Modeling Function Relation for Automatic Code Comment Generation

Anonymous ACL submission

Abstract

Comments are essential for software maintenance and comprehension. However, comments are often missing, mismatched or outdated in software projects. This paper presents a novel approach to automatically generate descriptive comments for methods and functions. Our work targets a practical problem where hand-written comments are only available for a few methods in a source file – a common problem seen in real-world software development. We develop a novel learning framework to model the code relation among methods based on graph neural networks. Our model learns to utilize the partially contextual information extracted from the existing comments to generate missing comments for all methods in a source file. We evaluate our approach by applying it to Java programs. Experimental results show that our approach outperforms prior methods by a large margin by generating comments that are judged to be helpful by human evaluators and of a higher quality measured by quantified metrics.

1 Introduction

Providing appropriate and adequate comments in the source code is important for software maintenance and comprehension (Xia et al., 2018). However, comments are often missing, incomplete or outdated in real-life software projects (Fluri et al., 2007). One solution to tackle this issue is to automatically generate descriptive comments from the source code.

Prior work in automatic code comment generation processes the target code region (e.g., functions or basic blocks) in isolation (Hu et al., 2018a; Alon et al., 2019; LeClair et al., 2020; Yu et al., 2020; Zhang et al., 2020). They often do not utilize the developer-written comments of other functions presented in the same source file. However, code components within the same source file are often closely related (McConnell, 2004). Such relation

```
1  /* Get the Action-Event associations for the
       current Event and create an XML Event
       definition. */
2  private ResultSet getEventDefinition () {
3          ...
4          return actevtValues;
5  }
6  /* Get the Trigger-Action associations for
       the current Action and create an XML
       Action definition. */
7  private ResultSet getActionDefinition () {
8          ...
9          return trigactValues;
10 }
```

Figure 1: An example to illustrate the implicit relationship between functions in a JAVA class.

is widely seen in object programming languages like Java and C++, where function implementations of a class are typically coded in the same source file. We argue that the human-written comments of other methods within the same source file can provide useful contextual information to generate missing comments of other functions or methods of the same file and hence cannot be ignored for code comment generation.

Some of the most recent works attempt to generate code comments that incorporates class level context. For example, the method presented in (Haque et al., 2020) first encodes each function of the same class with GRU (Cho et al., 2014) and then applies an attention mechanism towards the encoded functions during the generation process. Other work constructs a call graph to connect the functions within a class and applies a graph neural network (GNN) to extract information to generate code comments (Yu et al., 2020). These studies demonstrate the usefulness of exploiting class-level context to generate comments for functions.

While promising, existing techniques apply rough and coarse methods to model functions of the class. They neglect the diverse and subtle relationship between class methods. Because not all the function implementations are closely related,

indiscriminately utilizing all functions can introduce noise, which in turn degrades the performance of a machine learning method. Furthermore, we observe that there are other implicit relationships between functions, which can be useful in assisting code comment generation but are overlooked by prior work. As an example, consider Figure 1 that shows a Java class example from a real-world project. Here, methods `getEventDefinition` and `getActionDefinition` do not invoke each other in their function body. However, the two functions have closely related semantics because they both implement a get interface. Looking at the developer's comments closely, we see that the text descriptions are also similar, albeit the subjects in the sentence are different. This example shows that one can utilize the written comment of one function (or method) to automatically generate comments for another. However, doing so require carefully modeling and capturing the implicit relationship among functions. Prior work cannot do this because they only capture the function calling relation. Our work aims to bridge this gap.

In light of the observation described above, our approach models both explicit (like function calls) and implicit relationships of methods (such as two functions implementing similar operations). We want to model implicit relationships because we wish to capture the common programming idioms and patterns, where programmers often write pairing functions with closely related functionalities. For example, a Java class that implements a `read` related function is likely to also provide a `write` interface to access the same or other data members of the class. Figure 1 gives another example of this programming pattern, where the class implements a get method to access class members of `Event` and `Action` types.

This paper presents a new framework for automatic code generation by leveraging both explicit and implicit relationships. Our approach utilizes both relationships to extract key information from the source file to generate the intent description of a target function. To this end, we develop an encoder-decoder framework based on the GNN architecture. We do so by first encoding the the explicit and implicit code relationships as a heterogeneous graph. We then apply a GNN to enable different types of relationships to guide and communicate with each other. In the encoding stage, we design an attention-interactive GNN to embed all functions and the available comments within a class file. We then use a bi-GRU module to embed the source code of the target method. In the decoding stage, we use an attentive GRU decoder. To generate comments, we employ a by-copy mechanism to copy words from the code implementation of the target function and existing comments of other related functions.

We evaluate our approach on a JAVA dataset using automatic and human evaluation metrics. Experimental results show that our framework can generalize to different settings. It can efficiently capture both explicit and implicit relationships between methods. It outperforms prior approaches by generating comments that are judged to be more accurate by human evaluators and of a higher quality measured by automatic, quantified metrics.

This paper makes the following contributions:

- It is the first work to leverage multiple code relationships between functions of a class file to automatically generate code comments by utilizing partially presented comments.
- It presents a novel GNN model with an attention-interactive mechanism for function-level code comment generation.

2 Our Approach

2.1 Relation Extraction

In order to distinguish explicit and implicit relationships, we need to respectively extract them from the source code. As explicit relationship mainly consists of function calls, we use available toolkit to extract them. We observe that there is usually a syntactic pattern lying in function names between functions who have implicit relationship. Therefore, we propose two heuristic rules targeting function names to extract implicit relationship:

- If the verbs in function names are antonyms and they share the same object entities or there are no object entities in their names, we will consider an implicit relationship. This rule captures pairing functions such as `start` and `stop`, `add` and `delete`.
- If the verbs in function names are the same and their object entities have overlap, we will consider an implicit relationship. This rule captures pairing functions such as in Figure 1.

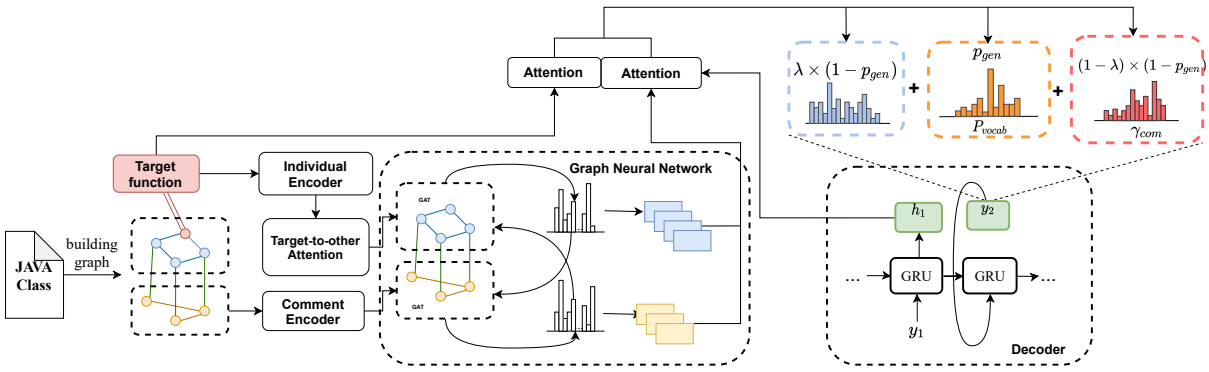

Figure 2: The overall architecture of our approach.

2.2 Graph Construction

In the task scenario, only a small part of comments are already presented and we do not have to model them independently from the functions. As a comment can be viewed as a super detailed and long function name, we mix the known comments with function names together, all as "names". Eventually, we build a heterogeneous graph structure to model both functions and their "names". Given a graph $G = \{(f_i, r, c_i) \cup (f_i, r_{explicit}, f_j) \cup (c_i, r_{implicit}, c_j)\}$, where vertex $f_i \in F$ represents a function and vertex $c_i \in C$ represents its name. We define three types of edges, where edge $r_{explicit}$ represents an explicit connection between functions, which is a two-way function calling relationship and $r_{implicit}$ represents an implicit connection between function names, which is determined by our heuristic rules. We also add an edge between a function and its name.

2.3 Individual Encoder

Our individual encoder extracts features from source codes of the target function. Given the source code of a function $X = (x_1, x_2 ... x_n)$, we use a bi-GRU (Cho et al., 2014) to encode it into a dense representation sequence $\{(\overrightarrow{q_1}, \overleftarrow{q_1}), ..., (\overrightarrow{q_n}, \overleftarrow{q_n})\}$, where $\overrightarrow{q_j}$ and $\overleftarrow{q_j}$ are the hidden states of x_j in both directions. We concatenate the hidden states of both directions as the final representation of the target function:

$$Q = \{\boldsymbol{q_1}, \boldsymbol{q_2} ... \boldsymbol{q_n}\} \quad (1)$$

$$\boldsymbol{q_i} = [\overrightarrow{\boldsymbol{q_i}} || \overleftarrow{\boldsymbol{q_i}}] \quad (2)$$

2.4 Graph Neural Network

2.4.1 Vertex Initialization

When encoding a function vertex, we want to maintain more useful information regarding the target function, so we apply a target-to-other attention mechanism. We firstly apply the individual encoder to each individual function and get their individual representation $\{Q_1, Q_2 ... Q_t, ... Q_K\}$ and then calculate the attention between target function and other functions in the class.

$$\beta_{ti} = \frac{exp(\boldsymbol{q_t^T W_l q_{ij}})}{\Sigma_{q_{ij} \in Q_i} exp(\boldsymbol{q_n^T W_l q_{ij}})} \quad (3)$$

where $\boldsymbol{q_t}$ is the last hidden state in Q_t and $\boldsymbol{W_l}$ is a learnable matrix. Then we use the weighted sum of individual representation Q_i as the initial vertex representation for functions, $\{\boldsymbol{g_i^0} | v_i \in V_f\}$:

$$\boldsymbol{g_i^0} = \Sigma_{q_{ij} \in Q_i} \beta_{ti} \boldsymbol{q_{ij}} \quad (4)$$

To encode a function name vertex, we apply a name encoder which shares the same structure with the individual encoder and concatenate the last hidden states in both directions as the initial vertex representation for function names, $\{\boldsymbol{d_i^0} | v_i \in V_c\}$.

2.4.2 Our Graph Attention Network

In order to let explicit and implicit relationships communicate with each other, we propose an attention-interactive module on GAT. In each layer, we first apply two individual GAT modules on functions and names and get two attention distributions, $\alpha_{ij,name}$ and $\alpha_{ij,func}$, calculated as :

$$e_{ij} = LeakyReLU((\boldsymbol{a}^T [\boldsymbol{W_a g_i^l} || \boldsymbol{W_a g_j^l}])) \quad (5)$$

$$\alpha_{ij} = softmax(e_{ij}) = \frac{exp(e_{ij})}{\Sigma_k exp(e_{ik})} \quad (6)$$

where $\boldsymbol{W_a}$ and $\boldsymbol{a}$ are learnable parameters and $[\cdot||\cdot]$ means the concatenation of two matrices. We then calculate the context vector under these two attention distributions.

$$\boldsymbol{g_{i,self}^{l+1}} = \Sigma_{j \in N(v_i,func)} \alpha_{ij,func} \boldsymbol{W_g} \boldsymbol{g_j^l} \quad (7)$$

$$\boldsymbol{g_{i,cross}^{l}} = \Sigma_{j \in N(v_i,name)} \alpha_{ij,name} \boldsymbol{W_{g,cross}} \boldsymbol{g_j^l} \quad (8)$$

where $N(v_i, func)$ is the neighbor nodes of v_i among functions, $N(v_i, name)$ is the neighbor nodes of v_i among function names and $\boldsymbol{W_{g,cross}}$ and $\boldsymbol{W_g}$ are learnable matrices. We aggregate these two context vectors to get an integral context vector considering both relationships.

$$\boldsymbol{g_{i,new}^{l}} = tanh((aggr([\boldsymbol{g_{i,cross}^{l}}||\boldsymbol{g_{i,self}^{l}}]))) \quad (9)$$

where $aggr$ is the aggregation method for two representations, we exploit addition, concatenation and linear transformation in experiments. Motivated by (Cho et al., 2014), we design an update gate to control the final output of each layer:

$$gate = sigmoid(\boldsymbol{W_{gate}}([\boldsymbol{g_{i,new}^{l}}||\boldsymbol{g_i^l}])) \quad (10)$$

$$\boldsymbol{g_{i,update}^{l+1}} = gate * \boldsymbol{g_i^l} + (1 - gate) * \boldsymbol{g_{i^l,new}} \quad (11)$$

In each layer, except the first, we apply a linear transformation to let functions directly communicate with their corresponding names.

$$\boldsymbol{g_i^{l+1}} = f([\boldsymbol{g_{i,update}^{l+1}}||\boldsymbol{d_{i,update}^{l+1}}]) \quad (12)$$

The calculations are same on function nodes and name nodes. We apply L layers and get the final outputs, represented as $\{\boldsymbol{g_i^L}|v_i \in V_f\}, \{\boldsymbol{d_i^L}|v_i \in V_c\}$.

2.5 Decoder

In the decoding phase, we follow the standard encoder-decoder framework and use a GRU module as decoder. We aggregate representations from both source codes and heterogeneous graph, concatenating the last hidden states of individual encoder and the last layer output of our graph neural network, as the initial state of the decoder GRU.

2.5.1 Attention

We leverage an attention mechanism to attend on both source codes and heterogeneous graph, deciding which part should be paid more attention to. Formally, we calculate multiple context vectors, cx_i towards the output of individual encoder, cg_i towards the final output of GNN and cy_i towards the output of name encoder, calculated as :

$$\boldsymbol{c_i} = \Sigma_{v_j \in G} \gamma_{ij} \boldsymbol{g_j} \quad (13)$$

$$\gamma_{ij} = \frac{exp(\boldsymbol{h_i^T W_s g_j})}{\Sigma_{v_k \in G} exp(\boldsymbol{h_i^T W_s g_k})} \quad (14)$$

where $\boldsymbol{W_s}$ is a trainable matrix. When calculating context vector towards graph, we respectively compute two context vectors for function nodes and name nodes and concatenate them as the final context vector $\boldsymbol{cg_i}$.

By-Copy Both source codes and known comments may contain information that is directly useful towards target comment. Motivated by (See et al., 2017; Sun et al., 2018), we propose a double-source copy mechanism which can copy words from source codes and known comments. In a standard pointer mechanism, the final prediction distribution is merged from a generative distribution and a copy distribution. Since we have two sources to copy from, we propose to merge the two distributions by a switch λ. In the i^{th} decoding step,

$$P_{copy} = \lambda * \gamma_{codes} + (1 - \lambda) * \gamma_{coms} \quad (15)$$

$$\lambda = \sigma(\boldsymbol{w_h^T h_i} + \boldsymbol{w_c^T c x_i} + \boldsymbol{w_y^T y_i}) \quad (16)$$

where $\boldsymbol{W_v}, \boldsymbol{b_v}, \boldsymbol{w_h}, \boldsymbol{w_c}, \boldsymbol{w_y}$ are all trainable parameters. γ_{codes} is the attention distribution between representation of target function and current hidden state, β_{coms} is the attention distribution between representation of known comments and current hidden state, calculated by Eq 14.

$$P_{vocab} = softmax(\boldsymbol{W_v}[\boldsymbol{h_i}||\boldsymbol{c_i}||\boldsymbol{cg_i}||\boldsymbol{cy_i}] + \boldsymbol{b_v}) \quad (17)$$

$$p_{gen} = \sigma(\boldsymbol{w_h'^T h_i} + \boldsymbol{w_c'^T c x_i} + \boldsymbol{w_{cy'}^T cy_i} + \boldsymbol{w_y'^T y_i}) \quad (18)$$

$$P(w) = p_{gen} P_{vocab} + (1 - p_{gen}) P_{copy} \quad (19)$$

where $\boldsymbol{W_v}, \boldsymbol{b_v'}, \boldsymbol{w_h'}, \boldsymbol{w_c'}, \boldsymbol{w_y'}$ are all trainable parameters. P_{vocab} is the normal output prediction distribution and p_{gen} serves as a switch that chooses between generating words normally from vocabulary or from copying.

3 Experimental Setup

3.1 Data Collection

Since most public datasets only contain function-level information by omitting function relationships in a class, we create our evaluation dataset

from real-life projects. Specifically, we collected our dataset from Google Code Archive and with the help of Sourcerer (Bajracharya et al., 2014), we manage to trace and recover the complete architecture of 1,000 JAVA projects. To better suit our task scenario, we only keep JAVA classes that are well commented, which contains more than 3 functions in the class and at least 70% of them have human written comments. In total, we have collected 3,344 JAVA files with 3,344 classes and 40,328 functions.

To prepare our dataset, we set the commented ratio as 10%, i.e., for each class, only 10% of functions or at least one function have comments. To decide which functions will be treated as commented, we propose two different experimental settings, random sampling and degree sampling. In addition to testing the adaptation ability of our approach, this will help us explore how the distribution of known comments will affect the generation quality.

Random Sampling Assuming that writing comments for programmers is a random behavior, without any prior patterns, we randomly sample 10% of functions in a class as commented.

Degree Sampling In a JAVA class, functions that are more frequently related to others usually play a vital role in the process of software developing, and they may provide strong assist to programmers. In this way, we sample top 10% functions according to its degree in the heterogeneous graph we build and use their comments as known comments.

After sampling the known comments, we split our dataset by projects and use 80% of the project data for training, 10% for validation and 10% for testing. It gives 25,247 functions in train set, 3,900 functions in valid set and 2,770 functions in test set.

During preprocessing, given a function, we extract the summative content in JavaDoc as the comment. We keep the first two sentences in the comment, remove all the format controlling tokens and only contain comments that have at least three words. After obtaining the function-comment pairs, we serialize them and remove all non-alphabetical letters and split identifiers that are written in the Camel or underscore style into dependent words. The average number of tokens in functions and comments are 62.8 and 8.14, respectively. The average number of functions in a class is 12.1.

3.2 Evaluation Metrics

We consider both automatic and human evaluation metrics in our experiments. For automatic evaluation, we adopt several widely used metrics in natural language generation tasks, including BLEU (Papineni et al., 2002), BLEU-1, BLEU-2, BLEU-3, BLEU-4, Rouge-1, Rouge-2 and Rouge-L (Lin, 2004). For human evaluation, we consider three aspects, fluency, relevance and informativeness. For all the three human evaluation metrics, we ask annotators to rate from 0 to 2 (where 2 indicates highly satisfied and 0 means highly unsatisfied).

3.3 Hyperparameters

We set both the embedding and the hidden dimension sizes to 256 and the word embeddings are randomly initialized. The layer of encoder and decoder GRU is 2. The GAT network has 3 layers. We set dropout (Srivastava et al., 2014) rate to 0.3, the weight decay rate to 1e-6. We train our model with Adam (Kingma and Ba, 2015) optimizer and learning rate 0.0001, and we use a scheduler that reduce learning rate by 0.1 every 15 epochs. We report the best scores over five different seeds.

3.4 Baselines

Copy models. This approach directly copies one comment from the known comment set as target comment. MaxCopy is the best performance that maximizes Rouge score between the target comment and the copied comment.

Retrieval models. Rencos (Zhang et al., 2020) retrieves a semantically similar and a syntactically similar functions from a code database, combines them together and applies a seq2seq model to generate comments.

Generation models. Seq2Seq (Sutskever et al., 2014) is a bi-directional GRU model with an attention mechanism. ASTGNN (LeClair et al., 2020) applies GCN (Kipf and Welling, 2017) on the AST structure and uses an attentive GRU decoder. ClassGAT (Yu et al., 2020) applies GAT module on the Call-Graph to extract a class-level representation while applies GRU on the target function to extract a function-level representation, combines them together and applies an attentive GRU decoder with pointer mechanism. We also conduct experiments with the pretraining model CodeBert (Feng et al., 2020) which demonstrates superior performance on a variety of code-related tasks. We use it to initial-

Table 1: Comparison between our model and baseline models. "KC" refers to the known comments in a class.

	Degree-Sampling							
Models	**BLEU**	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**	**Rouge-1**	**Rouge-2**	**Rouge-L**
MaxCopy	14.65	32.2	19.6	12.8	9.8	33.0	16.1	32.2
Rencos	14.15	33.6	16.6	11.7	10.2	32.0	15.7	31.5
Seq2Seq	15.45	38.2	19.0	12.3	10.0	37.4	19.1	37.0
ASTGCN	15.82	40.6	19.5	12.6	10.3	40.6	20.3	39.9
ClassGAT-CG	17.23	40.3	20.1	13.1	10.4	41.0	21.2	40.4
CodeBert	17.25	45.7	24.0	15.1	11.7	43.0	23.1	42.4
Seq2Seq + KC	17.44	40.4	20.8	12.7	9.5	41.9	22.1	41.4
ClassGAT + KC	18.58	43.1	22.6	15.0	12.3	42.8	23.4	41.9
CodeBert + KC	16.98	53.5	29.4	18.5	14.6	46.2	25.3	45.5
OurModel	**20.38**	**46.1**	**25.8**	**17.7**	**15.1**	**44.2**	**24.5**	**43.2**
w/ CodeBert	22.80	48.5	28.9	20.3	17.1	47.2	29.1	46.6

Table 2: Human evaluation results.

Models	**Fluency**	**Relevance**	**Informativeness**
Seq2Seq	1.52	1.30	0.84
ClassGAT	1.58	1.34	1.06
ClassGAT+KC	1.72	1.34	1.22
OurModel	**1.82**	**1.46**	**1.28**

ize a transformer encoder and train a decoder from scratch. We incoporate CodeBert into our model to verify whether function relationships still benefits when strong pretraining models are involved. We utilize CodeBert to embed functions instead of a trainable embedding layer and employ a transformer decoder instead of GRU.

The above baselines are designed to work from source code only, we thus modify them to introduce known comments into the models for our task setting. For Seq2Seq models, we follow (Zoph and Knight, 2016) and combines a weighted sum of known comment representations with the target function representation to generate comment. Towards ClassGAT model, we experiment with two different vertex initialization methods: concatenation of function and comment representations or a weighted sum of these two representations, and report the best performance of them. As for CodeBert, we concatenate the target function and known comments, separated by a SEP token, and use it as input.

4 Experimental Results

4.1 Main Results

Table 1 shows the performance of different models. Overall, our model outperforms all baselines by a large margin.

Both copy models and Rencos show relatively low performance under all metrics, indicating that known comments cannot be simply copy-paste to assist other functions within the same class as they may be beneficial in some cases but not all. In comparison, generation models typically perform better. ASTGCN outperforms sequential models by utilizing structural information from the AST. After introducing explicit relationship between functions within a class, ClassGAT outperforms all the non-pretrain models. The performance of generation models demonstrates that it is critical to incorporate both explicit and implicit relationships when determining the purpose of a function in class.

The baseline models also benefit from known comments. The Seq2Seq model with known comments(Seq2Seq+KC) improves the best-performing generation model by a little margin, due to that it does not utilize any other functions in the class as well as relationship between them, so it fails to extract more precise information from the known comments and the target function together. Introducing known comments into ClassGAT model leads to relatively strong performance, successfully combining known comments with given functions. However, this set of models only exchange information guided by function callings, while ignore the other possible relationship between functions. Although CodeBert shows strong performance due to the rich knowledge extracted from pretraining process, our model still manages to make a further improvement after introducing CodeBert into our model and achieve the best BLEU score among all the baseline models. The experiment results show that it is essential and beneficial to leverages both explicit

Table 3: Ablation results of our model.

Models	BLEU	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Rouge-1	Rouge-2	Rouge-L
OurModel	20.38	46.1	25.8	17.7	15.1	44.2	24.5	43.2
w/o individual encoder	15.10	40.8	19.6	11.2	8.1	41.2	20.9	40.8
w/o graph encoder	15.82	38.3	19.1	11.4	8.6	39.5	20.5	39.1
w/o by-copy	19.26	45.0	24.2	16.3	13.5	43.8	24.0	43.0
w/o graph attention	17.12	40.1	20.2	12.0	8.9	43.1	22.7	43.1

Table 4: The performance of our model under different sampling settings.

Setting	BLEU	B-4	R-1	R-L
Degree-sampling	20.38	15.1	44.2	43.2
- ClassGAT + KC	18.58	12.3	42.8	41.9
Random-sampling	17.64	11.8	43.8	42.9
- ClassGAT + KC	17.61	11.3	42.8	42.2
Degree-overlap	20.58	15.1	44.3	43.2
Random-overlap	18.05	12.3	44.2	43.1

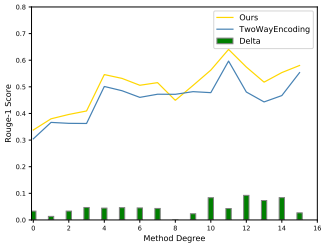

Figure 3: Performance of our model and baselines on functions of different degrees.

and implicit relationships when comprehending functions in a class.

4.2 Human Evaluation

We perform a human evaluation on the test dataset to assess the quality of the generated comments by our framework, Seq2Seq, ClassGAT and ClassGAT with known comments. [1] We randomly sample twenty cases and ask 3 raters to give scores in three aspects. As we can see in Table 2, our model outperforms other strong baselines by a large margin, especially on relevance and informativeness. Our model can effectively utilize rich information in the source file and generate comments that are more relevant with the functions and give more details.

4.3 Ablation Study

To examine the effect of each component in our framework, we evaluate the effect of removing the individual encoder, graph encoder, the by-copy mechanism, and the graph attention mechanism, as shown in Table 3. By and large, all of the components in our model contribute to the model's overall performance. We can see that after removing individual encoder and graph encoder, the performance all drops significantly, around 5 in BLEU score and 3 in Rouge score, indicating that both individual and graph encoders play an indispensable role in the final performance. Without graph attention or a by-copy module, performance deteriorates significantly. It is critical to use the attention mechanism to determine which section should receive the most attention and to copy useful words directly from the target function and from known comments.

4.4 Analysis

4.4.1 Does different known comments matter?

To check if our starting point still stands if the programmers choose different functions to comment, we conduct a set of experiments under different sampling settings where different function comments are known. As shown in Table 4, our model outperform the most competitive baseline ClassGAT+KC under both sampling settings. We can see that any comments available in the class, even randomly sampled, are helpful to produce quality comments for other functions, thanks to both the explicit and implicit relationship captured by our model. So, always write comments if you can.

4.4.2 How should we write comments?

Then, our next question is which functions we should first write comments for so that they can benefit the most. In Table 4, we compare the performance on the same (overlapped) test sets under different sampling settings, and clearly, performance under degree sampling is much better than under random sampling. We can infer that writing comments for functions with higher degrees may provide more contextual information, hence can benefit more functions in the class. Figure 3 depicts the trend of Rouge-1 scores over functions of different degrees. We can see that our model outper-

[1] More details of human evaluation are provided in Appendix A

Table 5: Examples of generated comments.

	Example 1	Example 2
Target Function	`public void start() throws Exception { if (serverMode){... server.start();} waitForServer(); started = true; }`	`public MAssociationEndRole createAssociationEndRole() { MAssociationEndRole modelElement = ... super.initialize(modelElement); return modelElement; }`
Known Function & Comment	`/* Stop the Database engine.*/ public void stop() { if (server != null) { ... } started = false; }`	`/* Builds a message within some interaction related to some assocation role. */ public MMessage buildMessage(MInteraction inter, MAssociationRole role) { ... return message; }`
Golden Comments	start the database engine	create an empty but initialized instance of a uml association end role
Seq2Seq	starts the application	returns the first association of the model
TwoWayEncoding	starts the database	create an empty but do not read associated to this interaction
Our Model	starts the database engine	create an empty but initialized instance of a uml association role

forms TwoWayEncoding(do not consider function relations) more in high degree functions than low degree functions.

In all, when engineering in real life, it may be a good idea to first write comments for those with higher degrees.

4.5 Case Study

Now, we present two examples with generated comments by different models in Table 5. As we can see, in the first case, it is hard to know what should be started only with source codes. However, it can be easily inferred from the known comments. As `start` and `stop` is connected by our defined rule, our model successfully captures this relevance and successfully generates keyword *database engine*, while other models generate messy objectives. In the second case, the commented function is less related to the target function `createAssociationEndRole`, and the main information should come from the source code itself. As there is no connection between these two functions under our relationship definition, our model is not affected by this known comment and successfully generate a good comment, while TwoWayEncoding is influenced too much by the known comment and the generated comment deviates a lot from the original intention.

5 Related Work

Code comment generation aims at generating descriptive natural language for source codes.

Early works (Iyer et al., 2016; Allamanis et al., 2016; Hu et al., 2018b) treats source codes as sequential text and employs attentive Seq2Seq methods to generate comments. Later works focus on procedural structure of source codes. (Hu et al., 2018a; Alon et al., 2019; Liang and Zhu, 2018; Allamanis et al., 2018; LeClair et al., 2020; Ahmad et al., 2020) applies a variety of models on Abstract Synatx Tree(AST) to extract structural information. (Shi et al., 2020) proposes to focus on more basic information that can truly reflect how programs execute and applies a GGNN(Li et al., 2016) module on the assembly code and dynamic memory states of the program. Some recent works start to take a broader view. Some works exploit retrieval-based methods. Zhang et al. (2020) proposes to utilize similar functions through retrieval method. Liu et al. (2020) makes a further efforts to combine the retrieved function and comment together into the generation process. Some works model functions in a class level. Yu et al. (2020) builds a class-level function graph and extract information from it as global context.

6 Conclusion

We present a novel approach for automatic code comment generation that targets a practical problem where human-written comments are only available for a few methods. We propose to focus on both explicit and implicit relationships between functions together and design a framework to effectively extract useful information from existing comments and functions in the source file. Experimental results show that our approach generates comments that outperforms prior methods in both automatic and human evaluation metrics.

References

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*, pages 4998–5007. Association for Computational Linguistics.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

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*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2091–2100. JMLR.org.

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.

Sushil Krishna Bajracharya, Joel Ossher, and Cristina Videira Lopes. 2014. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Sci. Comput. Program.*, 79:241–259.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734. ACL.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP 2020, Online Event, 16-20 November 2020*, pages 1536–1547. Association for Computational Linguistics.

Beat Fluri, Michael Würsch, and Harald C. Gall. 2007. Do code and comments co-evolve? on the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE 2007), 28-31 October 2007, Vancouver, BC, Canada*, pages 70–79. IEEE Computer Society.

Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved automatic summarization of subroutines via attention to file context. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, pages 300–310. ACM.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pages 200–210. ACM.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018b. Summarizing source code with transferred API knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 2269–2275. ijcai.org.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Thomas N. Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. *CoRR*, abs/2004.02843.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated graph sequence neural networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

Yuding Liang and Kenny Qili Zhu. 2018. Automatic generation of text descriptive comments for code blocks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 5229–5236. AAAI Press.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2020. Automatic code summarization via multi-dimensional semantic fusing in GNN. *CoRR*, abs/2006.05405.

Steve McConnell. 2004. *Code complete*. Pearson Education.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*, pages 311–318. ACL.

Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 1073–1083. Association for Computational Linguistics.

Zhan Shi, Kevin Swersky, Daniel Tarlow, Parthasarathy Ranganathan, and Milad Hashemi. 2020. Learning execution through neural code fusion. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958.

Fei Sun, Peng Jiang, Hanxiao Sun, Changhua Pei, Wenwu Ou, and Xiaobo Wang. 2018. Multi-source pointer network for product title summarization. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM 2018, Torino, Italy, October 22-26, 2018*, pages 7–16. ACM.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112.

Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring program comprehension: a large-scale field study with professionals. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, page 584. ACM.

Xiaohan Yu, Quzhe Huang, Zheng Wang, Yansong Feng, and Dongyan Zhao. 2020. Towards context-aware code comment generation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP 2020, Online Event, 16-20 November 2020*, pages 3938–3947. Association for Computational Linguistics.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1385–1397. ACM.

Barret Zoph and Kevin Knight. 2016. Multi-source neural translation. In *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016*, pages 30–34. The Association for Computational Linguistics.

A Human Evaluation Guidelines

Please assess the quality of comments towards the provided function with regard to the following features: fluency, relevancy and informativeness. Give ratings from 0 to 2. In the following, we give explanation towards each feature.

- Fluency: This metric evaluates whether the generated comment is fluent to read. It depends on the grammaticality and word usage of the sentence.
- Relevance: This metric evaluates how much the generated comment is relevant to the corresponding function. It measures whether or not the comment can describe or summarizes the purpose of the function.
- Informativeness: his metric evaluates how much concrete information the comment contains. The more informative comment should give more details instead of using meaningless words that can refer to a lot of things.