CoCoGUM: Contextual Code Summarization with Multi-Relational GNN on UMLs

Yanlin Wang
yanlwang@microsoft.com
Microsoft Research Asia

Lun Du
lun.du@microsoft.com
Microsoft Research Asia

Ensheng Shi*
s1530129650@stu.xjtu.edu.cn
Xi'an Jiaotong University

Yuxuan Hu†
huyuxuan@emails.bjut.edu.cn
Beijing University of Technology

Shi Han
shihan@microsoft.com
Microsoft Research Asia

Dongmei Zhang
dongmeiz@microsoft.com
Microsoft Research Asia

ABSTRACT

Code summaries are short natural language (NL) descriptions of code snippets that help developers better understand and maintain source code. Due to the pivotal role of code summaries in software development and maintenance, there is a surge of works on automatic code summarization to reduce the heavy burdens of developers. However, contemporary approaches only leverage the information within the boundary of the method being summarized (i.e., local context), and ignore that using broader context could assist with code summarization. In this paper, we explore two global context information, namely intra-class and inter-class context information, and propose the model **CoCoGUM**: **Co**ntextual **Co**de Summarization with Multi-Relational **G**raph Neural Networks on **UM**Ls. CoCoGUM first incorporates class names as the intra-class context, which is further fed to a Transformer-based sentence embedding model to extract the *class lexical embeddings*. Then, relevant Unified Modeling Language (UML) class diagrams are extracted as inter-class context and we use a Multi-Relational Graph Neural Network (MR-GNN) to encode the *class relational embeddings*. Class lexical embeddings and class relational embeddings, together with the outputs from code token encoder and AST encoder, are passed to the decoder armed with a two-level attention mechanism to generate high-quality context-aware code summaries. We conduct extensive experiments to evaluate our approach and compare it with other automatic code summarization models. The experimental results show that CoCoGUM outperforms state-of-the-art methods.

KEYWORDS

Source Code Summarization, Unified Modeling Language, Graph Neural Network, Transformer

1 INTRODUCTION

Code summaries (i.e., code comments) are short natural language descriptions of source code and an essential part of software development and maintenance [1, 2]. It often takes developers plenty of time to read and understand other people's code. Good comments can help developers quickly understand what a method does and achieve higher degrees of program comprehension in terms of both clarity and depth. But many developers are not used to writing comments or the comments are mismatched, missing or outdated. Therefore, it is desirable that source code can be automatically summarized in natural language.

There is a surge of work on source code summarization in order to enhance program comprehension and software maintenance [3]. Rule-based approaches rely on predefined rules or templates [4] and thus are limited in the types of summaries that can be generated. Information Retrieval (IR) based approaches use similar code snippets to select or synthesize a comment [5], and thus lack the capability of learning semantic meanings from existing code. With the accumulation of publicly available source code, data-driven approaches based on deep learning techniques have largely overtaken traditional methods [3]. Inspired by the success of Neural Machine Translation (NMT) [6], some researches have applied NMT models for the code summarization task [7–10].

We have noticed that most previous code summarization techniques [7–15] only leverage the information within the boundary of the method being summarized (code sequence, AST, etc), which means only *local* context has been used and this could be a limitation. From the perspective of Object Oriented Programming (OOP), a method typically defines an operation that can be conducted by its enclosing class, applying to the class itself or against other classes. The operation initiator/receptor classes (e.g., keywords in class name) and their interactions may be mentioned in a good summary, which is typically outside of the local context of a method. For example, the word *resource* in the following method summary comes from its enclosing class name `Resource`:

```
class Resource {
    ...
    // check whether a resource is occupied
    boolean isOccupied() { return ownerId >= 0; }
}
```

Take the following code snippet as another example, the word *compiler* in the method summary comes from class `Compiler` which the enclosing class `Parser` *extends*:

```
class Parser extends Compiler {
    ...
    // add source code to the compiler
    void addSource(String className, String
         sourceCode) {
        sourceCodes.put(className, new SourceCode(
             className, sourceCode)); }
}
```

Based on such an insight, we argue that leveraging broader context of a method would help better summarize the method.

*The contributions by Ensheng Shi have been conducted and completed during his joint PhD program with Microsoft Research Asia.

†The contributions by Yuxuan Hu have been conducted and completed during his internship at Microsoft Research Asia.

In this paper, we propose **CoCoGUM**: **Co**ntextual **Co**de Summarization with Multi-Relational **G**raph Neural Networks on **UM**Ls. It addresses the aforementioned limitations by exploring two global contexts: intra-class level context and inter-class level context. For the intra-class context, we aim to capture the ⟨method, class⟩ relationship by using class name information to enrich generated code summaries. For the inter-class context, we choose the ⟨class, class⟩ relationship as the target where the Unified Modeling Language (UML) class diagram* is a proper choice. Nodes in a UML represent classes/interfaces and edges reflect the class-class relationship such as extension, association, implementation, etc.

In our model design of CoCoGUM, in addition to the conventional code token encoder and AST encoder as in other approaches [8–10, 15], we incorporate intra-class context by using the class name representations learned from the Transformer-based sentence embedding model [16]. This is used as an additional encoder to the prevalent 2-encoder architecture (i.e., code token encoder and AST encoder). Moreover, we extend the idea of graph neural network GAT [17] and design a Multi-Relational Graph Neural Network (MR-GNN) that uses the attention mechanism to learn the UML graphs of source code to encode the inter-class context.

We summarize our main contributions as follows:

- To our best knowledge, we are the first to consider global contexts (i.e., class name and UML) in the automatic code summarization task.
- We propose the Multi-Relational Graph Neural Network (MR-GNN) to model the UML diagrams of source code so that inter-class context can be captured.
- We design an end-to-end automatic code summarization framework CoCoGUM with two novel components, i.e., a Transformer-based sentence embedding model for encoding class names and a MR-GNN for modeling UML graphs, which work together with the components of code token encoder and AST encoder from any existing Seq2Seq-based neural code summarization model.
- We conduct extensive experiments which demonstrate that CoCoGUM outperforms state-of-the-art code summarization methods, since incorporating intra-class and inter-class contexts and combining them with the attention mechanism improves the quality of generated code summaries.

The remainder of this paper is organized as follows: Sec. 2 presents the background. Sec. 3 introduces the design of CoCoGUM. We present details about our experiment settings in Sec. 4. We compare CoCoGUM with state-of-the-art works in Sec. 5. Sec. 6 illustrates the related work. Finally, we conclude our work in Sec. 7.

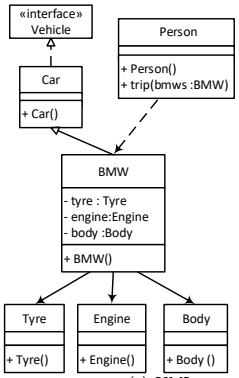
2 BACKGROUND

In this section, we briefly illustrate some background knowledge.

2.1 Unified Modeling Language

The Unified Modeling Language (UML) [18, 19] has being widely accepted as a modeling notation for visualization during the design and development of software systems. UML is also part of many software engineering course curricula of universities worldwide,

*There are several types of UML diagrams. For simplicity, we will use 'UML' or 'UML graph/diagram' to refer to 'UML class diagram' in this paper.



(a) UML

```
interface Vehicle {}
class Car implements Vehicle
      {}

class Tyre {}
class Engine {}
class Body {}

class BMW extends Car {
    Tyre tyre;
    Engine engine;
    Body body;
}

class Person {
  void trip(BMW bmws) {}
}
```

(b) Source code snippet

Figure 1: UML class diagram example

providing a recognized tool for practical training of students in understanding and visualizing software design.

There are several types of UML diagrams and each of them serves a different purpose. The most common one for software documentation is UML class diagram. In a nutshell, a UML class diagram contains classes, alongside with their attributes and methods (also referred to as member functions). More specifically, each class in the UML class diagram has 3 fields: the class name at the top, the class attributes right below the name, the class methods at the bottom. The relation between different classes (represented by a connecting line) makes up a class diagram.

Fig. 1 shows a basic class diagram example. The class `Car` implements the interface `Vehicle`. The dashed line with a hollow arrow connecting class `Car` and interface `Vehicle` indicates the 'implement' relationship. The class `BMW` inherits from the more general class `Car`. The 'inheritance' relationship is represented by a solid line with an hollow arrow. Further, a `BMW` class is associated with its children classes `Type`, `Engine` and `Body`. The 'association' relationship is represented by a solid line with a solid arrow. The `Person` class should depend on `BMW` class for its method `trip`. The 'dependency' relationship is represented by a dashed line with a solid arrow.

2.2 Seq2Seq

Seq2Seq [20] is a family of machine learning models which aims at transforming a sequence into another sequence. Typical applications are machine translation [6], text summarization [21], image captioning [22], etc.

Seq2Seq approaches are based on an encoder-decoder architecture, with an encoder mapping a source sequence to a latent vector and a decoder translating the latent vector into a target sequence. Recurrent Neural Networks (RNNs) [23], Long Short-term Memory Networks (LSTMs) [24], Gated Recurrent Units (GRUs) [6], Transformers [25] or other deep neural networks [26], which are capable of modeling sequential data, are are widely used for the encoder and the decoder in such a design. Furthermore, there are some techniques that are commonly used in the encoder-decoder architecture, including the attention mechanism [25], teacher forcing [27], etc. Attention mechanism endows the decoder with the

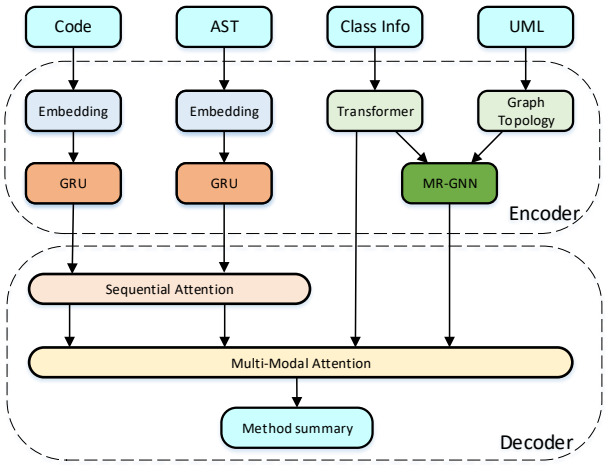


Figure 2: The overview of CoCoGUM.

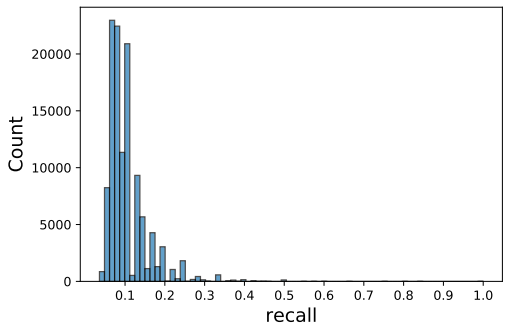


Figure 3: The distribution of class name recall in summaries

ability to process the input sequence selectively. Teacher forcing uses the ground-truth sequences to guide the generation in the training phase.

2.3 Graph Neural Networks

The recent advances of deep learning techniques have facilitated many machine learning tasks like object detection, machine translation, and speech recognition [28]. The key to such a technical evolution is several deep neural networks such as Convolutional Neural Networks (CNNs) [29]. They are able to automatically learn the data features instead of heavily relying on handcrafted feature engineering.

Unlike the data that are originally organized in euclidean space (e.g., image on 2D grid and text in 1D sequence) where CNNs and RNNs can effectively capture hidden patterns, graph data is irregular and each node may have a different number of neighbors. The irregularity of graph data makes some important operations (e.g., convolution), which are easy to compute in euclidean data, difficult for graph data [28]. The attempt to extend deep neural networks to graph data has spawned Graph Neural Networks (GNNs) [28, 30].

The first motivation of GNNs roots in CNNs, since CNNs have a strong ability to learn local stationary structures via localized convolution filter and compose them to form multi-scale hierarchical patterns. Many efforts have been devoted to defining the convolution operation for graph data. This branch of GNNs is therefore called Graph Convolutional Networks (GCNs) and can be divided into spectral methods which define convolution in spectral domain and spatial methods which define convolution in the vertex domain.

The prevalence of graph data in real-world makes GNNs applicable for many learning-based systems. Due to the strong ability of learning from graph data, GNNs have achieve a great success in many tasks like learning chemistry rule and phenomenon [31], recommender systems [32], document classification [33], traffic predication [34], etc.

3 COCOGUM MODEL

In this section, we present our model CoCoGUM. The architecture of CoCoGUM (see Fig. 2) follows the Seq2Seq framework, including two main components, i.e., the encoder module and the decoder module. The encoder module can be divided into two parts, namely the local (method-level) encoder and the global (class-level) encoder. The former is responsible for representing the lexical and syntactic information of each method, and the latter encodes the lexical information of each class and the relationship between classes. The decoder integrates the multi-modality representations from the above encoders through an attention mechanism and generates the final summary.

3.1 Local Context Encoders using GRU

Contemporary code summarization approaches mainly leverage local contexts including code tokens and abstract syntax trees (ASTs) to capture the lexical and syntactic information of a code snippet. For the local context encoders, we follow the Ast-attendgru model of LeClair et al. [10]. We preprocess the code snippets to ASTs and adopt the structure-based traversal (SBT) format [15] to flatten ASTs into sequences. CoCoGUM feeds the code sequence and SBT sequence into a code encoder and an AST encoder, respectively. Both encoders adopt GRU [6], but the parameters are not shared:

$$\begin{aligned}
\mathbf{r}_t &= \sigma(\mathbf{W}_r\mathbf{x}_t + \mathbf{b}_r + \mathbf{W}_{hr}\mathbf{h}_{t-1} + \mathbf{b}_{hr}) \\
\mathbf{z}_t &= \sigma(\mathbf{W}_z\mathbf{x}_t + \mathbf{b}_z + \mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_{hz}) \\
\mathbf{n}_t &= tanh\big(\mathbf{W}_n\mathbf{x}_t + \mathbf{b}_n + \mathbf{r}_t \circ (\mathbf{W}_{hn}\mathbf{h}_{t-1} + \mathbf{b}_{hn})\big) \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \circ \mathbf{n}_t + \mathbf{z}_t \circ \mathbf{h}_{t-1}
\end{aligned} \tag{1}$$

where $\mathbf{x}_t$ indicates the embedding of the t-th token in either code token sequence or SBT sequence, $\mathbf{h}_t$ is the hidden state for the t-th token, and $\mathbf{r}_t$, $\mathbf{z}_t$, $\mathbf{n}_t$ indicate the reset, update and new gates, respectively. σ denotes the sigmoid function and "$\circ$" is the Hadamard product. $\mathbf{W}_r$, $\mathbf{W}_{hr}$, $\mathbf{W}_z$, $\mathbf{W}_{hz}$, $\mathbf{W}_n$ and $\mathbf{W}_{hn}$ are learnable weight matrices. $\mathbf{b}_r$, $\mathbf{b}_{hr}$, $\mathbf{b}_z$, $\mathbf{b}_{hz}$, $\mathbf{b}_n$ and $\mathbf{b}_{hn}$ are bias vectors.

It is worth noting that the detailed design of the local context encoder can be replaced by the encoder from any Seq2Seq-based code summarization methods (e.g., [9, 15, 35]), as the local context encoder is orthogonal to the global context encoder (illustrated in Sec. 3.2) in CoCoGUM.

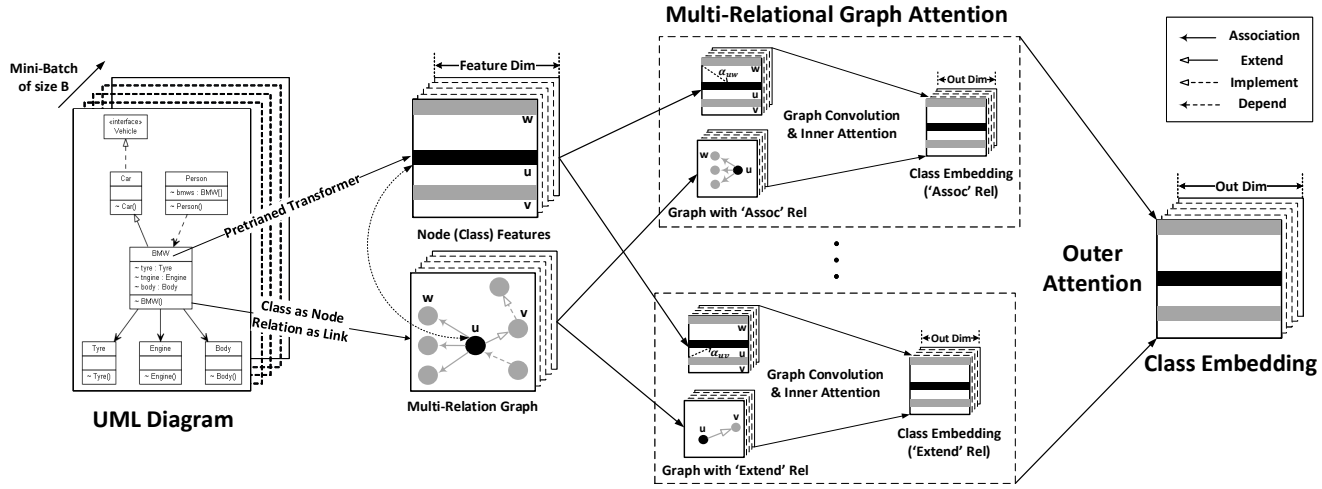


Figure 4: Global context encoder based on our proposed MR-GNN

3.2 Global Context Encoder based on MR-GNN

The global context in CoCoGUM contains the intra-class (i.e., class name) and the inter-class (i.e., relationship between classes) information. In order to intuitively show the importance of class names, we define *class name recall* to be the rate of tokens in the class name that also appear in the method summaries. As shown in Fig. 3, there is a strong correlation between method summaries and the corresponding class names: a large number of summaries directly refer to some of the words in class names. For the class-class relationship, we extract the UML of the package that contains the target method and retain four common relationships: `IMPLEMENTS`, `EXTENDS`, `DEPEND`, and `ASSOCIATION`. As illustrated in Fig. 4, the global context encoder takes two steps to encode global contexts.

Firstly, a class name is preprocessed to a token sequence by the steps described in Sec. 4.1. Then CoCoGUM generates class name embeddings using the Transformer-based sentence embedding model [16] to obtain the *class lexical embedding*. The sentence embedding model is trained and optimized for greater-than-word length text, such as sentences or short paragraphs on a variety of NL data sources and tasks, and the output is a 512-dimensional vector. The class lexical embedding is part of the output of the global context encoder, which will be passed to the subsequent GNN module as initial features of the nodes in the UML graph.

Secondly, we extract structure from UML graphs and use the proposed Multi-Relational Graph Neural Network (MR-GNN) to encode inter-class context. Specifically, we set each class as a node and each relationship between two classes as an edge between corresponding nodes. Since there are four types of class relationships, we get four graphs accordingly with each graph containing one type of the four relationships. Each class node v_i has one general node embedding $\mathbf{h}_i^{(g)}$ and four relation-specific embeddings $\mathbf{h}_i^{(r)}$ with r being one of the four class-class relations. The class lexical embedding $\mathbf{h}_i^{(l)}$, which is generated by the sentence embedding model for class node v_i, will be used as the initial hidden state for $\mathbf{h}_i^{(g)}$. The UML graphs have certain characteristics which are rarely considered in general graphs. On one hand, they have multi-relational edges. And on the other hand, the definition of each type is relatively broad, resulting in a huge difference in the semantics of edges with the same relationship. For instance, `ASSOCIATION` relationship may exist between `<Student, Professor>` and also `<Student, Course>`, but their semantics are obviously different. Some classic GNN models such as GCN [33] cannot handle such scenarios. As a comparison, our proposed MR-GNN is able to capture different relation semantics in UML graphs via inner-attention and outer-attention. The former is mainly used to distinguish the effects of different neighbors in the same relationship, while the latter reflects the influence of different relationships.

To be specific, similar to Graph Attention Network (GAT) [17], inner attention defines an attention layer to capture the different importance of each neighbor in the same relation r:

$$\alpha_{i,j}^{(r)} = \frac{\exp\left(\Gamma^{(r)}\left(\mathbf{W}_a^{(r)}\mathbf{h}_i^{(r)} \oplus \mathbf{W}_a^{(r)}\mathbf{h}_j^{(r)}\right)\right)}{\sum_{k\in\mathcal{N}_i^{(r)}} \exp\left(\Gamma^{(r)}\left(\mathbf{W}_a^{(r)}\mathbf{h}_i^{(r)} \oplus \mathbf{W}_a^{(r)}\mathbf{h}_k^{(r)}\right)\right)}, \tag{2}$$

where $\alpha_{i,j}^{(r)}$ is the inner attention coefficient of node v_j to node v_i, $\mathbf{h}_i^{(r)}$ is the hidden representation of node v_i for relation r, $\mathcal{N}_i^{(r)}$ is the first-order neighbors of node v_i with the relation r, "$\oplus$" is the concatenation operation, $\Gamma^{(r)}(\cdot)$ is a single-layer feedforward neural network with a scalar being the output, and $\mathbf{W}_a^{(r)}$ is a learnable parameter matrix. Given the inner attention coefficient, the hidden representation of node v_i with respect to the relation r is:

$$\mathbf{h}_i^{(r)} = \sigma\left(\sum_{j\in\mathcal{N}_i^{(r)}} \alpha_{i,j}^{(r)}\mathbf{W}_h^{(r)}\mathbf{h}_j^{(r)}\right), \tag{3}$$

where $\mathbf{W}_h^{(r)}$ is the learnable weight matrix and $\sigma(\cdot)$ is LeakyReLU.

Outer attention, defined in the relation level, is to aggregate the hidden representations of different relations for each node. The

outer attention coefficient of relation r to node v_i is defined as:

$$\beta_i^{(r)} = \frac{\exp\big(< \mathbf{W}_b \mathbf{h}_i^{(g)}, \mathbf{W}_c^{(r)} \mathbf{h}_i^{(r)} > \big)}{\sum_{r \in \mathcal{R}} \exp\big(< \mathbf{W}_b \mathbf{h}_i^{(g)}, \mathbf{W}_c^{(r)} \mathbf{h}_i^{(r)} > \big)}, \tag{4}$$

where "$< \cdot \ , \ \cdot >$" indicates inner product, $\mathbf{W}_b$ and $\mathbf{W}_c^{(r)}$ are parameter matrices to ensure space alignment, and $\mathcal{R}$ is the set of the four relationships in UML graphs. Based on the outer attention coefficient, we calculate the updated general node representation as:

$$\mathbf{h}_i^{'(g)} = \sigma\Big(\sum_{r \in \mathcal{R}} \beta_i^{(r)} \mathbf{W}_d^{(r)} \mathbf{h}_\mathbf{i}^{(\mathbf{r})} \Big), \tag{5}$$

where $\mathbf{h}_i^{'(g)}$ is the output representation of node v_i in the current MR-GNN layer and also the input of the next MR-GNN layer when stacking multiple layers, and $\mathbf{W}_d^{(r)}$ is a learnable parameter matrix. Compared to using concatenation and feedforward network in Eq. 2, we simply adopt an inner product in Eq. 4 due to the small number of UML relationship types. It will help avoid introducing too many parameters and thus causing over-fitting when calculating the attention coefficient. In practice, we stack two MR-GNN layers to obtain the *class relational embedding*.

3.3 Attention-based Decoder

Similar to the code token encoder and AST encoder, we deploy GRU as the backbone of the decoder. Additionally, we use a two-level attention mechanism to endow CoCoGUM with the ability to learn from code token sequence, SBT sequence and UML graph selectively.

Let $\mathbf{H}^{(c)} = [\mathbf{h}_1^{(c)}, \cdots, \mathbf{h}_{T_c}^{(c)}]$, $\mathbf{H}^{(a)} = [\mathbf{h}_1^{(a)}, \cdots, \mathbf{h}_{T_a}^{(a)}]$ be the outputs from code token encoder and AST encoder, where T_c and T_a are the length of code token sequence and SBT sequence, respectively. Firstly, the hidden state $\mathbf{h}_t^{(s)}$ of the t-th summary token is obtained similar to Eq. 1. Then the sequential attention [36] is used to incorporate the different coefficients between each token in code token sequence (and SBT sequence) and the t-th summary token, and generate the code context vector $\mathbf{s}_t^{(c)}$ and SBT context vector $\mathbf{s}_t^{(a)}$ for the t-th summary token:

$$\begin{aligned} \mathbf{s}_t^{(c)} &= \sum_{i=1}^{T_c} \gamma_{i,t} \cdot \mathbf{h}_i^{(c)}, \quad \gamma_{i,t} = \frac{\exp\big(< \mathbf{h}_i^{(c)}, \mathbf{h}_t^{(s)} > \big)}{\sum_{k=1}^{T_c} \exp\big(< \mathbf{h}_k^{(c)}, \mathbf{h}_t^{(s)} > \big)} \\ \mathbf{s}_t^{(a)} &= \sum_{i=1}^{T_a} \delta_{i,t} \cdot \mathbf{h}_i^{(a)}, \quad \delta_{i,t} = \frac{\exp\big(< \mathbf{h}_i^{(a)}, \mathbf{h}_t^{(s)} > \big)}{\sum_{k=1}^{T_a} \exp\big(< \mathbf{h}_k^{(a)}, \mathbf{h}_t^{(s)} > \big)}. \end{aligned} \tag{6}$$

We then perform a modality-level attention to integrate code context, SBT context, intra-class level context $\mathbf{h}^{(l)}$ (i.e., class lexical embedding) and inter-class level context $\mathbf{h}^{(g)}$ (i.e., class relational embedding). The integrated encoding vector $\mathbf{q}_t$ for t-th summary token can be obtained by the following formula:

$$\mathbf{q}_t = \sigma\big(\lambda_t^{(c)} \mathbf{V}_c \mathbf{s}_t^{(c)} + \lambda_t^{(a)} \mathbf{V}_a \mathbf{s}_t^{(a)} + \lambda_t^{(l)} \mathbf{V}_l \mathbf{h}^{(l)} + \lambda_t^{(g)} \mathbf{V}_g \mathbf{h}^{(g)}\big), \tag{7}$$

where $\lambda_t = \frac{1}{Z_t}(\lambda_t^{(c)}, \lambda_t^{(a)}, \lambda_t^{(l)}, \lambda_t^{(g)})^{\mathrm{T}}$ is a normalized attention coefficient vector with Z_t being the normalized factor, and $\mathbf{V}$ denotes a learnable parameter matrix. To be specific, each entry of vector λ_t is defined as:

$$\begin{pmatrix} \lambda_t^{(c)} \\ \lambda_t^{(a)} \\ \lambda_t^{(l)} \\ \lambda_t^{(g)} \end{pmatrix} = \exp\begin{pmatrix} < \mathcal{V}_c \mathbf{s}_t^{(c)}, \mathcal{V}_s \mathbf{h}_t^{(s)} > \\ < \mathcal{V}_a \mathbf{s}_t^{(a)}, \mathcal{V}_s \mathbf{h}_t^{(s)} > \\ < \mathcal{V}_l \mathbf{h}_t^{(l)}, \mathcal{V}_s \mathbf{h}_t^{(s)} > \\ < \mathcal{V}_g \mathbf{h}_t^{(g)}, \mathcal{V}_s \mathbf{h}_t^{(s)} > \end{pmatrix}, \tag{8}$$

where $\mathcal{V}_c, \mathcal{V}_s, \mathcal{V}_a, \mathcal{V}_g$ and $\mathcal{V}_l$ are parameter matrices to ensure space alignment.

Given the t-th summary token hidden state $\mathbf{h}_t^{(s)}$ and the corresponding integrated encoding vector $\mathbf{q}_t$, CoCoGUM predicts the probability distribution of the t-th summary token as:

$$\mathbf{y}_t = \mathrm{softmax}\big(f(\mathbf{h}_t^{(s)} \oplus \mathbf{q}_t)\big), \tag{9}$$

where "$\oplus$" is the concatenation operation and $f(\cdot)$ is a single-layer feedforward neural network. The cross entropy loss is utilized to evaluate the gap between the prediction and the ground truth, i.e.

$$l = \frac{1}{T_s} \sum_{t=1}^{T_s} < \hat{\mathbf{y}}_t, \log \mathbf{y}_t >, \tag{10}$$

where $\hat{\mathbf{y}}_t$ is the t-th target token by one hot encoding and T_s is the length of the target summary. Many gradient descent based methods can be used for optimizing CoCoGUM and we adopt adaptive moment estimation (i.e., AdamW) [37].

4 EXPERIMENTS SETUP

In this section, we will explain data preprocessing, experimental settings and evaluation metrics.

4.1 Dataset and Preprocessing

In our experiment, we use the Java corpus from CodeSearchNet [38]. It contains 542,991 Java methods and associating properties across the training, validation and test sets. In order to build the UML class diagrams for the programs, project-level source code is needed. Fortunately, the CodeSearchNet dataset provides the `url` and `repo` information for each function. We download all the projects (publicly available open-source non-fork GitHub repositories) according to the urls provided in the data.

Then, we filter the dataset with a set of constraints:

(1) Code snippets with missing GitHub repositories are removed.
(2) Code snippets that cannot be processed by *UMLGraph*[†] to build UML class diagrams are removed.
(3) Summaries shorter than three words are removed.
(4) Non-English summaries are removed.

Finally, we get 267,047 pairs of source code and summaries. Tab. 1 illustrates the statistics of the filtered corpus. The dataset is partitioned into training, validation, and test sets by project so that methods from the same repository can only exist in one partition. The relative size of train:validation:test is 9:0.4:0.6.

Table 1: Statistics of the CodeSearchNet dataset.

# Github repositories	# Package uml diagram.	# Method
4,116	73,856	267,047

[†]https://www.spinellis.gr/umlgraph/index.html

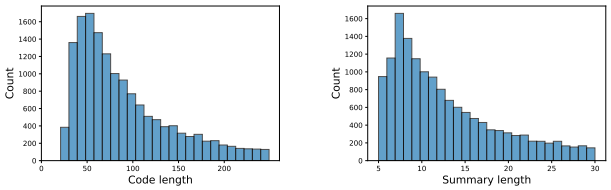


Figure 5: Length distribution of code and summary.

We use *UMLGraph* to extract UML class diagrams at package level. Four class-class relationships are extracted: implementation, extension, dependency and association (which are represented by `IMPLEMENTS`, `EXTENDS`, `DEPEND` and `ASSOC` in UMLGraph, respectively). We also take the following rules to preprocess code and summaries in the dataset:

(1) We split code and summary tokens into subtokens by applying the spiral ronin algorithm[‡] to reduce data sparsity, and then all the tokens are transformed to lower cases.
(2) Punctuations in summaries are removed.
(3) Numerals and string literals are replaced with the generic tokens `<NUM>` and `<STRING>`, respectively.
(4) We use srcml[§] to parse Java methods into ASTs and then we flatten the ASTs to SBT sequences as described in [15].

For class names used in global context, we preprocess a class name by: (1) splitting by camel case or underscores; (2) removing the `<T>`-like tokens in template classes; and (3) removing parent class name, e.g. `Parent.Child` will be processed to `Child`.

Fig. 5 shows the length distribution of code and summaries. We can observe that the length of most code snippets distributes between 50 and 150 and the length of most summaries distributes between 5 and 15.

4.2 Experiment Setting

The maximum lengths of code, summary and SBT are set to 100, 12, and 435 respectively (each covering at least 80% data). We use the special symbol `<NULL>` to pad the short sequences and `<UNK>` to replace the out-of-vocabulary tokens, `<s>` and `</s>` to represent the start and end of the summary sequence, respectively. The vocabulary size is 10,000 (most frequent words in trainning set) for code, summary and SBT sequences.

For UML diagram, we use a 2-layer MR-GNN to encode the class-level information. The class embedding size and hidden dimensions of MR-GNN are 512 and 256, respectively. For baseline Ast-attgru [10], the embedding size and hidden dimension of GRU are 128 and 256 respectively. We set the mini-batch size to 256 and a maximum of 40 epochs for each approach. We use the optimizer AdamW [37] with the learning rate 0.001 for training. To prevent over-fitting, we use Dropout with drop probability 0.5, and Weight Decay with decay ratio 0.3. The experiments are conducted on a server with 2 GPUs of NVIDIA Tesla V100.

[‡]https://github.com/casics/spiral
[§]https://www.srcml.org/

4.3 Evaluation Metrics

Similar to previous work [7, 8, 39], we evaluate the performance of our proposed model based on four widely-used metrics including BLEU [40], METEOR [41], ROUGE-L [42] and CIDER [43]. BLEU, METEOR, ROUGE-L and CIDER are prevalent metrics in machine translation, text summarization and image captioning tasks.

BLEU measures the average n-gram precision between the reference sentences and generated sentences, with brevity penalty for short sentences. The formula to compute BLEU-1/2/3/4 is:

$$\text{BLEU-N} = BP \cdot \exp \sum_{n=1}^{N} \omega_n \log p_n, \tag{11}$$

where p_n (n-gram precision) is the fraction of n-grams in the generated sentences which are present in the reference sentences, and ω_n is the uniform weight 1/N. Since the generated summary is very short, high-order n-grams may not overlap. We use the +1 smoothing function [44]. BP is brevity penalty given as:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \le r \end{cases} \tag{12}$$

Here, c is the length of the generated summary, and r is the length of the reference sentence.

Based on longest common subsequence (LCS), ROUGE-L is widely used in text summarization. Instead of using only recall, it uses F-score which is the harmonic mean of precision and recall values. Suppose A and B are generated and reference summaries of lengths c and r respectively, we have:

$$\begin{cases} P_{ROUGE-L} = \frac{LCS(A,B)}{c} \\ R_{ROUGE-L} = \frac{LCS(A,B)}{r} \end{cases} \tag{13}$$

$F_{ROUGE-L}$, which indicates the value of ROUGE-L, is calculated as the weighted harmonic mean of $P_{ROUGE-L}$ and $R_{ROUGE-L}$:

$$F_{ROUGE-L} = \frac{\left(1+\beta^2\right) P_{ROUGE-L} \cdot R_{ROUGE-L}}{R_{ROUGE-L} + \beta^2 P_{ROUGE-L}} \tag{14}$$

β is set to 1.2 as in [8, 39].

METEOR is a recall-oriented metric that measures how well the model captures the content from the references in the generated sentences and has a better correlation with human judgment. Suppose m is the number of mapped unigrams between the reference and generated sentence with lengths c and r respectively. Then, precision, recall and F are given as:

$$P = \frac{m}{c}, \; R = \frac{m}{r}, \; F = \frac{PR}{\alpha P + (1-\alpha)R} \tag{15}$$

The sequence of mapping unigrams between the two sentences is divided into the fewest possible number of "chunks". This way, the matching unigrams in each "chunk" are adjacent (in two sentences) and the word order is the same. The penalty is then computed as:

$$\text{Pen} = \gamma \cdot \text{ frag}^{\beta} \tag{16}$$

where frag is a fragmentation fraction: frag = ch/m, where ch is the number of matching chunks and m is the total number of matches. The default values of α, β, γ are 0.9, 3.0 and 0.5 respectively.

CIDER is a consensus-based evaluation metric used in image captioning tasks. The notions of importance and accuracy are inherently captured by computing the TF-IDF weight for each n-gram

and using cosine similarity for sentence similarity. To compute CIDER, we first calculate the TF-IDF weighting $g_k(s_i)$ for each n-gram ω_k in reference sentence s_i. Here ω is the vocabulary of all n-grams. Then we use the cosine similarity between the generated sentence and the reference sentences to compute CIDER_n score for n-grams of length n. The formula is given as:

$$\text{CIDER}_n(c_i, s_i) == \frac{< \boldsymbol{g^n}(c_i), \boldsymbol{g^n}(s_i) >}{\|\boldsymbol{g^n}(c_i)\| \, \|\boldsymbol{g^n}(s_i)\|} \tag{17}$$

where $\boldsymbol{g^n}(s_i)$ is a vector formed by $g_k(s_i)$ corresponding to all the n-grams (n varying from 1 to 4). c_i is the i^{th} generated sentence. Finally, the scores of various n-grams can be combined to calculate CIDER as follows:

$$\text{CIDER}(c_i, s_i) = \sum_{n=1}^{N} w_n \, \text{CIDER}_n(c_i, s_i) \tag{18}$$

Note that we usually report the scores of BLEU, METEOR and ROUGE-L in percentages since they are in the range of $[0, 1]$. As CIDER scores range in $[0, 10]$, we display them in real values.

5 EVALUATION

In this section, we evaluate different approaches by comparing the scores of the metrics mentioned in Sec. 4.3. We explore the evaluation results to answer the following Research Questions (RQs):

RQ1: What is the effectiveness of our proposed model CoCoGUM?
RQ2: Does employing global context help improve code summarization results?
RQ3: What is the stability of CoCoGUM?

5.1 RQ1: What is the effectiveness of our proposed model CoCoGUM?

We evaluate the effectiveness of CoCoGUM by comparing it to the most recent works on code summarization.

- **Code-NN** [7] is the first neural approach that learns to generate summaries of code snippets. It is a classical encoder-decoder framework in NMT that encodes code to context vector with attention mechanism and then generates summaries in decoder.
- **Ast-attendgru** and **Attendgru** [10] are essentially encoder-decoder network using GRUs with attention. Ast-attendgru is a multi-encoder neural model that encodes both code and AST.
- **H-Deepcom** [9] is the SBT-based model, which is more capable of learning syntactic and structure information of Java methods.
- **Hybrid-DRL** [8] is an improved approach with hybrid code representations (with ASTs) and deep reinforcement learning. It encodes the sequential and structural content of code by LSTMs and tree-based LSTMs and uses a hybrid attention layer to get an integrated representation.

Tab. 2 illustrates the evaluation results of various code summary generation baselines. CoCoGUM_m is a mini (simplified) version of CoCoGUM without inter-class context, keeping only the intra-class context (class name representations learned by the Transformer-based sentence embedding model). From the results, we can observe that CoCoGUM_m and CoCoGUM perform the best. In particular, CoCoGUM achieves better performance than CoCoGUM_m. Other neural models outperform Code-NN because Code-NN only uses raw code token sequences and fails to capture the semantic and syntax information. H-Deepcom applies SBT to capture semantic and structural information but the encoded information cannot be fully utilized in the decoding phase. Ast-attendgru combines the 3-channel information of code, SBT and summary in the decoder to generate better summaries and it outperforms the best among all the baselines. The comparison result show that our proposed model CoCoGUM is effective in terms of the four metrics: BLEU-4, METEOR, ROUGE-L and CIDER.

Table 2: Comparison for code summarization baselines.

MODEL	BLEU-4	METEOR	ROUGE-L	CIDER
Code-NN	13.37	7.99	20.42	0.44
Hybrid-DRL	14.67	8.86	22.33	0.69
H-Deepcom	15.16	9.81	24.83	0.56
Attendgru	17.68	12.49	28.32	0.88
Ast-attendgru	17.81	12.97	29.33	0.96
CoCoGUM_m	18.07	13.04	29.76	0.96
CoCoGUM	**19.49**	**14.70**	**32.42**	**1.20**

5.2 RQ2: Does employing global context help improve code summarization results?

We apply two levels of global context: intra-class level and inter-class level. We compare the performance of CoCoGUM_m (employing intra-class context) and CoCoGUM (employing inter-class context). Since the functionality of a method has a strong relationship with its corresponding class, CoCoGUM_m uses the information of class names to help generate summaries. We split a class name to a sequence of words and embed it to a feature vector to guide the summaries generation process.

CoCoGUM additionally uses the inter-class context information by using the UML class diagrams to capture the $\langle class, class \rangle$ relationship and generate better summaries of source code. We first obtain the embedding of each node based on the class information and UML graph structure. Then we encode classes based on MR-GNN and eventually get the representation of all nodes via the attention mechanism. From Tab. 2, we can see that both CoCoGUM_m and CoCoGUM outperform other baselines and CoCoGUM performs better in four metrics. As we know, METEOR and ROUGE-L consider both precision and recall. CIDER is a consensus-based evaluation. CoCoGUM achieves higher precision and recall and a better consensus sentence by considering both global and local information. Thus we can conclude that employing global context information is helpful for generating high-quality summaries.

5.3 RQ3: What is the stability of CoCoGUM?

We evaluate the stability of CoCoGUM on two perspectives: (1) how does it perform with different dataset splittings; (2) how does it perform on source code and comments of different lengths.

5.3.1 Stability on different dataset splittings. Different projects have different coding style conventions and naming conventions (leading to different vocabularies) due to programmers' programming style or companies' coding guidelines. For example, when predicting the summary of a given method, if similar methods

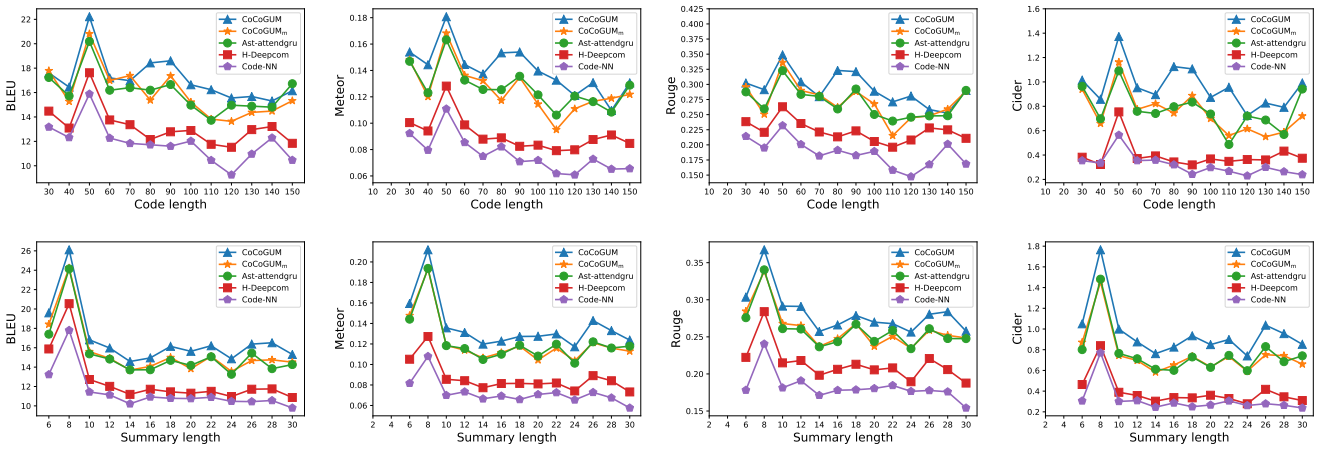


Figure 6: Comparison on different code lengths and summary lengths

Table 3: Evaluation results on package-wise dataset

MODEL	BLEU-4	METEOR	ROUGE-L	CIDER
Code-NN	14.97	9.13	22.65	0.58
H-Deepcom	17.17	11.56	26.89	0.78
Attendgru	20.40	14.55	32.17	1.22
Ast-attendgru	20.58	14.74	32.40	1.25
CoCoGUM$_m$	20.87	15.11	33.31	1.27
CoCoGUM	**22.20**	**16.51**	**35.48**	**1.50**

Table 4: Evaluation results on method-wise dataset

MODEL	BLEU-4	METEOR	ROUGE-L	CIDER
Code-NN	17.34	11.89	27.28	0.82
H-Deepcom	18.99	13.26	29.65	0.96
Attendgru	23.04	16.30	34.56	1.53
Ast-attendgru	23.25	16.25	34.30	1.56
CoCoGUM$_m$	24.34	17.94	37.90	1.74
CoCoGUM	**24.93**	**18.33**	**38.89**	**1.80**

(probably from the same project) exist in the training set, then the predicting result is more likely to be accurate. We evaluate how this factor affects the model performance by experimenting CoCoGUM and other models for different dataset splittings.

Tab. 2 shows the results on project-wise splitting (default dataset splitting): methods in one project can only exist in one of the training, validation or test sets. Tab. 3 and Tab. 4 report results on package-wise and method-wise splitting, respectively. Firstly, the results show that scores for method-wise splitting is generally higher than package-wise splitting, and package-wise splitting higher than project-wise splitting. This is consistent with the instinct that if the model has seen similar methods (from the same project/package), it will perform better when predicting methods from that project/package. Secondly, the results show that CoCoGUM still outperforms other approaches in all metrics. We conclude that CoCoGUM has good stability on different dataset splittings.

5.3.2 Stability on code and comments of different lengths. We further analyze the stability of CoCoGUM on code and comments of different lengths. Fig. 5 displays the length distribution of code and summary on testing set. The lengths of most code are less than 150 and the lengths of most summaries are less than 30, so we conduct the experiment to evaluate the performance for code lengths varying from 1 to 150 and summary lengths varying from 1 to 30.

Fig. 6 presents the performance of CoCoGUM and other models for code and summaries of different lengths. In Fig. 6, the first row shows that CoCoGUM performs the best on four metrics of varying code lengths. All the models achieve high performance on code length near 50. The reason is that all the models are data-driven and perform better when more data is available. Similarly, the second row of Fig. 6 shows that CoCoGUM outperforms others and all models achieve higher performance on summary length near 8. Additionally, the performance of baselines decreases a lot when code length increases, while our model performs more stably on all metrics even for longer code and summaries. We conclude that CoCoGUM has good stability on code and comments of different lengths.

5.4 Case Study

To conduct qualitative analysis of our approach, we compare the generated summaries from different models and present 2 case studies as shown in Tab. 5. We have the following observations from the results which demonstrates the superiority of our approach:

- In general, CoCoGUM and CoCoGUM$_m$ can generate summaries better than other baselines and the summaries of our models are more similar to the ground truth.
- Compared to baselines, CoCoGUM and CoCoGUM$_m$ can generate summary tokens that accurately describe the purpose of the method but cannot be found within the scope of the method. For example, one important summary token should be generated in case 1 is "class" but it cannot be generated only based on the local context (source code itself). CoCoGUM and CoCoGUM$_m$ also consider the global context (e.g. the class name "ClassVisitor"). Therefore, they are able to produce better summarizations.

Table 5: Case studies

Case 1	`public FieldVisitor visitField(`
	`  final int access,`
	`  final String name,`
	`  final String descriptor,`
	`  final String signature,`
	`  final Object value) {`
	`  if (cv != null) {`
	`  return cv.visitField(access, name, descriptor, signature, value);`
	`  }`
	`  return null;`
	`}`
	
	`Enclosing Class Name: ClassVisitor`

Ground Truth	*visits a field of the class*
CoCoGUM	**visit a field in the class**
CoCoGUM$_m$	visit a field in the given class
Ast-attendgru	this method is called when a constant is a member of
Attendgru	this method is called when a constant is a member of
H-Deepcom	visit a visitor for a the
Hybrid-DRL	adds the visitor to the visitor
Code-NN	create the visitor to the the the

Case 2	`public PropertyStatus getProperty(QualifiedName propertyName) throws DAVException {`
	`  Collection names = new HashSet();`
	`  names.add(propertyName);`
	`  URLTable result = getProperties(names, IContext.DEPTH_ZERO);`
	`  URL url = null;`
	`  try {`
	`  url = new URL(locator.getResourceURL());`
	`  } catch (MalformedURLException e) {`
	`  throw new SystemException(e);`
	`  }`
	`  Hashtable propTable = (Hashtable) result.get(url);`
	`  if (propTable == null)`
	`  throw new DAVException(Policy.bind("exception.lookup", url.toExternalForm()));`
	`  return (PropertyStatus) propTable.get(propertyName);`
	`}`
	
	`Enclosing Class Name: AbstractResourceHandle`
	`UML (partial): ['AbstractResourceHandle',' PropertyStatus ',{'relationtype': ' DEPEND '}]`

Ground Truth	*return the property status for the property with the given name*
CoCoGUM	**return a property status object for the given property name**
CoCoGUM$_m$	get a property from the given property
Ast-attendgru	get the property value of the property
Attendgru	get the property value of the property
H-Deepcom	get the property property of the the the
Hybrid-DRL	return the property value
Code-NN	get the property of the the the the

- Case 2 shows that CoCoGUM generates the best summary containing the full meaning as in ground truth. Specifically, the ground truth summary contains a phrase "property status" that cannot be generated by other models. The reason is that the enclosing class of the target method is `AbstractResourceHandle`, which has a `DEPEND` relation with class `PropertyStatus`. This information is learned by our MR-GNN module which analyzes the corresponding UML diagram. One may notice that although the token 'PropertyStatus' also appears in the method local context, it is hard for other models to capture this information without the help of global context, which emphasizes the importance of class-class relationships.

6 RELATED WORK

In this section, we survey two lines of research that are related to our work.

6.1 Source code representation

Previous works suggest various representations of source code for follow-up analysis [7, 12, 45–52].

Allamanis et al. [12] and Iyer et al. [7] simply consider source code as plain text and use traditional token-based methods to capture lexical information. Gu et al. [45] use the Seq2Seq model to learn intermediate vector representations of queries in natural language and then predict relevant API sequences. Mou et al.

[46] propose a novel Tree-Based Convolutional Neural Network (TBCNN). In TBCNN, program vector representations are learned by the coding criterion; structural features are detected by the convolutional layer; and TBCNN can handle trees with varying children sizes with the continuous binary tree and dynamic pooling. There are works [47, 48] that learn representations of source code by input/output pairs. Piech et al. [47] introduce a neural network method to encode programs as a linear mapping from an embedded precondition space to an embedded postcondition space and propose an algorithm for feedback at scale using these linear mappings as features. Parisotto et al. [48] produce a continuous representation of the set of input/output pairs. Given the continuous representation of the examples, they use RNNs to synthesize a program by incrementally expanding partial programs. Dam et al. [49] build a language model for modeling software code using LSTMs. Ling et al. [50] and Allamanis et al. [51] combine the code-context representation with representations of other modalities (e.g. natural language) to synthesize code. Alon et al. [53, 54] represent a code snippet as a set of compositional paths in the abstract syntax trees. Zhang et al. [52] propose an AST-based Neural Network (ASTNN) that splits each large AST into a sequence of small statement trees, and encodes the statement trees to vectors by capturing the lexical and syntactical knowledge of statements and apply the representation to tasks such as source code classification and code clone detection.

6.2 Source code summarization

Popular source code summarization approaches include rule-based approaches, IR-based approaches, , learning-based approaches, etc.

6.2.1 Rule-based and IR-based approaches. In the early stage of automatic source code summarization, rule-based approaches and information retrieval (IR) techniques and are widely used [4, 5, 35, 55, 56].

Sridhara et al. [4] design heuristics to choose statements from Java methods and use the Software Word Usage Model (SWUM) to identify keywords from those statements and create summaries though manually-defined templates. The summaries are generated according to the function/variable names via these templates.

The basic ideas of IR approaches are retrieving terms from source code for generating term-based summaries or retrieving similar source code and use its summary as the target summary [5, 35, 55, 56]. Haiduc et al. [5, 55] use the TF-IDF metric to rank the terms in a method body based on the importance those terms are to that method. TF-IDF gives higher scores to keywords which are common within a particular method body, but rare throughout the rest of the source code. They treat each function of source code as a document and index on such a corpus by LSI or VSM, then the most similar terms based on cosine distances between documents are selected as the summary. Eddy et al. [56] use topic modeling to improve the work, and Rodeghero et al. [35] use eye-tracking and modify the weights of VSM for better code summarization. Code clone detection techniques are used to retrieve similar code snippets from a large corpus and reuse their summaries as the targets [57, 58].

6.2.2 Learning-based approaches. Movshovitz-Attias and Cohen [11] use a statistical language model based approach to predict comments from Java source code using topic models and n-grams. Allamanis et al. [12] create the neural logbilinear context model for suggesting method and class names by embedding them in a high dimensional continuous space. Allamanis et al. [13] also suggest a convolutional model for summary generation that uses attention over a sliding window of tokens. They summarize code snippets into extreme, descriptive function name-like summaries.

The NMT-based models are also widely used to generate summaries for code snippets [7–10, 14, 15, 59]. Iyer et al. [7] designed a token-based neural model using LSTMs and attention for translation between source code snippets and natural language descriptions. Haije [14] model the code summarization problem as a machine translation task, and some translation models such as Seq2Seq [20] and attentional Seq2Seq are employed. Hu et al. [15] structurally flatten an AST and then pass it on to a standard Seq2Seq model. Hu et al. [59] leverage API sequences and improve the summary generation by learned API sequence knowledge. LeClair et al. [10] designed a multi-input neural network using GRUs and attention to generate summaries of Java methods. Wan et al. [8] encode the sequential and structure content of code by LSTM and tree-based LSTM and use a hybrid attention layer to get an integrated representation. The performance is further improved by deep reinforcement learning [60] to solve the exposure bias problem during.

Compared with the above work, our approach also take the general encoder-decoder architecture but incorporate global context into consideration with the help of GNNs, resulting in better performance than the most recent works on code summarization.

7 CONCLUSION

In this paper, we propose a new code summarization model called CoCoGUM, which leverages two types of global context information, i.e., class names and UMLs, to assist with code summarization generation. Contemporary automatic code summarization approaches only consider a given code snippet in the local context, without considering its broader context information. We firstly represent intra-class context by using transformer-based representation of the class name that a given method belongs to and inject it into the Seq2Seq model for code summarization. Then CoCoGUM adopts our proposed model MR-GNN to learn the rich structural information behind the UMLs in order to capture the class-class relationships and further enhances the performance of automatic code summarization. The learned representations of class names and UMLs are incorporated into the Seq2Seq code summarization model with the help of our designed two-level attention mechanism to capture the different importance of differing context information. Experimental results has demonstrated the effectiveness of CoCoGUM and confirmed the usefulness of the two global context information.

We believe this work will shed some light on future research by pointing out that source code context including local and global context can play a vital role in summary generation. In the future, we plan to explore more information (e.g., call graphs, data flow graphs) which is shipped with source code to improve the quality of generated code summarization and enhance the interpretability on how the summaries are generated and how they relate to the corresponding global/local contexts.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4):81:1–81:37, 2018.

[2] Triet Huynh Minh Le, Hao Chen, and Muhammad Ali Babar. Deep learning for source code modeling and generation: Models, applications and challenges. *arXiv Preprint*, 2020. URL https://arxiv.org/abs/2002.05442.

[3] Yuxiang Zhu and Minxue Pan. Automatic code summarization: A systematic literature review. *arXiv Preprint*, 2019. URL https://arxiv.org/abs/1909.04352.

[4] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *ASE*, pages 43–52, 2010.

[5] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *ICSE*, volume 2, pages 223–226. ACM, 2010.

[6] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, pages 1724–1734. ACL, 2014.

[7] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *ACL*, volume 1. The Association for Computer Linguistics, 2016.

[8] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. Improving automatic source code summarization via deep reinforcement learning. In *ASE*, pages 397–407. ACM, 2018.

[9] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, 2019.

[10] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. In *ICSE*, pages 795–806, 2019.

[11] Dana Movshovitz-Attias and William W. Cohen. Natural language models for predicting programming comments. In *ACL*, volume 2, pages 35–40, 2013.

[12] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. Suggesting accurate method and class names. In *ESEC/SIGSOFT FSE*, pages 38–49, 2015.

[13] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. In *ICML*, volume 48, pages 2091–2100, 2016.

[14] Tjalling Haije. *Automatic comment generation using a neural translation model.* Bachelor's thesis, University of Amsterdam, 2016.

[15] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *ICPC*, pages 200–210, 2018.

[16] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Universal sentence encoder. *arXiv Preprint*, 2018. URL https://arxiv.org/abs/1803.11175.

[17] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *ICLR*, 2018.

[18] Grady Booch, James E. Rumbaugh, and Ivar Jacobson. *The unified modeling language user guide - the ultimate tutorial to the UML from the original designers.* Addison-Wesley object technology series. Addison-Wesley-Longman, 1999.

[19] James E. Rumbaugh, Ivar Jacobson, and Grady Booch. *The unified modeling language reference manual.* Addison-Wesley-Longman, 1999.

[20] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS*, pages 3104–3112, 2014.

[21] Alexander M. Rush, Sumit Chopra, and Jason Weston. A neural attention model for abstractive sentence summarization. In *EMNLP*, pages 379–389, 2015.

[22] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *CVPR*, pages 3156–3164, 2015.

[23] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[24] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017.

[26] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing [review article]. *IEEE Comput. Intell. Mag.*, 13(3):55–75, 2018.

[27] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.

[28] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Trans. Knowl. Data Eng.*, 2020.

[29] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521 (7553):436–444, 2015.

[30] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv Preprint*, 2018. URL https://arxiv.org/abs/1812.08434.

[31] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. In *ICLR*, 2020.

[32] Qitian Wu, Hengrui Zhang, Xiaofeng Gao, Peng He, Paul Weng, Han Gao, and Guihai Chen. Dual graph attention networks for deep latent representation of multifaceted social effects in recommender systems. In *WWW*, pages 2091–2102, 2019.

[33] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[34] Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. Gaan: Gated attention networks for learning on large and spatiotemporal graphs. In *UAI*, pages 339–349, 2018.

[35] Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney K. D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *ICSE*, pages 390–401. ACM, 2014.

[36] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *EMNLP*, pages 1412–1421, 2015.

[37] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *ICLR*, 2019.

[38] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv Preprint*, 2019. URL https://arxiv.org/abs/1909.09436.

[39] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Retrieval-based neural source code summarization. 2020.

[40] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *ACL*, pages 311–318. ACL, 2002.

[41] Satanjeev Banerjee and Alon Lavie. METEOR: an automatic metric for MT evaluation with improved correlation with human judgments. In *IEEvaluation@ACL*, pages 65–72. Association for Computational Linguistics, 2005.

[42] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, 2004.

[43] Ramakrishna Vedantam, C. Lawrence Zitnick, and Devi Parikh. Cider: Consensus-based image description evaluation. In *CVPR*, pages 4566–4575. IEEE Computer Society, 2015.

[44] Chin-Yew Lin and Franz Josef Och. ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *COLING*, 2004.

[45] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep API learning. In *SIGSOFT FSE*, pages 631–642, 2016.

[46] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, pages 1287–1293, 2016.

[47] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas J. Guibas. Learning program embeddings to propagate feedback on student code. In *ICML*, volume 37, pages 1093–1102, 2015.

[48] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv Preprint*, https://arxiv.org/abs/1611.01855, 2016.

[49] Hoa Khanh Dam, Truyen Tran, and Trang Pham. A deep language model for software code. *arXiv Preprint*, https://arxiv.org/abs/1608.02715, 2016.

[50] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Fumin Wang, and Andrew W. Senior. Latent predictor networks for code generation. In *ACL*, volume 1. The Association for Computer Linguistics, 2016.

[51] Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In *ICML*, volume 37, pages 2123–2132, 2015.

[52] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *ICSE*, pages 783–794, 2019.

[53] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *PACMPL*, 3(POPL):40:1–40:29, 2019.

[54] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *ICLR*, 2019.

[55] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the use of automated text summarization techniques for summarizing source code. In *WCRE*, pages 35–44. IEEE Computer Society, 2010.

[56] Brian P. Eddy, Jeffrey A. Robinson, Nicholas A. Kraft, and Jeffrey C. Carver. Evaluating source code summarization techniques: Replication and expansion. In *ICPC*, pages 13–22. IEEE Computer Society, 2013.

[57] Edmund Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *ASE*, pages 562–567. IEEE, 2013.

[58] Edmund Wong, Taiyue Liu, and Lin Tan. Clocom: Mining existing source code for automatic comment generation. In *SANER*, pages 380–389. IEEE Computer

Society, 2015.
[59] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with transferred API knowledge. In *IJCAI*, pages 2269–2275. ijcai.org, 2018.
[60] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8:229–256, 1992.