Towards Context-Aware Code Comment Generation

Xiaohan Yu, Quzhe Huang, Zheng Wang, Yansong Feng, Dongyan Zhao
Wangxuan Institute of Computer Technology, Peking University, China
School of Computing, University of Leeds, United Kingdom
The MOE Key Laboratory of Computational Linguistics, Peking University, China
{yuxiaohan,huangquzhe,fengyansong,zhaodongyan}@pku.edu.cn
z.wang5@leeds.ac.uk

Abstract

Code comments are vital for software maintenance and comprehension, but many software projects suffer from the lack of meaningful and up-to-date comments in practice. This paper presents a novel approach to automatically generate code comments at a function level by targeting object-oriented programming languages. Unlike prior work that only uses information locally available within the target function, our approach leverages broader contextual information by considering all other functions of the same class. To propagate and integrate information beyond the scope of the target function, we design a novel learning framework based on the bidirectional gated recurrent unit and a graph attention network with a pointer mechanism. We apply our approach to produce code comments for Java methods and compare it against four strong baseline methods. Experimental results show that our approach outperforms most methods by a large margin and achieves a comparable result with the state-of-the-art method.

1 Introduction

High-quality code comments are important for software maintenance. Yet, few software projects adequately document the code (Kajko-Mattsson, 2005). One way to overcome the lack of human-written comments, and guard against mismatch and obsolete comments is to automatically generate them.

Classical approaches for auto-comment generation use hand-crafted templates to produce code descriptions (Sridhara et al., 2010; Cortes-Coy et al., 2014; Dawood et al., 2017), but suffer from poor scalability and high maintenance cost due to the expensive overhead of writing comment templates. More recent work takes a learning-based approach by employing neural network (NN) models developed for natural language processing tasks like machine translation to automatically generate comments (Sutskever et al., 2014; Luong et al., 2015). Compared to hand-written templates, a learning-based approach based on empirical data is more scalable and sustainable.

The key to generating high-quality comments is to utilize as much relevant information as possible from the source code to infer the high-level algorithmic intents. Prior work achieves this by converting a representation of the program, e.g., an Abstract Syntax Tree (AST), into a sequential sequence where a sequential model like LSTM can be applied to translate the token sequence into natural language descriptions (Hu et al., 2018a; Alon et al., 2019; LeClair et al., 2019). Some more recent work has employed the Graph Convolution Network (GCN) to directly operate on a graph representation, e.g., an adjacency matrix, of the AST (LeClair et al., 2020). While promising, all existing learning-based approaches only capture information from the target function (or method) to be commented but fail to capitalize on the abundant information and algorithmic intentions available in a broader context like the definition of the callee functions or other information (like the purposes and semantic meanings of data variables) that can only be gleaned through relevant methods in the same class. Because the programmer's strategic intentions are often encapsulated in a scope greater than a local function, ignoring such contextual information would miss massive opportunities.

As a motivation example, consider the Java code shown in Figure 1. Here, our task is to describe the purpose of method `append` defined at lines 10 and 12. This example is from a real-life open-source project, where a developer-written comment is given. The human-written comment states that this method adds a component to a *panel* object and then moves to *the next data column*. Because neither *panel* nor *the next data column* appears in

```
public class DefaultFormBuiler {
  public DefaultFormBuilder (Composite panel,
        FormLayout layout) {
        this(panel, layout, null);
        this.panel = panel;
  }
  ...
  /* Human-written comment:
     Adds a component to the panel using the
          default constraints and proceeds to
           the next data column
  */
  public void append(Control component) {
    append(component, 1);
  }

  public void append(Control component, int
       columnSpan){
    ...
    setColumnSpan(columnSpan);
    add(component);
    setColumnSpan(1);
    nextColumn(columnSpan + 1);
  }
  ...
}
```

Figure 1: An example to illustrate that functions in the same class can help generate meaningful comments.

the target function, existing approaches operate on the AST of this local method would fail to generate meaningful descriptions. Simply inlining the callee function, `append(Control, int)`, does not offer the context of the panel object, which is also important for understanding the programmer intentions. If we could look at a broader context outside the target function, i.e., by leveraging the construction function, `DefaultFormBuilder`, and the callee function, we can then obtain much of the contextual information needed for generating a good quality comment text.

The above example demonstrates the importance of leveraging broader contextual information for comment generation. In object-oriented programming, object classes are the building block for expressing algorithmic intentions. Indeed, it is the class (but not a single local method) that forms the mental boundary of functionality. Since a class encapsulates much of the calling relationship and semantic information that cannot be obtained from a local function, the global structural information in a class should not be ignored when we attempt to understand the purpose of a function.

This paper thus presents a new code comment generation approach by leveraging the global structural information in object-oriented programming languages. Doing so allows us to utilize much of the contextual information within a class to enhance function-level comment generation. As a departure from all prior methods that only consider local information during encoding, our approach employs a two-way encoding mechanism by considering both information within and outside the target function. We achieve this by simultaneously modeling the token sequence of the target function and a program graph that connects all methods of the same class to the target function. We then learn the approximate synergy between the information available within the local function and the wider class scope. A key challenge here is how to determine the importance and relevance of information available at code scopes. To this end, we design a novel decoder for the comment generation process by learning what information at both the local and class level should be emphasized. Our encoder is composed of a local encoder for extracting information from the local function and a global encoder for extracting information at the class level. The decoder then decides which code segmentations within the class should be paid most attention to so that we can employ a pointer mechanism to copy words directly from the source code to generate comments.

We apply our approach to generate function-level comments for Java programs. We evaluate our approach on Java methods collected from over 1,600 open-source repositories hosted on GitHub, and compare it against four strong baseline methods of code comment generation. Experimental results show that our approach consistently delivers higher-quality comments, improving BLUE and Rouge by at least 7.7% and 5.1% (up to 87.1% and 68.3%), respectively.

This paper makes the following contributions:

- It presents the first approach to exploit relevant methods of the same class to enhance the understanding of the target function.
- It is first to show that other functions or methods outside the call graph of the target function can also have a positive contribution to the generated comment.
- It proposes a novel learning framework that can leverage both local and class-level contextual information for code comment generation.

2 Our Approach

As depicted in Figure 2, our code comment generation framework consists of three innovative components. The local encoder, based on a bi-directional

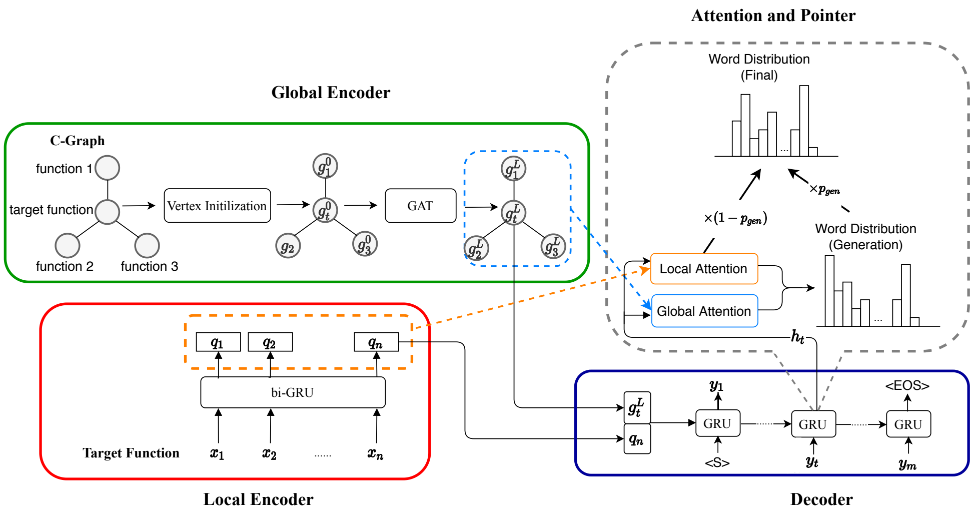

Figure 2: The overall architecture of our approach.

Gated Recurrent Unit (bi-GRU) (Cho et al., 2014), extracts features from the source code of the target function. The global encoder, built upon a Graph Attention (GAT) network ((Velickovic et al., 2018), propagates and exchanges information between all functions with the target class. The decoder aggregates the local and global information learned by the local and global encoders. Our decoder employs an attention mechanism to determine which part of the local and global contexts we should pay attention to and then uses a pointer mechanism to copy words from the source code to generate comments.

2.1 Local Encoder

Our local encoder extracts features from the source code token sequence of the target function. Given the source code of a function $\boldsymbol{x} = (x_1, ..., x_n)$ of n words, we use a bi-GRU to encode it to a dense representation sequence $\{(\overrightarrow{\boldsymbol{q_1}}, \overleftarrow{\boldsymbol{q_1}}), ..., (\overrightarrow{\boldsymbol{q_n}}, \overleftarrow{\boldsymbol{q_n}})\}$, where $\overrightarrow{\boldsymbol{q_j}}$ and $\overleftarrow{\boldsymbol{q_j}}$ are the hidden state of x_j in both directions. We concatenate the last hidden states of both directions to be used as the local representation of the function:

$$\boldsymbol{q_n} = [\overrightarrow{\boldsymbol{q_n}} || \overleftarrow{\boldsymbol{q_n}}] \tag{1}$$

2.2 Global Encoder

Graph Construction. Unlike prior work that only focuses on extracting information from the target function, we aim to exploit the information available at the class level. To do so, we connect the target function to all other functions in the same class to form a class-level contextual graph, for which we refer to as a C-Graph. We then use a GAT network to exchange information between all the nodes in C-Graph to construct a global, class-level representation for the target function. Given a C-Graph $G = (V, E)$, each vertex $v_i \in V$ represents a function in the class, v_t represents the target function, and each edge $e_{t,j} = (v_t, v_j) \in E$ indicates the connection between the target function v_t and other functions v_j.

Vertex Initialization. To encode each vertex in the graph to a hidden vector, we apply the local encoder to each individual function (i.e., a vertex in our C-graph) and concatenate the last hidden states in both directions as the initial vertex representation $\{\boldsymbol{g_i^0} | v_i \in V_f\}$ for GAT.

Graph Attention Network. We feed the initial state of each vertex into a GAT, which applies different weights to different nodes when exchanging information with neighbors. In the $l + 1^{th}$ layer of the GAT network, each vertex v_i sends its own current hidden state to all neighbors represented as $N(v_i)$. At the same layer, each vertex v_i also receives a set of messages $\{\boldsymbol{g_j^l} | v_j \in N(v_i)\}$, where $\boldsymbol{g_j^l}$ is the hidden state of vertex v_j in the l^{th} layer. Then, each vertex calculates a linear combination of the neighborhood hidden states from the message as its new hidden state:

$$\boldsymbol{g_i^{l+1}} = \Sigma_{j \in N(v_i)} \alpha_{ij} \boldsymbol{W_g} \boldsymbol{g_j^l} \tag{2}$$

where $\boldsymbol{W_g}$ is a learnable matrix and α_{ij} is the graph attention distribution between vertex v_i and its neighbors v_j. α_{ij} is computed as:

$$e_{ij} = \sigma(\boldsymbol{a}^T [\boldsymbol{W_a} \boldsymbol{g_i^l} || \boldsymbol{W_a} \boldsymbol{g_j^l}]) \tag{3}$$

$$\alpha_{ij} = softmax(e_{ij}) = \frac{exp(e_{ij})}{\Sigma_k exp(e_{ik})} \quad (4)$$

where $\boldsymbol{W_a}$ and $\boldsymbol{a}$ are learnable parameters, σ stands for an activation function, usually LeakyReLU and $[\cdot||\cdot]$ means the concatenation of two matrices.

We repeat this neighborhood aggregation process L times, and get the final state for each vertex of the C-Graph, represented as $\{\boldsymbol{g_i^L}|v_i \in V_f\}$. We use the final state of the target function $\boldsymbol{g_t^L}$ as the global representation.

2.3 Decoder

As our encoders embed both local and global information, the decoder needs to integrate information extracted at different levels and takes the integrated information into consideration during the generation of comment tokens. Once again, we adopt a GRU as decoder and use the concatenation of local representation $\boldsymbol{q_n}$ and global representation $\boldsymbol{g_t^L}$ as its initial state. $\boldsymbol{h_i}$ is the hidden state of decoder in step i. When generating the i^{th} words, we leverage a graph attention mechanism to extract the most relevant methods of the class and use a pointer mechanism to copy words from the method body to form the comment text.

Graph Attention. In the i^{th} decoding step , we leverage results of the global encoder and compute a global context vector $\boldsymbol{cg_i}$ reflecting which parts of the graph structure should be paid attention to.

$$\boldsymbol{cg_i} = \Sigma_{v_j \in G} \gamma_{ij} \boldsymbol{g_j^L} \quad (5)$$

$$\gamma_{ij} = \frac{exp(\boldsymbol{h_i^T W_{ga} g_j^L})}{\Sigma_{v_k \in G} exp(\boldsymbol{h_i^T W_{ga} g_k^L})} \quad (6)$$

where $\boldsymbol{g_j^L}$ and $\boldsymbol{g_k^L}$ are the representations of functions in the C-Graph and W_{ga} is a trainable matrix.

Local Attention. We apply another attention mechanism to locate most relevant words in the body of target function when generating the i^{th} word of comment, which is represented as a local context vector $\boldsymbol{c_i}$:

$$\boldsymbol{c_i} = \Sigma_{x_j \in x} \beta_{ij} \boldsymbol{q_j} \quad (7)$$

$$\beta_{ij} = \frac{exp(\boldsymbol{h_i^T W_{la} q_j})}{\Sigma_{x_j \in x} exp(\boldsymbol{h_i^T W_{la} q_k})} \quad (8)$$

where $\boldsymbol{q_j}$ and $\boldsymbol{q_k}$ are the contextual embeddings of words x_i and x_j in the input sequence of the target function and $\boldsymbol{W_{la}}$ is a trainable parameter.

Pointer. As source codes may contain information which can be directly used in comment, we propose to add a pointer mechanism (See et al., 2017) which can copy useful words from source codes. Pointer mechanism merges a copy distribution with a normal output prediction distribution. In the i^{th} decoding step,

$$P_{vocab} = softmax(\boldsymbol{W_v}[\boldsymbol{h_i}||\boldsymbol{c_i}||\boldsymbol{cg_i}] + \boldsymbol{b_v}) \quad (9)$$

$$p_{gen} = \sigma(\boldsymbol{w_h^T h_i} + \boldsymbol{w_c^T c_i} + \boldsymbol{w_y^T y_i}) \quad (10)$$

$$P(w) = p_{gen} P_{vocab} + (1 - p_{gen})\boldsymbol{\beta} \quad (11)$$

where $\boldsymbol{\beta}$ is the local attention distribution in Eq 8 and $\boldsymbol{W_v}, \boldsymbol{b_v}, \boldsymbol{w_h}, \boldsymbol{w_c}, \boldsymbol{w_y}$ 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 directly copying from the source codes.

3 Experimental Setup

3.1 Dataset

We apply our approach to Java programs. Our dataset[1] downloaded from 1,634 open-source repositories hosted on Github. These projects are top-ranked Github projects with Java as the primary programming language. Our dataset consists of 150239 target methods within 27323 classes, where each target method has a developer-written comment as the golden comment.

We split the dataset by projects, and use 90% of the project data for training, 5% for validation and 5% for testing. This gives us 133,058 method-comment pairs for training, 6,952 pairs for validation, and 10,229 pairs for testing. The average number of tokens in target functions and target comments are 26.27 and 8.25 respectively. The average number of nodes of the C-Graph is 37.

3.2 Evaluation Metrics

We evaluate our approach by using BLEU, BLEU-1, BLEU-2, BLEU-3, BLEU-4 (Papineni et al., 2002) and ROUGE-L (Lin, 2004). These metrics are widely used in natural language generation.

[1] Available to be downloaded at: *[url redacted for double-blind review].*

3.3 Preprocessing

At the preprocessing stage, we serialize source code of the target method as a sequential token sequence and remove any none-alphabetical letters. We also split identifier and function names written in camelCase or underscore style into independent words. Due to the memory limitation of applying GRU to long sequences, we truncate the source sequence of a function to 100 tokens.

3.4 Model Structures and Hyper-parameters

The dimension of embedding and hidden vectors are set to 128, and word embeddings are randomly initialized. The layer of our GRU encoder is set to 1. The GAT network has four layers since we find using more layers does not improve the results. Each of the GAT layers has a residual connection to avoid gradient vanishment except for the last layer. We set the dropout (Srivastava et al., 2014) rate to 0.1, the weight decay rate to 0.0001 and the batch size to 20. We use Adam optimizer (Kingma and Ba, 2015) with a learning rate of 0.001. To reduce randomness, we run each model setting five times and take the average as the report performance.

3.5 Baseline Models

We compare our model with three types of models, described as follows.

The first kind of models, **CodeNN** (Iyer et al., 2016) and **Seq2Seq** (Luong et al., 2015), treat source code as a sequential sequence of words. CodeNN is a modified language model with attention mechanism, and it is among the earliest NN models for code comment generation. Seq2Seq is the most widely used model for many generation tasks. For these models, we use a bidirectional GRU as the encoder and a GRU with attention as the decoder.

The second kind of models incorporate information from the AST into the model and exploit the structural information of the AST to assist comment generation. We choose two state-of-the-art AST-based generation models: **DeepCom** (Hu et al., 2018a) and **Code2Seq** (Alon et al., 2019). DeepCom works by flattening the whole AST and the applying a Seq2Seq model with attention mechanism to the flatten sequence. It has a specially designed method called structure-based traversal (SBT) that guarantees the relationship between functions and flatten sequences is injective. Code2Seq exploits the AST structural information lies in the paths between leaf nodes. It randomly samples k paths and encodes them together as the representation of AST and apply a simple decoder with attention to generate comments.

Besides, we propose two kinds of model that also leverage class-level information but in a simpler way. The first model, **GraphAttn**, utilizes the C-Graph with an attention mechanism. It applies a bi-GRU on all functions in C-Graph as the graph encoding results and during decoding stage, it attends to each function with a graph attention module. The second model, **GraphFlatten**, flattens the C-Graph into sequences. It concatenates all functions in the C-Graph together, applies a bi-GRU and takes concatenation of last hidden states as the global representation for target function.

4 Experimental Results

4.1 Main Result

We summarize the main results in Table 1. As we can see, most models manage to outperform Seq2Seq in all alternative metrics and introducing AST structural information or global information can both bring improvements. We can draw that structural information is essential if we want to better comprehend source codes and generate more accurate comments.

AST related models show quite different performance results. DeepCom has a BLEU score of less than 15 and this result consists with LeClair et al. (2019) who also performs experiments on Java projects and indicates that splitting functions by project and length growth of input sequence for GRU both have a bad influence on DeepCom. In contrast to DeepCom, Code2Seq achieves the best results in all baseline models, which means it effectively extracts the local structural information provided by AST. Code2Seq generally generates comments that are shorter than ours. It achieves a higher score at n-gram accuracy but suffers from a larger penalty, so our model outperforms it by Rouge-L and BLEU. We can draw that AST provides local information that can assist comment generation and we assume combining AST structure with our methods together can bring further improvement, as it can benefit both AST structural information and global context information of related methods.

After introducing class-level information, both GraphAttn and GraphFlatten show improvements compared to Seq2Seq, indicating that related functions in the same class are beneficial to the target

Models	BLEU	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Rouge-L
CodeNN	12.71	27.67	14.72	9.92	6.80	26.08
Seq2Seq	18.74	39.13	21.17	13.97	10.66	39.36
DeepCom	14.69	31.80	16.90	12.50	10.30	27.50
Code2Seq	**22.07**	**44.79**	**26.48**	**19.32**	**15.77**	**41.77**
GraphAttn	19.38	40.19	22.12	14.67	10.85	39.63
GraphFlatten	20.24	41.77	23.40	15.14	11.46	41.83
Ours	**23.78**	**43.49**	**26.03**	**18.86**	**14.98**	**43.90**
Call	21.06	40.14	23.30	16.35	12.86	40.70
Random	19.13	39.25	21.51	14.36	11.06	39.59
Sample	18.86	39.19	21.29	14.12	10.73	39.36

Table 1: Model performance compared with baselines.

function in comment generation. Despite the inadequate utilization, global information still shows effectiveness in both models.

Our model manages to outperform most baseline models which indicates that global information has an obviously positive effect towards comment generation. When comparing our model with GraphAttn and GraphFlatten, we can find that building C-Graph is a better choice than utilizing attention mechanism or simply flattening the whole class to extract global information.

4.2 Ablation Study

We perform an ablation study to evaluate the effect of each component of our model. As shown in table 2, we can see that all components of the model contribute to the final results.

Among all the results, removing local encoder has the worst performance, dropping 8 in BLEU score, indicating the importance of local encoder. Though introducing graph structure into the model brings much improvement, the target function alone still manages to provide indispensable information towards the comment generation process.

Comparing our model with ours-pointer (our model without pointer module), we can see that adding pointer mechanism brings about 0.3 improvement. However, comparing Seq2Seq with PointerNet, which is the same architecture as ours-global encoder (our model without global encoder module), we find a 0.6 improvement. Thus, we can see that introducing global information into model enhances the ability to copy words directly from the source codes.

After removing attention module, both performance drops, indicating the benefit of attention mechanism in our decoder. In each step of word generation, it is essential to attend to different functions and words. The attention module not only helps decoder determine the most related part to focus on, but also enhances the local and global information during the decoding process.

4.3 Analysis

Q1: What kind of information is useful?

When the scale of a class grows larger, the functions in the same class start to become various. Given a function, not all of its neighbors are guaranteed to be closely related to it when the class is large. Therefore, it leads to the question of what kind of information is useful in C-Graph. To explore this question, we run a set of experiments and find that inside a class, function calls are able to provide the most valuable information while other functions can provide useful but messier information.

We divide all functions connected to the target function into two categories, function calls and other functions. Experiment **Call** uses a graph only containing edges between target function and function calls. Experiment **Random** uses a graph which has same structure as **Call** but replaces all nodes with functions randomly selected from other functions in class. Experiment **Sample** also has the same architecture as Call but replaces all nodes with functions randomly selected from other classes.

As shown in Table 1, Call outperforms most baseline models by a large margin and exceeds Random by 2 points in BLEU score, indicating that function calls are particularly effective towards comment generation process and they serve the key role in forming the global information. Although Random is outperformed by Call, it still achieves some improvement, indicating that other functions except function calls also have a positive effect towards comment generation. But their enhancement is relatively weaker compared to function

Models	**BLEU**	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**	**Rouge-L**
Ours	23.78	43.49	26.03	18.86	14.98	43.88
- local encoder	15.15	34.43	17.39	10.86	8.09	34.26
- global encoder	19.30	39.61	21.45	14.62	11.18	39.98
- graph attention	20.47	39.92	22.43	15.70	12.49	40.07
- local attention & pointer	20.09	38.97	22.02	15.67	12.11	39.68
- pointer	23.46	43.07	25.84	18.61	14.64	43.50

Table 2: Results of ablation study, where '-' means removing this module from our model.

calls. Sample shows nearly no improvement at all, indicating functions from other classes are of no use towards target function as we expect.

Q2: What is the impact of C-Graph on comment generation?

We hypothesize that the global information of the C-Graph can have a positive impact on comment generation. To quantify the impact of C-Graph, we define two metrics at the word level.

The first metric P_o evaluates if C-Graph can help emphasize key information in the target function. We define set S_o as the set of words in target comment that exist in both target function and neighborhood functions and P_o is the percentage of S_o that can be correctly generated by a model.

The second metric P_c evaluates if C-Graph can provide information that does not exist in the target function. We define set S_c as the set of words of target comment that exist in neighborhood functions instead of target function and P_c is the percentage of S_c that can be correctly generated by a model. To be noted, we neglect all the stop words when collecting sets S_o and S_c.

Seq2Seq has a result of $P_o = 51.6\%, P_c = 19.1\%$ while our model has $P_o = 53.1\%, P_c = 25.1\%$, which proves that C-Graph is able to provide extra information as well as help emphasize important information to the comment generation process.

Q3: Does more contextual information bring more improvement?

We apply a statistic experiment on our results and find that with more information brought by C-Graph, there is more improvement. We evaluate the amount of information that C-Graph provides by word overlaps.

3 depicts the BLEU score over the percentage of functions that has word overlap with target comment on C-Graph on the left. As we can see, with more functions overlap with target comment, we achieve better BLEU scores, except the last point. Figure 3 depicts the BLEU score over word over-

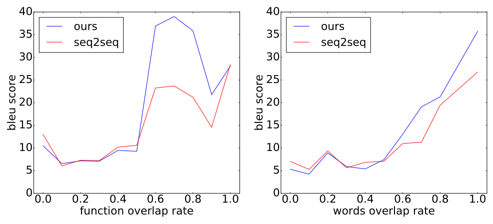

Figure 3: The left sub-figure shows BLEU score over the percentage of functions that has word overlap with target comment on C-Graph. The right sub-figure illustrates BLEU score over word overlap rate between target comment and comments of its related functions on C-Graph.

lap rate between target comment and comments of neighbor functions on C-Graph on the right. Along with the increase of this word overlap rate, the metric improvement start to increase, presenting a horn-shape. Figure

Using word overlap as a simple measure of the information amount given by C-Graph, we found that the more information is provided, the more improvement can be observed.

4.4 Case Study

Table **??** gives four example comments generated by our approach and Seq2seq, as well as the golden comment written by the developer. As we can see, the comments given by our model are more closer to the golden comments and is more meaningful than the one produced by Seq2seq.

In the first case, function `y2` does not give much information and we can not understand its meaning by looking at it alone. Therefore, the comment generated by Seq2Seq is meaningless and unreadable. However, when taking other functions in the same class into consideration, we can find that function `setBox` uses `y2` as well as other similar variables including `x1, x2, y1`, and they are obtained from a object named `location`. Based on these extra information, we can infer that `y2` is a coordinate of a point. As we can see, comment generated by our model successfully capture the

	Example 1	Example 2
Target function	`public double y2() { return y2; }`	`private String elementString(String name, Object value) { return name + "=" + toStringHelper(value); }`
Related function	`public void setBox (BoundingBox location) { x1 = location.x1(); y1 = location.y1(); x2 = location.x2(); y2 = location.y2(); }`	`protected String toStringHelper(Map<String, SoyData> map) { StringBuilder mapStr = new StringBuilder(); ... mapStr.append(entry.getKey()).append(":␣").append(entry.getValue().toString()); ... return mapStr.toString(); }`
Golden comment	getter function for the y coordinate of the second point on the box	string representation of the value pair of the form
Seq2Seq	returns the value of the pay pal field code x code	this is used to acquire the element from the element
Our model	returns the y coordinate of the point	this method is used to convert the value into the string
	Example 3	**Example 4**
Target function	`public void insert(ForceItem item) { try{ insert(item, root, xMin, yMin, xMax, yMax); } catch(StackOverflowError e) { e.printStackTrace(); } }`	`public String PALO_EFIRST(String servdb, String dimensionName) { return PALO_ENAME(servdb, dimensionName, new Double(1), null, null); }`
Related function	`private void insert(ForceItem p, QuadTreeNode n, float x1, float y1, float x2, float y2) { if (n.hasChildren) { insertHelper(p,n,x1,y1,x2,y2); } else if (n.value != null) { if (isSameLocation(n.value, p)) { insertHelper(p,n,x1,y1,x2,y2); } else { ForceItem v = n.value; n.value = null; insertHelper(v,n,x1,y1,x2,y2); insertHelper(p,n,x1,y1,x2,y2); } } else { n.value = p; } }`	`public String PALO_ENAME(String servdb, String dimensionName, Object aindex, Object afig, Object apath, Object alias) { try { if (aindex instanceof Double) { IDimension dimension = manager.getDimension(servdb, dimensionName); ... return getAliasElementName(dimension, element.getName(), alias); } else if (aindex instanceof String) { return PALO_ESIBLING(servdb, dimensionName, (String)aindex, 0, alias); } return "#INVALID␣PARAMETERS"; } catch(Exception e) { return null; } }`
Golden comment	inserts an item into the quadtree	retrieves the name of the first element within a dimension
Seq2Seq	insert the methods description here	returns the value of the specified string
Our model	inserts an item into the tree	returns the value of the specified name from the given dimension

Table 3: Example comments produced by our approach and Seq2Seq.

correct meaning of `y2`.

In the second case, the function `elementString` mainly calls a function `toStringHelper` in its return statement. When we refer to function `toStringHelper`, we know that it is to convert a map object to a string representation. With this extra information, we can understand the goal of the target function. As we can see, the name of the function `elementString` is not a good reflection of this function. Comment generated by Seq2Seq is misled to a completely irrelevant meaning while our model successfully captures the important position of function call and manage to generate correct description for target function.

In the third case, the target function `insert(ForceItem item)` mainly calls another function `insert(ForceItem p, QuadTreeNode n...)` to accomplish its goal. The target function itself is too short to give enough information to generate a meaningful and useful comment. We can only tell that the target function is to insert something but we do not know more details. However, when referring to its callee function, we can know that it is to insert an item into a `quadtree` With this extra information, we can understand the goal of the target function. As we can see, comment generated by our model successfully captures that the it is to insert an item into the tree structure.

In the fourth case, comment generated by Seq2Seq model only uses ambiguous words such as ”value” and ”special string” which do not offer any useful information. However, with related function involved, our model manages to capture the key information ”dimension” and ”name”.

5 Related Work

As a critical task in software engineering, code comment generation has been exploited with various solutions. In recent years, deep learning based methods has dominated this line of research, most of which follow an encoder-decoder framework and can produce readable comments. Iyer et al. (2016) proposes an LSTM based language model with attention mechanism to generate short description for C# and SQL queries. Hu et al. (2018b) exploits the transferred knowledge from automatic API summaries to enhance the generation of code comments. However, this kind of methods fail to utilize the structural characteristics of programming codes. More recent efforts explicitly adopt the AST structure to explore the structure of code. Hu et al. (2018a) introduces AST sequences generated by Structure-Based Traversal (SBT) as a structured summary of the code into the generation model. Liang and Zhu (2018) applies an RNN unit over AST trees to extract both semantic and syntactic information and design a special decoder Code-GRU for the generation process. Alon et al. (2019) decomposes AST trees into paths between leaf nodes and sample a certain amount from them as the structural input for the model. LeClair et al. (2019) proposes to use both AST sequences and source code sequence as multiple input for model. LeClair et al. (2020) proposes to employ GNN over AST structures to better extract structure information. Our work, on the contrast, explores broader context, class-level neighboring functions, to introduce rich information for comment generation.

6 Conclusion

We have presented a novel approach for automatic code comment generation, targeting object-oriented programming languages. Unlike prior work that only leverages information of the target function, our approach leverages related methods of the same class to exploit the information available in a broader context to improve the quality of the generated comment. Our novel learning framework extracts local information from the target function and global contextual information at the class level. Experiment results show that our model can efficiently combine both local and class-level information and generate more detailed and higher-quality comments over prior methods.

Acknowledgments

This work is supported in part by the National Hi-Tech RD Program of China (No. 2018YFB1005100), the NSFC under grant agreements 61672057, 61672058, 61872294 and 61872294. For any correspondence, please contact Yansong Feng.

References

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.

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.

Luis Fernando Cortes-Coy, Mario Linares Vásquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On automatically generating commit messages via summarization of source code changes. In *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*, pages 275–284. IEEE Computer Society.

K. A. Dawood, K. Y. Sharif, and K. T. Wei. 2017. Source code analysis extractive approach to generate textual summary. *Journal of Theoretical and Applied Information Technology*, 95(21):5765–5777.

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.

Mira Kajko-Mattsson. 2005. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering*, 10(1):31–55.

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*.

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

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 795–806. IEEE / ACM.

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.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pages 1412–1421. The Association for Computational Linguistics.

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.

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 43–52. ACM.

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.

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.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.