
<tr>
<td>w/o R2</td>
<td>20.52</td>
<td>46.48</td>
<td>25.67</td>
<td>44.37</td>
</tr>
<tr>
<td>w/o R3</td>
<td>20.28</td>
<td>46.22</td>
<td>24.98</td>
<td>44.03</td>
</tr>
<tr>
<td>w/o R4</td>
<td>20.48</td>
<td>46.39</td>
<td>25.09</td>
<td>44.30</td>
</tr>
<tr>
<td>w/o R5</td>
<td>21.09</td>
<td>46.68</td>
<td>25.72</td>
<td>44.54</td>
</tr>
<tr>
<td>fully connected</td>
<td>17.73</td>
<td>45.60</td>
<td>23.73</td>
<td>43.42</td>
</tr>
</tbody>
</table>

5.4 Threshold $\alpha$

The hyper-parameter $\alpha$ determines the lower limit of TF-IDF similarity scores. To explore how model performance varies with $\alpha$, we run a series of experiments with different values of $\alpha$, while keeping other hyper-parameters constant. Fig. 3 shows the corresponding results. It illustrate that $\alpha = 0.7$ achieves peak performance in both BLEU and ROUGE-L. When $\alpha$ is equal to 0.5 or 0.9, our model performs poorly in both metrics. This may be because when the $\alpha$ is too small or too large, there are too many or too few functions associated with the target function, and the model is unable to make effective use of contextual information.

5.5 Sampling Settings

To investigate the impact on our approach when programmers select functions to be commented in different ways, we conduct a series of experiments under random and degree sampling settings. The experimental results are reported in Fig. 4. Compared to the competitive baselines, our model presents better performance under both settings. Since our model is able to capture function relations within a class, even randomly commenting functions can help improve the quality of generated comments. Furthermore, it clearly shows that our model performs much better under degree sampling than random sampling, due to the reason that commenting functions with higher degrees can benefit more functions in the same class. This finding implies that when engineering in the real world, it may be a good idea to start by writing comments for functions with higher degrees.

5.6 Relation Types

To examine the utility of edges constructed by TF-IDF similarity and five types of inductive relations, we remove them from the class graph to observe how it affects model performance, as presented in Table 5. It demonstrates that removing either type of edge leads to a drop in BLEU and ROUGE scores. Specifically, the performance degradation is greatest after dropping edges of type TF-IDF Sim and R1, indicating that they are the most influential relation types. While removing R5 edges has the least impact on model performance, this can...
public boolean isCurrent() {  
    boolean result = false;
    if (this.getNumber().intValue() == EasyCalendar.getOne().getCurrentYear()) {  
        result = true;
    }  
    return result;
}

public Vector parse(final String str, char separator) {  
    if (str == null) {  
        return new Vector();
    }  
    return parse(str.toCharArray(), separator);
}

gets the current month, but only if this year is the current one

public Month getCurrentMonth() {  
    Month month = null;
    if (isCurrent()) {  
        int currentMonthNumber = EasyCalendar.getOne().getCurrentMonth();
        month = getMonth(currentMonthNumber);
    }
    return month;
}

extracts a list of name value pairs from the given string

Table 6: Examples of the source codes, graph structure and generated comments. "T" refers to the target function and "C" refers to the commented function in the class context.

be attributed to the low occurrence of this relation type in the dataset. Furthermore, we conduct an experiment to investigate the impact of exploiting contextual information in a cruder manner. To be more specific, rather than modeling functional relations, we construct a fully connected graph to introduce the entire class context. Our model suffers greatly as a result of this operation, with the BLEU and ROUGE-L dropping 3.66% and 1.45%, respectively. This performance loss verifies the effectiveness of our design for utilizing class-level contextual information.

5.7 Case Study

Table 6 shows two examples of generated code comments. In the first case, there is a commented function in the class that is the caller of the target function. The second half of its comment provides a description of the target function isCurrent. Although there is no direct connection between these two functions in the comment graph, our model is still able to extract information from the known comment through the cross-graph attention mechanism and generate a high-quality comment. In contrast, baseline models do not capture the true functionality of the source code and their generation results have a large deviation from the original intention.

In the second case, it is difficult to figure out the purpose of target function solely from the source code. Since there is a defined (R1) pattern between the target function and a commented function from the class context, the constructed edge in the comment graph allows our model to aggregate comment of this related function. Therefore, our model successfully generates "a list of name value pairs", whereas other models fail to capture this key information and produce meaningless comments. Moreover, it is worth noting that our model is unaffected by irrelevant information in the known comment, yielding the true object "string" rather than "array of characters". The final output of our model is exactly same as the human-written comment.

6 Conclusion

In this paper, we propose a graph-based learning framework for code comment generation. Our approach targets a practical scenario where only a few functions in the class file have human-written comments. To identify valuable information from the class context, we model function relations and develop a graph attention network to aggregate class-level contextual information. We conducted experiments on Java programs collected from real-world projects and the results demonstrate that our approach outperforms prior methods.
Limitations

There are four main limitations of our work. First, we only evaluate our model on Java code snippets. Although we expect that our approach could be generalized to other programming languages, further experiments are required to confirm this hypothesis. Second, our model does not utilize syntactic information (e.g., ASTs) of the source code. Thus, our next effort will incorporate this type of information into our framework to advance comment generation. Third, we do not employ Transformers in our approach due to limited resources. This will also be left to our future work. Fourth, in comparison with the widely used datasets TL-CodeSum (Hu et al., 2018b), CodeSearchNet (Husain et al., 2019) and Funcom (LeClair et al., 2019b), the size of our collected dataset is relatively small (Table 7). A large-scale code comment generation dataset that retains class structure information is needed in future studies.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Ours</th>
<th>TL-CodeSum</th>
<th>CodeSearchNet</th>
<th>Funcom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples</td>
<td>40,328</td>
<td>87,136</td>
<td>496,688</td>
<td>2.1 M</td>
</tr>
</tbody>
</table>

Table 7: Number of dataset examples.

Acknowledgements

This work is supported in part by National Key R&D Program of China (No. 2020AAA0106600) and NSFC (62161160339). We would like to thank the anonymous reviewers for their insightful comments and helpful suggestions.

References


A Examples of the inductive relation

Table 8 shows examples that correspond to the five inductive relation rules defined in Section 3.1.

B Dataset

In order to better suit the scenario of our task, only well-commented JAVA classes are retained, which means classes containing more than three functions and at least 70% of them have manually written comments. The detailed statistics of our dataset are shown in Table 9. Figure 5 shows the length distribution of target function and golden comment of our dataset. The length of functions is basically less than 100 and the length of comments are mainly between 3 and 25.

The data is preprocessed in following ways: for each function, (1) we extract the summative content in the Javadoc and take the first sentence as the comment; (2) we remove all the format controlling tokens and only retain comments having at least three words; (3) we serialize the function-comment pairs, remove non-alphabetical characters, and split tokens written in camelCase or underscore style; (4) we truncate the source code sequences to 200 tokens.

C Known Comments

As shown in the section 5.1, incorporating known comments into comment generation can significantly improve baseline model performance. To further demonstrate the effect of known comments, we remove them from our model and evaluate model performance. Specifically, we use function names instead of known comments for comment nodes during the graph construction step while leaving other settings unchanged. The experimental results are shown in Table 10. It illustrates that removing known comments reduces BLEU and ROUGE-L scores by 4% and 3%, respectively, in-
/* Adds a service to the framework. */
public void addService (Service service) {
    Settings settings = service.getSettings();
servicesMap.put(settings.getName(), service);
    settingsMap.put(settings.getName(), settings);
}

/* Removes a service from the framework. */
public void removeService (Service service) {
    Settings settings = service.getSettings();
servicesMap.remove(settings.getName());
servicesMap.remove(settings.getHeader());
}

/* Get degrees of latitude in different formats. */
public String getLatDeg (int format) {
    switch (format) {
        case DD:
            return Double.toString(getLatDec());
        case DMS:
            return getDMS(getLatDec(), 0, format);
        default:
            return "";
    }
}

/* Get degrees of longitude in different formats. */
public String getLonDeg (int format) {
    switch (format) {
        case DD:
            return Double.toString(getLonDec());
        case DMS:
            return (((getLonDec() < 100.0) &&
                (getLonDec() > -100.0)) ? "0": "") +
                getDMS(getLonDec(), 0, format);
        default:
            return "";
    }
}

/* Method to calculate the bearing of a waypoint. */
public double getBearing (CWPoint dest) {
    if (!this.isValid() || dest == null || !dest.isValid())
        return 361;
    return GeodeticCalculator.calculateBearing(TransformCoordinates.WGS84, this, dest);
}

/* Method to calculate the distance to a waypoint. */
public double getDistance (CWPoint dest) {
    return GeodeticCalculator.calculateDistance(TransformCoordinates.WGS84, this, dest) / 1000.0;
}

/* Returns the Action at the specified index. */
public Action get (int i) {
    return (Action)m_actions.get(i);
}

/* Removes the Action at the specified index. */
public Action remove (int i) {
    return (Action)m_actions.remove(i);
}

/* Add a Log to the list. */
public int addLog (Log log) {
    resetRecommendations();
    if (log != null && log.getLogType() != null) {
        return merge(log);
    }
    return -1;
}

/* Get the Log at a certain position in the list. */
public Log getLog (int i) {
    return logList.get(i);
}

<table>
<thead>
<tr>
<th>Item</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>3,344</td>
</tr>
<tr>
<td>Functions</td>
<td>40,328</td>
</tr>
<tr>
<td>Training examples</td>
<td>25,247</td>
</tr>
<tr>
<td>Validation examples</td>
<td>3,900</td>
</tr>
<tr>
<td>Test examples</td>
<td>2,770</td>
</tr>
<tr>
<td>Avg functions per class</td>
<td>12.4</td>
</tr>
<tr>
<td>Avg tokens per function</td>
<td>62.8</td>
</tr>
<tr>
<td>Avg tokens per comment</td>
<td>8.14</td>
</tr>
</tbody>
</table>

Table 8: Examples of function pairs with the inductive relation

<table>
<thead>
<tr>
<th>Item</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>BLEU</td>
</tr>
<tr>
<td>Ours</td>
<td>21.39</td>
</tr>
<tr>
<td>w/o KC</td>
<td>17.55</td>
</tr>
</tbody>
</table>

Table 9: Statistics of Our Dataset

indicating that known comments are quite essential for our model to capture code features and generate accurate comments.