The Journal of Systems & Software 188 (2022) 111257

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Automatic source code summarization with graph attention networks☆

Yu Zhou [a,c], Juanjuan Shen [a], Xiaoqing Zhang [a], Wenhua Yang [a,c], Tingting Han [b], Taolue Chen [b,c,*]

[a] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China
[b] Department of Computer Science, Birkbeck, University of London, UK
[c] State Key Lab. for Novel Software Technology, Nanjing University, China

ARTICLE INFO

Article history:
Received 7 June 2021
Received in revised form 15 November 2021
Accepted 2 February 2022
Available online 8 February 2022

Keywords:
Source code summarization
Recurrent neural network
Graph neural network

ABSTRACT

Source code summarization aims to generate concise descriptions for code snippets in a natural language, thereby facilitates program comprehension and software maintenance. In this paper, we propose a novel approach–GSCS–to automatically generate summaries for Java methods, which leverages both semantic and structural information of the code snippets. To this end, GSCS utilizes Graph Attention Networks to process the tokenized abstract syntax tree of the program, which employ a multi-head attention mechanism to learn node features in diverse representation sub-spaces, and aggregate features by assigning different weights to its neighbor nodes. GSCS further harnesses an additional RNN-based sequence model to obtain the semantic features and optimizes the structure by combining its output with a transformed embedding layer. We evaluate our approach on two widely-adopted Java datasets; the experiment results confirm that GSCS outperforms the state-of-the-art baselines.

1. Introduction

Program comprehension is an indispensable activity in software development and maintenance, since programs must be sufficiently understood before they can be properly modified (Cornelissen et al., 2009). It is reported that, on average, software developers spend around 58% of time on program comprehension related activities (Xia et al., 2018). Numerous studies (Stapleton et al., 2020; Takang et al., 1996; Tenny, 1988; Woodfield et al., 1981) show that descriptive summaries, which explain the functionality of code snippets by brief natural language sentences, are conductive to software comprehension. However, manually writing code summaries is time-consuming and laborious. To ease the process, the last decade has witnessed a proliferation of automatic code summarization techniques (Song et al., 2019). These techniques were mainly based on pre-defined templates and keyword extraction (Hill et al., 2009; Sridhara et al., 2010, 2011). More recently, inspired by the application of deep learning in the field of natural language processing (NLP), researchers have resorted to deep learning for source code summarization. Particularly, models for neural machine translation (NMT) have become the mainstream approach, where, generally speaking, source code is viewed as pure text and an encoder–decoder framework is used to provide code summaries (Sutskever et al., 2014).

Similar to natural languages, programming languages also exhibit "naturalness" (Hindle et al., 2016), so adapting NMT models to source code can be potentially effective. For instance, Iyer et al. (2016) established the first end-to-end model, using Long–Short Term Memory (LSTM) networks with an attention mechanism to build a language model for summary generation. Allamanis et al. (2016) introduced a neural convolutional attention model to predict a short and descriptive name of a source code snippet. Bolin et al. (Wei et al., 2019) applied dual learning framework to train code summarization and code generation models based on the duality between the two tasks. Ahmad et al. (2020) employed the basic Transformer model to summarize source code and improved the behavior with relative position representation and a copy attention mechanism.

However, Maletic and Marcus (2001) suggested that software engineers must examine both the structural aspect of source code and the naturalness of code tokens to fully understand a program. Different from natural languages, source code is not merely a sequence of words (LeClair et al., 2020). Instead, it is strongly structured and has massive identifiers. Besides, compared with machine translation, source code has far fewer words directly related to summaries, and the functionality of the code is usually encoded in its nested structure. In light of this, researchers began

☆ Editor: Earl Barr.

* Corresponding author at: Department of Computer Science, Birkbeck, University of London, UK.
E-mail address: t.chen@bbk.ac.uk (T. Chen).

https://doi.org/10.1016/j.jss.2022.111257

to study approaches to extract the structural information of the code. In particular, the abstract syntax tree (AST), which expresses the syntactic structure of a program in a tree form, has become the mainstream tool for code analysis. For instance, Hu et al. (2018) proposed the algorithm SBT which can flatten the AST of a Java method to a sequence using nested parentheses. LeClair et al. (2019) introduced two encoders to encode the code tokens and the AST sequences separately. Shido et al. (2019) applied extended Tree-LSTM to learn tree structures in ASTs directly. LeClair et al. (2020) utilized Convolutional Graph Neural Networks (ConvGNNs), together with the source code sequence as separated inputs, to the code summarization model.

The aforementioned neural network models have advanced the state-of-the-art and can generate useful summaries in practice, but still have several limitations. First, source code usually contains complex nested structures, thus cannot be simply treated as plain texts which would otherwise lose too much information and inevitably degrade the model performance (Hellendoorn and Devanbu, 2017). Second, existing models are insufficient in processing structural information. For instance, SBT introduces numerous brackets, which aggravates the long dependency problem and introduces unnecessary redundant information. Neither Tree-LSTM nor ConvGNN considers the current context of the node in an AST. Namely, each node has its local characteristics in an AST. Even if two nodes are with the same identifiers, the roles they play in the local environment could be different, but are treated equally in Tree-LSTM and ConvGNN. As a result, for code summarization, the extraction and modeling of code structure require further investigation. Third, the order in source code is not fixed, and swapping partial tokens or statements may not affect its functionality (e.g., $a + b$ is equal to $b + a$). However, for basic RNNs which were designed to process sequential data, the position of each token is fixed during encoding as a result of their sequential structure.

To overcome these limitations, in this paper, we propose GSCS (**G**raph structure and **S**emantic sequence for **C**ode **S**ummarization), a novel approach which is based on graph structure and semantic sequence for code summarization. It is widely accepted that code is more amenable to graph or tree representations (Binkley, 2007; Ottenstein and Ottenstein, 1984), whereby we apply graph neural networks to capture the structural features of ASTs. To mitigate the out-of-vocabulary problem of AST nodes and assimilate inner representations, we put forward a variant of ASTs by splitting the method names, variable names, and long string literals into subtokens which are linked to the corresponding labeled nodes to retain the tree structure. Similar to the work of LeClair et al. (2019) which encoded code and AST separately, we adopt the sequence model, BiGRU (Chung et al., 2014), as the semantic encoder for modeling the tokenized code sequences.

Once a code snippet is parsed into an AST, it may contain redundancy. If one simply accumulates the child node features with Tree-LSTM or ConvGNN, noise contained in these nodes may propagate deeper especially when using multi-layer networks. To this end, we exploit Graph Attention Network (GAT) proposed by Petar et al. (Velickovic et al., 2017) to address the shortcomings of graph convolution. In a nutshell, GAT can learn to automatically assign different weights to neighbor nodes, encouraging the adoption of useful structural information while ignoring noise messages. Moreover, to handle the code sequence, we optimize the basic Seq2Seq model inspired by Iyer et al. (2016) who used a single embedding layer without any position feature. We combine BiGRU with a transformed embedding layer (such as a residual connection) to alleviate the effect of absolute position and long-term dependence, which considerably improves the quality of the generated summary.

In summary, in this paper, we highlight the importance of the structural information in code snippets and design a new neural network architecture to process both the structural and the semantic information, yielding a novel data-driven approach for code summarization. We conduct experiments on three public datasets collected from GitHub, and the results endorse the effectiveness of our approach over the-state-of-the-art solutions. Moreover, to facilitate replication and reuse of our work, we have made the implementation publicly available.[1]

Structure of the paper. The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 describes the background of BiGRU and GAT. Section 4 presents the details of our proposed approach. Section 5 empirically evaluates the performance of the new approach with comparisons to baselines. Section 6 discusses the threats to validity. Section 7 draws the conclusion and discusses briefly the future work.

2. Related work

As a critical task in software engineering, a multitude of studies have been conducted on code summarization. The goal of automatic code summarization is to generate a brief yet accurate representation of a code snippet. Heuristic rule-based methods were first proposed. Hill et al. (2009) extracted natural language phrases from source code identifiers, helping developers to quickly identify relevant program elements. Sridhara et al. (2010) summarized a Java method's overall actions by exploiting both structural and linguistic clues in the method. Sridhara et al. (2011) described a technique to automatically generate descriptive summaries for parameters of Java methods and provide sufficient context for developers to understand the role of the parameter in achieving the computational intent of a method. All of the above approaches are based on pre-defined templates and rules, which completely rely on prior knowledge of developers. Thus it is inevitable that they fail to work when the code content is outside the scope of the templates.

Several works exploited Information Retrieval (IR) technology for source code summarization. Wong et al. (2013) mined code-description mappings from Q&A sites and leveraged such mappings to generate description summaries automatically for similar code segments matched in open-source projects. Haiduc et al. (2010) applied Latent Semantic Indexing (LSI) to select the top five terms in the list ordered by cosine similarity to construct the summary. However, the effect of these IR-based approaches is limited by the similarity between the corpus and the tested examples, which may generate summaries completely different from the actual semantics.

In recent years, inspired by the development of machine learning, researchers began to investigate application of deep learning in software engineering. Similar to machine translation, code summarization is the mapping from programming language to natural language. Iyer et al. (2016) firstly presented an end-to-end neural network called CODE-NN, which used an embedding layer to encode the code tokens and combined them with Long–Short Term Memory (LSTM) networks via an attention mechanism to produce sentences that describe C# code snippets and SQL queries. Allamanis et al. (2016) designed a convolutional neural network to produce short, descriptive, function name-like summaries with two attention mechanisms, i.e., one predicts the next summary token based on the attention weights of the input tokens, while another one is able to copy a code token directly into the summary. Zhou et al. (2019) proposed to augment context information with techniques of program analysis and then automatically generate code comments. Dual learning (He et al., 2016) was first proposed to learn from unlabeled data based on the duality of tasks. It was further demonstrated

[1] https://github.com/sjj0403/GSCS.

that dual learning at a model level can solve a group of dual tasks, such as neural machine translation and text analysis. Bolin et al. (Wei et al., 2019) considered code summarization (CS) and code generation (CG) as a pair of dual tasks. They added regularization terms in the loss function to constrain the duality between the two Sequence-to-Sequence models. Zhang et al. (2020) proposed a retrieval-based neural source code summarization approach by enhancing the neural network model with the most similar code snippets retrieved from the training set. This approach still suffered from the limitation of similarity. Transformers (Vaswani et al., 2017) have become the mainstream architecture in machine translation, which works on seq2seq tasks using the multi-headed attention mechanism and realizes parallel computing in the encoder. Ahmad et al. (2020) presented a Transformer-based approach for source code summarization, replacing the original absolute position representation with relative position and adding a copy attention mechanism to the basic Transformer architecture, which has been proved to perform well.

Some neural network models consider the structural characteristics of code. Hu et al. (2018) proposed an SBT algorithm, transforming ASTs into a unique sequence as the input to an LSTM. LeClair et al. (2019) expand on this idea by separating the code sequence and the AST sequence into two different encoders to learn diverse features of code snippets. An AST is of a tree-structure, so the inner features will be omitted after being flattened while the problem of long-term dependence is aggravated due to the geometric growth of the sequence length. Thus Shido et al. (2019) applied extended Tree-LSTM for source code summarization that can keep an AST's internal structure. Fernandes et al. (2018) extended the code sequence encoders with Gated Graph Neural Networks that can reason about long-distance relationships in strong structured data. LeClair et al. (2020) used a ConvGNN-based encoder of graph2seq to model the AST, combined with an RNN-based encoder modeling the code sequence. Zügner et al. (2021) proposed a multilingual model for code summarization, called Code Transformer, which combined distances computed on structure and context in the self-attention operation. However, the latest technology is still facing the difficulty of extracting and employing structural information. ASTs are complex and huge, so simple feature accumulation may blur useful message without filtering.

3. Background

This section covers the supporting technologies behind our work, including bidirectional gated recurrent units and graph attention networks. Familiarity with neural network concepts like RNNs and GNNs are needed as they are the basics in our approach.

3.1. Bidirectional gated recurrent units

RNNs establish a connection between the front and rear units in the same layer, which considers the influence from front to back and makes the unit have a certain memory function. Since the semantics of one token is not only connected to the previous information but also closely related to the information after the current token. BiRNN (Schuster and Paliwal, 1997) was proposed to train two RNNs based on forward and backward sequences respectively, both of which are connected to an output layer. This structure provides the output layer with complete past and future context information for each item in the input sequence. As a variant of BiRNN, bidirectional gated recurrent unit (BiGRU) has been widely used in sentiment classification (Han et al., 2020), entity-relationship extraction (Lv et al., 2020) and text classification (Duan et al., 2020). At each time step t, the GRU (Cho et al., 2014) takes not only the input of the current step but also the hidden state outputted by its previous time step $t-1$. In a BiGRU, two independent GRUs are combined in a bidirectional fashion, with one reading the input sequence in the forward direction and the other in the backward direction. In addition, to capture the long-term dependence in a long sequence, BiGRU controls the flow of information through two different gating mechanisms, i.e., reset gate and update gate. The reset gate combines the new input information with the previous hidden state to filter memories that have nothing to do with the future work and the update gate defines the amount of previous memory saved to the current time step. Compared with another variant, BiLSTM (Hochreiter and Schmidhuber, 1997), BiGRU significantly reduces the number of parameters and training time (Chung et al., 2014). As a result, in our framework, we adopt BiGRU to encode the code sequence.

3.2. Graph attention networks

Graph neural networks (GNNs) embed the graph structure into the neural models and have been applied in various domains, for instance, sequence labeling, relationship extraction, event extraction, image classification, etc. Graph convolution networks (GCNs) (Kipf and Welling, 2017) extend convolution neural networks for images to handle graph data. However, GCNs fail to directly act on directed graphs or dynamic graphs, and all neighbor nodes are treated equally. To address these problems, graph attention networks (GAT) (Velickovic et al., 2017) were developed to adaptively control the contribution of adjacent nodes by introducing the attention mechanism. The advantage is that it can amplify the influence of the most important part of the data. The main idea of GAT is to compute the hidden representations of each node in the graph, by attending over its neighbors, following a self-attention strategy. It means that the weight of adjacent node features is completely determined by the node features themselves and is independent of the graph structure. To learn the attention weights in different subspaces, GATs integrate the outputs of multiple graph attention layers (GALs). Each GAL trains the parameters of the specific attention function to determine the weights of neighboring nodes when aggregating feature information.

4. Approach

Fig. 1 gives an overview of our approach GSCS, which mainly consists of three parts: data processing, model training, and online code summary generation. The neural network architecture is designed to process both semantic and structural information from source code. In particular, BiGRU and GAT are utilized to process code tokens (cf. Section 4.2.1) and ASTs (cf. Section 4.2.2) respectively.

4.1. Data processing

Three types of inputs are subject to data processing, i.e., source code, related tokenized code, and the corresponding summary. Specifically, the summary is the first descriptive sentence extracted from its Javadoc document. We parse the summary and tokenize any CamelCase or snake_case user-defined identifiers once found. The source code is used to generate structure information, i.e., the node sequence and the adjacency matrix, and is sent to the structural encoder. The tokenized code is generated from the source code and used as the input of the semantic encoder. For the above three sequences, we set a length threshold. The ones below the threshold are filtered. Vocabularies for code, node, and summary are built from the training sets, and the words which are out of the vocabularies are replaced with "unknown".

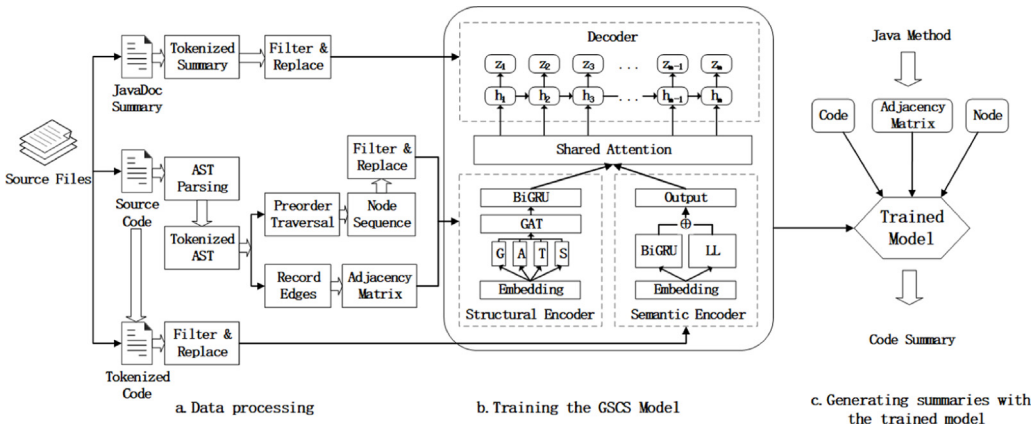

Fig. 1. An overview of the GSCS approach.

To generate the structural dataset, we parse the source code of each Java method to ASTs and filter out those that fail to parse. Based on the obtained ASTs, we set rules to tokenize partial terminal nodes and mark them with specific class labels (such as method names, variable names, and long strings), which is to preserve the tree structure of the ASTs and equip each node with explicit semantic information. Example 1 provides a Java method, which serves as a running example to show the details of splitting specific nodes.

Example 1. An example to show the details of tokenized AST

```
public void ClientInit(ParsingEvent pe){
    m_teamName = pe.get("team_name");
    CaseEvent ce = new CaseEvent(this, m_teamName);
    for(CaseEventListener cel:CEListeners)
        cel.Connecting(ce);
}
```

Compared with the original AST which is parsed with javalang and shown in Fig. 2, three additional labeled nodes (i.e., MethodnameSplit, VariableSplit and StringSplit) are designed to mark the nodes that need to be tokenized. The original method name node "ClientInit" is replaced by the "MethodnameSplit" node connected with several sub-nodes, i.e., "Client" and "Init", which are generated in accordance with the CamelCase rule. For string nodes, we filter quotation marks out first, and then treat the blank space as the separator. If the generated list still contains CamelCase or snake_case tokens, we carry out the segmentation again. Following these rules, we obtain the final graphical structure which ensures that each node retains its own semantic information. We found that the tokenized AST reduces the vocabulary overflow rate and is beneficial to the attention mechanism. The tokenized AST of Example 1 is shown in Fig. 3. We obtain the node sequence by preorder traversing the tokenized AST and record edge information between nodes with the adjacency matrix (cf. Section 4.2.2 for details).

4.2. Code summarization model

The architecture of our model is illustrated in Fig. 4. The model contains two encoders, i.e., the semantic encoder and the structural encoder, which are to process tokenized code sequences and tokenized ASTs respectively. The semantic encoder builds up a language model on the code sequence with BiGRU, while the structural encoder uses multiple Graph Attention Layers (GALs) connected with BiGRU to encode tokenized AST. In general, given the input code sequence $X = (x_1, \ldots, x_n)$, the input node sequence $V = (v_1, \ldots, v_N)$ and the adjacency matrix A, the code summarization model aims to learn to generate the summary sequence $Z = (z_1, \ldots, z_m)$ based on the conditional probability

$$p(z_1, \ldots, z_m \mid x_1, \ldots, x_n, v_1, \ldots, v_N, A)$$

As shown in Fig. 4, our model architecture is mainly composed of the semantic encoder, the structural encoder and the decoder. In what follows, we introduce the details of these three parts.

4.2.1. Semantic encoder

We opt for a single layer BiGRU (Chung et al., 2014), a variant of RNN, to encode the code sequence after an embedding layer, which is simpler than LSTM but has a similar effect. Formally,

$$\text{OutX},\ \text{hidden} = \text{BiGRU}(\text{EM}(X))$$

where OutX and hidden are the two outputs of BiGRU based on the embedded input EM(X). In particular, *hidden* represents the final hidden state of the BiGRU, which will be sent to the decoder as the initial hidden state. OutX records the output features at each time step, which consists of two independent GRUs' outputs with opposite directions, i.e., $\text{OutX} = [\overrightarrow{\text{OutX}} \parallel \overleftarrow{\text{OutX}}]$. To record both the forward and backward contextual information in the output of BiGRU, a vector addition is applied on OutX, i.e.,

$$\text{GRUOutX} = \overrightarrow{\text{OutX}} + \overleftarrow{\text{OutX}}$$

As a sequence model, RNN emphasizes the order of the processed tokens. However, in the current case some of the tokens are indeed swappable. As a result, we put forward a *transformed embedding layer* which fuses the features of the embedding layer with the output of BiGRU to reduce the influence of absolute positions. Specifically, through a learnable linear transformation, i.e.,

$$\text{EmOutX} = \text{Linear}(\text{EM}(X)),$$

we obtain EmOutX, which is to be added to GRUOutX, yielding the final output OutputX of semantic encoder.

$$\text{OutputX} = \text{EmOutX} + \text{GRUOutX}$$

4.2.2. Structural encoder

Similar to the semantic encoder, the structural encoder first sends the sequences of nodes to an embedding layer, by which each token is mapped to a vector. To take advantage of GAT, we treat the tokenized AST as an undirected graph $G = (V, E, A)$, where V and E are respectively the sets of nodes and edges, and A is the corresponding (symmetrical) adjacency matrix. In particular, $V = \{v_1, \ldots, v_N\}$ is generated from the embedding layer (where N is the number of nodes). To abuse the notation slightly, we also use $v_i \in R^F$ to denote the feature vector of the

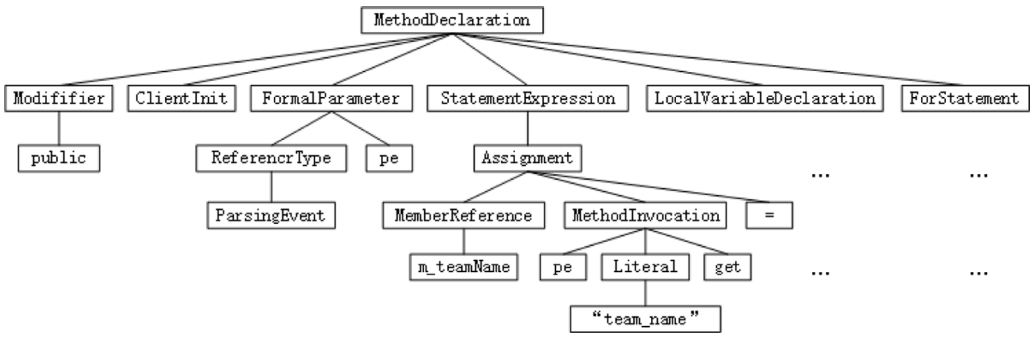

Fig. 2. Parsed AST of Example 1 with javalang.

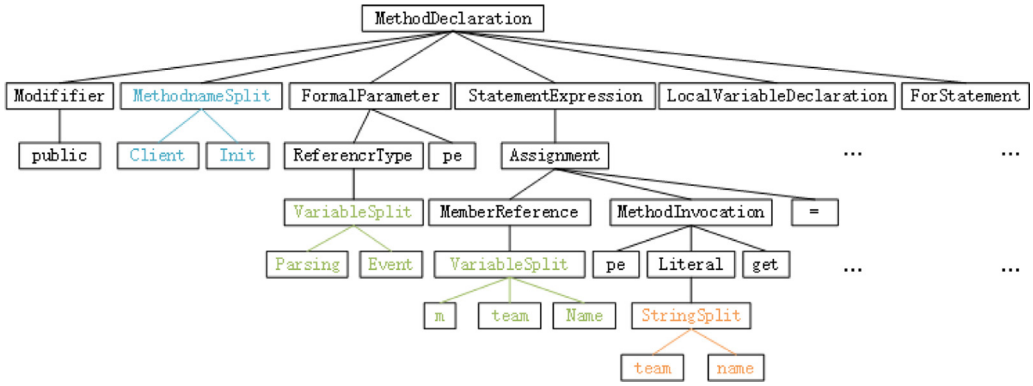

Fig. 3. Tokenized AST of Example 1 with our approach.

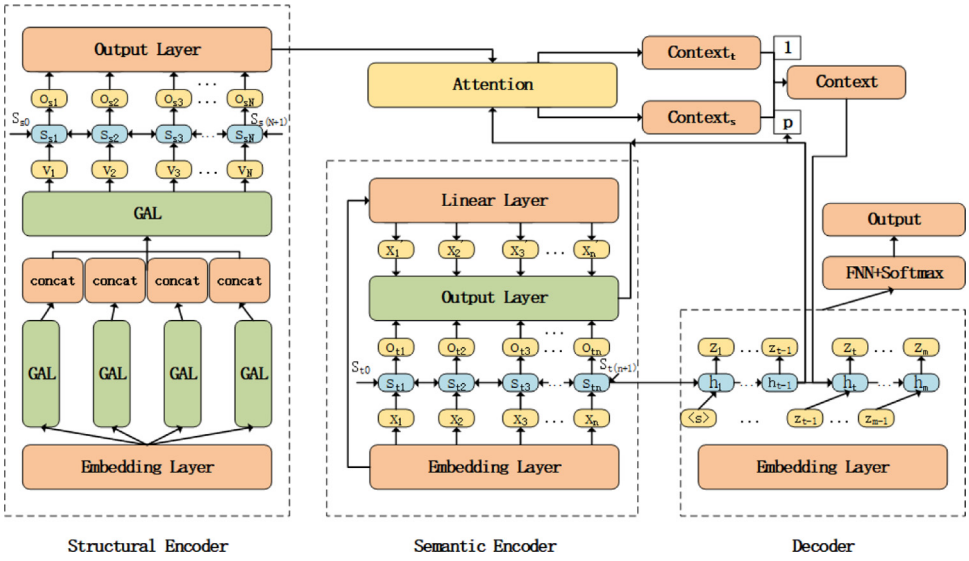

Fig. 4. Model architecture of our approach.

node (where F is the dimension of the feature space). We send $G = (V, E, A)$ to GAT for updating the features of each node, i.e.,

$$\widetilde{V} = \mathrm{GAT}(V, E, A)$$

Structure of GAL. The graph attentional layer (GAL) is the sole layer utilized throughout the GAT architecture (Velickovic et al., 2017). As shown in Fig. 4, the structural encoder stacks GALs to construct GAT. Each GAL selectively aggregates node information within 1-hop neighborhood when updating node features to produce a new set of node vector $V' = \{v'_1, \ldots, v'_N\}$. (Note that $v'_i \in R^{F'}$ and F' can be different from F.) Firstly, to obtain high-level features one learnable linear transformation W is required to transform the input feature v to Wv, where $W \in R^{F' \times F}$ is referred to as a weight matrix.

The core of GAL is to introduce self-attention to calculate the attention coefficient q for each pair of nodes, which is determined by the feature vector itself and other nodes. $q_{i,j}$ indicates the importance of node j's features to node i and is defined as

$$q_{i,j} = b^T [W v_i \parallel W v_j] \tag{1}$$

where $\|$ is the concatenation operation and $[Wv_i \| Wv_j] \in R^{2F'}$ is a column vector, $b^T = [b_1^T, b_2^T] \in R^{2F'}$ is a row vector. b^T is to be learned during model training. Intuitively, b_1^T is the self attention coefficient while b_2^T is the attention coefficient of the node j.

As some pair of nodes may not have connections, the masked attention mechanism, adopted by Petar et al. Velickovic et al. (2017), assigns the attention coefficient to those connected nodes only, which means that the node feature updating only focuses on its neighbors and itself. Based on the edge matrix A, if there is an edge from node j to node i, the attention coefficient $q_{i,j}$ is calculated by Eq. (1), otherwise it is set to be infinity.

$$q_{i,j} = \begin{cases} b^T[Wv_i \| Wv_j], & A[i][j] > 0 \\ \infty, & o/w \end{cases} \tag{2}$$

By Eq. (2), we can observe that the attention mechanism is asymmetric, thereby we regard the tokenized AST as an undirected graph to enable the network to learn bidirectional features. Namely, the weight assigned by the node i to its neighbor j can be different from the weight which j allocates to i, although the graph G is undirected.

To ensure the comparability of coefficients on different nodes, the softmax function is applied to normalize these scores, after applying a LeakyReLU nonlinearity with negative input slope α to the obtained $q_{i,j}$. Hence, the attention coefficient is calculated as

$$\begin{aligned} \beta_{i,j} &= \text{softmax}(\text{LeakyReLU}(q_{i,j})) \\ &= \frac{\exp(\text{LeakyReLU}(b^T[Wv_i \| Wv_j]))}{\sum_{k\in N_i}\exp(\text{LeakyReLU}(b^T[Wv_i \| Wv_k]))} \end{aligned} \tag{3}$$

where N_i is the set of all adjacent nodes of the node i (including itself); $\beta_{i,j}$ represents the final weight coefficient assigned by the node i to the node j and $q_{i,j}$ is obtained by Eq. (2).

All the nodes in N_i which are weighted by its attention coefficients are now combined to form a high-level node vector, followed by the nonlinearity activation function *sigmod* to standardize the updated features:

$$v_i' = \sigma\left(\sum_{j\in N_i}\beta_{i,j}Wv_j\right) \tag{4}$$

Eq. (4) describes the update of a node from v_i to v_i' through a single GAL. To capture the characteristics of nodes from various angles, usually K independent attention mechanisms (multi-head attention) can be built simultaneously as follows.

$$v_i'' = \|_{k=1}^{K}\sigma\left(\sum_{j\in N_i}\beta_{i,j}^k W^k v_j\right) \tag{5}$$

where $\|$ is the concatenation operation and K is the number of heads. Intuitively, each independent GAL learns a different weight matrix W^k. The aggregated features from each GAL are concatenated to obtain v_i'' that contains KF' features, so another GAL is needed to realize the transformation of dimensions from KF' to F''.

We further introduce a GAL which takes the concatenation of the outputs of the above K GALs as the input, namely,

$$\widetilde{v}_i = \sigma(\sum_{j\in N_i}\beta_{i,j}^* W^* v_j'') \tag{6}$$

where $W^* \in R^{F''\times KF'}$. The purpose of this GAL is to aggregate the neighboring node features, which encourages identifying deeper information. As a result, the original node vector V is updated as $\widetilde{V} = \{\widetilde{v_1}, \widetilde{v_2}, \ldots, \widetilde{v_N}\}$ with the dimension of 512 through the GAT shown in Fig. 4.

Finally, since GAT only captures the features of its local neighbors (i.e., the information we aggregate is within 2 hops), another BiGRU is connected with GAT to add the global information. We carry out preorder traversal on the updated nodes $\widetilde{V}$, obtaining the node sequences in accordance with the order of the code representation and send them to BiGRU.

$$\text{OutV},\ \text{hidden} = \text{BiGRU}(\widetilde{V})$$

Similar to the semantic encoder, the final output of the structural encoder is based on a vector addition, viz., $\text{OutputV} = \overrightarrow{\text{OutV}} + \overleftarrow{\text{OutV}}$.

4.2.3. Decoder

For the Seq2Seq framework, the attention mechanism can effectively guide decoding by assigning weights to the input sequence according to the last hidden state, so the model can focus on some more significant parts. In our approach, we adopt the global attention mechanism proposed by Luong et al. (2015). It defines an individual context vector c_i for predicting each target word z_i as a weighted sum of all hidden states $s_1, \ldots, s_n$ in the encoder, i.e.,

$$c_i = \sum_{j=1}^{n}\alpha_{ij}s_j \tag{7}$$

where n is the length of the input sequence and the weight α_{ij} of each hidden state s_j is computed as

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{m}\exp(e_{ik})}$$

where $e_{ij} = score(h_{i-1}, s_j)$. Here *score* refers to a content-based function which scores how well the inputs around position j and the output at position i match. We choose the following score function proposed by Luong et al. (2015) to calculate the alignment score vector

$$score(h_{i-1}, s_j) = U_a^T\tanh(W_a[h_{i-1} \| s_j]) \tag{8}$$

Here, s_j represents the hidden state at time step j in the encoder, h_{i-1} is the last hidden state at the time step i in the decoder. After concatenating h_{i-1} and s_j, the activation function tanh is used to score the attention weight, where U_a^T and W_a are learnable network parameters.

When using double encoders, the previous work tends to adopt a dual-attention architecture which computes the weight distribution of two inputs with different parameters and concatenates the context vector directly. Our approach opts to use a shared attention mechanism aiming to assign different weights proportional to the two encoders' hidden state. As shown in Fig. 4, the weight is decided by the last state of the decoder h_{i-1} and is set to $1 : p$, where $p = \sigma(\gamma^{\text{softmax}}(h_{i-1}))$. Based on the ratio, we generate the final c_i vector as

$$c_i = \sum_{j=1}^{n}\alpha_{ij}\text{OutputX}_j + p\sum_{j=1}^{N}\alpha_{ij}'\text{OutputV}_j$$

where α_{ij} and α_{ij}' are obtained from the alignment model shown in Eq. (8); n and N are the sequence lengths of the two encoders respectively; OutputX_j and OutputV_j represent the final outputs generated by two encoders respectively. In this manner, the model can adaptively adjust the attention weights to aggregate the text and structural information.

Based on the last hidden state of the decoder and the concatenation of the embedded z_i with the context vector c_i, we obtain two outputs at each step of the GRU, i.e., outZ_i and h_i. The purpose of the decoder is to generate the target word according to the distribution on the output vocabulary, which is guided

by the shared attention and the output of the GRU at the last time step. Specifically, we perform a linear transformation on the concatenation of out_i and c_i and apply the softmax function to normalize the distribution.

$$\text{outZ}_i,\ h_i = \text{GRU}([\text{EM}(z_i) \parallel c_i],\ h_{i-1})$$

$$\text{dist} = \text{softmax}(W_z[\text{outZ}_i \parallel c_i])$$

According to the probability distribution dist, the token with the highest probability in the target vocabulary is selected. In our approach, we adopt beam search with a width of 4, a heuristic graph search algorithm.

5. Evaluation

In this section, we first introduce our experimental set-up, followed by overall results and the ablation study to evaluate different approaches by measuring their accuracy on generating Java methods' summaries.

5.1. Experimental setup

Datasets. We conduct experiments on three publicly available Java datasets. The first one collected by Hu et al. (2018) contains over 87,000 Java methods with relatively complex inner structures. The second one extracted by Leclair and Mcmillan (2019) contains over 2 million Java methods with relatively simple inner structures. The third corpus is built by Husain et al. (2019) to enable an evaluation of progress on code search. It includes six programming languages and we select the Java subset which contains about 500,000 Java methods.

For each dataset, we prepare two corresponding datasets, i.e., the structural dataset and the semantic dataset for the structural encoder and semantic encoder respectively. For the semantic dataset, we directly employ the tokenized datasets by Ahmad et al. (2020) and LeClair et al. (2019) respectively, which would facilitate the comparisons with baselines. In the Hu dataset, the literals in a Java method were replaced by specified tokens and then further split source code tokens of the form CamelCase and snake_case to their respective sub-tokens. In the LeClair dataset, all non-alphanumeric characters in the code sequence were filtered and identifiers were split into subtokens. As the Husain dataset does not provide a tokenized version, we simply tokenize CamelCase and snake_case into sub-tokens and filter empty functions or data containing chinese words. None of the datasets include ASTs, so we use the javalang library[2] to generate the associated ASTs from the raw source code and filter out the part that fail to parse. For the summary sequence, the first sentence of Javadoc was extracted as the natural language description in the Hu dataset while the first line of the Javadoc was selected as the corresponding summary in the LeClair dataset. Similarly, we extract the first sentence from the original JavaDoc and filter out low-quality data, such as non-English descriptions, incomplete sentences, etc.

Table 1 shows the partition of the three datasets for the training, validation and test respectively after filtering. In addition, to reduce the impact of the order of the data on the model, all datasets are shuffled in advance.

Metrics. We evaluate the source code summarization performance using three metrics, BLEU-4 (Papineni et al., 2002), METEOR (Denkowski and Lavie, 2014), and ROUGE-L (Lin, 2004). These metrics are widely used in the literature to measure the quality of the generated summaries. BLEU is defined as the geometric mean of n-gram matching precision scores multiplied by a brevity penalty to prevent very short generated sentences. Particularly, we compute it based on the NLTK[3] library and choose sentence level BLEU-4 with the smoothing function 4. METEOR combines the unigram matching precision and the recall scores using harmonic mean values and employs the synonym matching mechanism. ROUGE-L computes the length of the longest common sub-sequence between the generated sentence and the reference, and focuses on the recall scores.

Table 1
Corpus statistics for the partitioning of the three Java datasets.

Dataset	Training	Validation	Test
Hu dataset	69,696	8,704	8,704
LeClair dataset	1,930,944	105,664	90,624
Husain dataset	337,760	11,136	20,896

Table 2
Corpus statistics for the unique tokens and average lengths of the three Java datasets.

Dataset	Hu dataset	LeClair dataset	Husain dataset
Unique tokens in code	20,162	221,716	225,082
Unique tokens in AST	28,264	470,523	380,582
Unique tokens in summary	25,619	96,208	49,028
Avg. tokens in code	120.10	29.58	153.51
Avg. tokens in AST	130.07	52.57	181.39
Avg. tokens in summary	17.76	7.61	11.89

Hyper-parameters. Table 2 shows the corpus statistics of the three Java datasets. We follow Ahmad et al. (2020) and LeClair et al. (2019) to set the vocabulary sizes and maximum input lengths on code and summary sequence. Based on the statistical results, we set the parameters on the Husain dataset. The vocabulary sizes of the code and summary in all datasets are set to be 50,000 and 30,000 respectively. The maximum lengths of code and summary are set to be 150 and 50 in the Hu dataset, 50 and 13 in the LeClair dataset, and 180 and 30 in the Husain dataset. Similarly, we build a 50,000-word list for AST nodes, and limit its sequence length to 200, 100 and 240 respectively. We use a single-layered GRU with 512 dimensions of the hidden states and 512-dimensional word embeddings. The head number of GAT is 4 and each GAL has a dimension of 128 on its output, the concatenation of which are sent to another 512-dimension GAL. Model parameters are optimized by Adam (Kingma and Ba, 2015) with the initial learning rate of 0.002. The dropout rate and mini-batch sizes of our model are set to be 0.2 and 32, respectively. We modify the learning rate with StepLR supported by PyTorch, which decays the learning rate for each epoch. We use beam search in the inference process, whose size is set to be 4. We apply gradient clipping to prevent gradients from becoming too large and set the teacher forcing ratio to be 0.5. According to the performance of the validation set, the best model is selected after training 200 epochs on the Hu dataset, 10 epochs on the LeClair dataset and 30 epochs on the Husain Dataset in the experiments. The hyper-parameters of our GSCS model are listed in Table 3. Our implementation is based on PyTorch 1.3.1.[4]

Hardware Configurations. To train our models, we use a server with two Intel Xeon Silver 4216 CPUs, 64 GB RAM and four 2080Ti GPUs.

[2] https://pypi.org/project/javalang/.

[3] https://nltk.org/.

[4] https://pytorch.org/.

Table 3
Hyper-parameters of the GSCS model.

Parameter	Value
Negative input slope of LeakyReLU nonlinearity (α)	0.2
Amount of independent attention mechanisms (K)	4
Dimension of the feature space (F)	512
Output dimension of GAL (F')	128
Out put dimension of an additional GAL (F'', cf. (6))	512
Code vocabulary size	50,000
Node vocabulary size	50,000
Summary vocabulary size	30,000
Dropout rate	0.2
Teacher forcing ratio	0.5
Beam search size	4

5.2. Overall results

We compare the performance of GSCS with five baseline methods, i.e., CODE-NN (Iyer et al., 2016), Dual model (Wei et al., 2019), RNN+ConvGNN (LeClair et al., 2019), Transformer (Ahmad et al., 2020) and Code Transformer (Zügner et al., 2021). CODE-NN is the first approach for code summarization based on neural networks, which encodes code sequence with embedding matrices and applies LSTM to generate summary guided by the attention mechanism. Dual model uses two independent attention-based seq2seq neural networks to pretrain CS and CG, and then trains them jointly by adding a regularization term utilizing the dual constraints to the loss function. RNN+ConvGNN introduces additional 2-layer ConvGNNs to encode the tree structure of the code snippets and uses a dense layer to generate the summary sequence. Transformer implements parallel encoding embedded tokens and optimizes the CS model by introducing the copy mechanism and relative positions. Code Transformer uses relative distances instead of absolute positions in the attention computation and learns jointly from the structure and context of programs while only relying on language-agnostic features. For CODE-NN, we customize the identifier replacement rules for Java, as it was originally designed for SQL and C#. For Dual model, to reduce the time cost we expand the iteration interval of model validation from 500 to 30,000 on the LeClair dataset which contains over 2 million Java methods. For Code Transformer, we change its output from method names to summaries.

The experiment results are summarized in Table 4. We can observe that GSCS outperforms the state-of-the-art techniques on all datasets, with the combination of the semantic encoder and structural encoder guided by a modified attention mechanism. CODE-NN performs the lowest among all the baselines. The result of RNN+ConvGNN shows that there is no obvious advantage when introducing the structure information with ConvGNN. Compared with other baselines, Transformer achieves the best on the Hu dataset while it performs unstably on the LeClair dataset. Code Transformer performs well on the LeClair dataset but performs poorly on the other two. This may be because the Code Transformer is designed to predict a method name based on the function body. To test whether the improvements of our approach over baselines are statistically significant, we apply the Wilcoxon Rank Sum test (WRST) (Wilcoxon, 2010) to compare GSCS with RNN+ConvGNN and Transformer, and the p-values at 95% confidence level are 7.277e−13 and 8.212e−05 respectively, indicating significant improvements. Specifically, on the Hu dataset, the metric scores of GSCS are increasing from 44.58, 26.43, 54.76 to 46.35, 29.19, 54.98 for the Transformer model. On the LeClair dataset, compared to Code Transformer, the metric scores of GSCS are increased from 27.33, 20.69, 45.57 to 29.43, 22.81, 47.32 respectively. On the Husain dataset, GSCS improves three metrics' scores from 15.28, 12.33, 25.85 to 16.01, 12.87, 26.52 compared with RNN+ConvGNN.

To analyze the advantages of GSCS, we select the best two baseline models of code summarization, i.e., RNN+ConvGNN and Transformer, for further comparison. Particularly, we study the impact of code length, summary length, and AST size on the performance of the selected models. Fig. 5 shows the average BLEU-4 scores on GSCS, RNN+ConvGNN and Transformer when varying code length, summary length, and AST sizes (including AST length, depth and width). It shows that, compared with Transformer and GSCS, RNN+ConvGNN has a large performance gap. However, by observing the performance trend in the five figures, we can find that the gap is narrowing with the growth of the code length, the summary lengths, or the AST lengths. In addition, according to the average BLEU-4 scores on the code, summary and AST of different lengths, we observe that GSCS performs considerably better when the source code has a relatively long length. This indicates that RNN-based model may comprehend code snippets better on long sequences than the Transformer architecture. Note that the depth represents the max depth of an AST and the width means the maximum number of children in an AST, both of which measure the complexity of a code snippet. The statistics in Figs. 5(d) and 5(e) show that, in contrast to RNN+ConvGNN and Transformer, GSCS enables code summarization to comprehend the inner representations of Java methods with complex structures better.

5.3. Ablation study

In the ablation experiment, we aim to identify the components which contribute most to the performance of GSCS. To this end, we train four models: RNN is the basic Seq2Seq model that applies a single-layer BiGRU to build a language model on code sequence and generates natural language summaries with GRU guided by the attention mechanism. RNN_EM introduces embedding, which uses the combination of the transformed embedding layer and BiGRU as the semantic encoder output.

Table 5 shows the experimental results. We can see that adding the transformed embedding layer is more effective than applying a single BiGRU when encoding the code tokens. Furthermore, we tested another model which simply encoded code tokens using a single embedding layer (without BiGRU) which achieved lower BLEU-4 scores. These results suggest that the combination of (transformed) embedding layer and BiGRU is the best option, as intuitively they can eliminate the impact on the absolute location information of the code sequence, and alleviate the long dependence problem of RNN.

Based on the basic Seq2Seq model, RNN_GAT adds GAT to build the structural encoder on tokenized AST and combines the semantic encoder with the structural encoder by a shared attention mechanism. The result shows that GAT improves the performance of the code summarization model, indicating the availability of the stacking GALs. Specifically, compared with the Seq2Seq model, the BLEU-4 score increases from 44.39 to 45.37 on the Hu Dataset, from 27.42 to 28.78 on the LeClair Dataset and from 14.97 to 15.68 on the Husain dataset. This also illustrates that the supplement of structure information can optimize the code summarization model. Naturally, GSCS combining the two components further enhances our model's performance.

Fig. 6 records the impact of the introduction of transformed EM and GAT on the performance of the Hu dataset, which demonstrates the average BLEU-4 scores when varying code lengths, summary lengths, AST sizes (including AST depth and width). These figures demonstrate the main improvement with EM and GAT. In particular, RNN, RNN_EM, RNN_GAT and GSCS correspond to the four models in Table 4 respectively. Comparing the yellow and blue solid lines in all five plots, we can see that the full model, GSCS, is clearly superior to the basic Seq2Seq model.

Table 4
Comparison of GSCS with the baselines: CODE-NN, Dual Model, RNN+ConvGNN, Transformer, Code Transformer and Seq2Seq.

Method	Hu dataset			LeClair dataset			Husain dataset		
	BLEU-4	METEOR	ROUGE-L	BLEU-4	METEOR	ROUGE-L	BLEU-4	METEOR	ROUGE-L
CODE-NN	27.60	12.61	41.10	16.20	13.44	26.85	10.77	6.19	17.5
Dual model	42.39	25.77	53.61	22.43	18.81	40.67	13.14	11.39	**32.33**
RNN+ConvGNN	42.24	26.28	50.71	25.24	19.35	42.96	15.28	12.33	25.85
Transformer	44.58	26.43	54.76	24.99	20.64	43.67	12.69	12.74	26.44
Code transformer	39.01	20.70	41.07	27.33	20.69	45.57	15.30	10.58	23.98
Seq2Seq	44.39	27.94	54.37	27.42	21.39	44.88	14.97	12.56	26.43
GSCS	**46.35**	**29.19**	**54.98**	**29.43**	**22.81**	**47.32**	**16.01**	**12.87**	26.52

Table 5
Ablation study on our model. EM represents a transformed embedding layer.

Method	EM	GAT	Hu dataset			LeClair dataset			Husain dataset		
			BLEU-4	METEOR	ROUGE-L	BLEU-4	METEOR	ROUGE-L	BLEU-4	METEOR	ROUGE-L
RNN	–	–	44.39	27.94	54.37	27.42	21.39	44.88	14.97	12.56	26.43
RNN_EM	✓	–	45.37	28.26	54.28	28.78	22.38	46.61	15.68	12.52	26.22
RNN_GAT	–	✓	45.74	28.91	54.57	28.73	22.50	46.80	15.76	12.81	26.25
GSCS	✓	✓	**46.35**	**29.19**	**54.98**	**29.43**	**22.81**	**47.32**	**16.01**	**12.87**	**26.52**

(a) Code length (b) Summary length

(c) AST length (d) AST depth

(e) AST width

Fig. 5. The average BLEU-4 scores (standardized with a range from 0.3 to 0.6) for RNN+ConvGNN, Transformer, and GSCS as a function of code (a), and summary length (b) as well as AST length (c), depth (d), and width (e). We observe that the performance gap narrows as these metrics increase, however, we also observe GSCS consistently performing well at high values.

(a) Code length (b) Summary length

(c) AST length (d) AST depth

(e) AST width

Fig. 6. The average BLEU-4 scores (standardized with a range from 0.3 to 0.6) for RNN, RNN_EM, RNN_GAT, and GSCS as a function of code (a), and summary length (b) as well as AST length (c), depth (d), and width (e).

Figs. 6(a) and 6(c) illustrate that RNN_EM mainly improves the result of short sequences whereas RNN_GAT contributes more to long code snippets, which indicates that the transformed embedding layer in our model mainly improves code summarization on the source code with simple structures. Comparing the gray dotted line with the blue solid line in Figs. 6(d) and 6(e), we find that RNN_GAT advances the accuracy of the summaries better than RNN_EM, especially when the AST is deeper or wider.

6. Threats to validity

Internal Validity. We study the internal threats from dimension setting, data shuffling and data filtering. As discussed in Section 5.2, in the code summarization task, the setting of dimensions has a significant impact on the final results. To eliminate this effect, we conduct several experiments on different dimensions from 256 to 640 with a separation of 128, and find that the improvement tends to fatten out when the dimension reaches a threshold. Thus, when comparing with baselines, we select the results of the corresponding dimension for comparison. Specifically, the dimension size setting in Transformer and Dual model is 512 while RNN+ConvGNN is 256. In addition, shuffle is employed to avoid the impact of data input order on network training. We do so because data in the LeClair dataset are ordered according to projects, and large amounts of Java methods with similar functions or structures gather in a region. Experimental results show that whether shuffling data or not affects the performance of the Seq2Seq model around 2 percent. Another threat originates from data filtering. GSCS requires that the source code can be parsed into ASTs to extract structure information. However, some data in the LeClair dataset fail to generate ASTs through the

javalang library. To address this concern, we cut off these data pairs, which account for less than 1%. We also conduct contrastive experiments with baselines and ablation study based on the filtered dataset, to guarantee consistency.

External Validity. External validity can be illustrated by the generalizability and scalability of an approach. Experiments demonstrate that GSCS is superior to all baselines in terms of multiple evaluation metrics on three different Java datasets. Even if the two datasets are both extracted from the open-source platform, their quality and processing methods are different which, to a great extent, reflects the universality of our method. In addition, our approach is only experimented on Java methods to generate corresponding summaries. However, we believe it can be extended in a rather straightforward manner to other programming languages that can parse into ASTs with merely minor adaptation.

7. Conclusion

In this paper, we presented a novel approach, GSCS, to generate summaries for Java methods, which combines text features with structure information of code snippets. GSCS splits terminal nodes of the AST and treats the tokenized AST as an undirected graph. The tokenized code sequence and split-ASTs are respectively fed to two different encoders, namely, one is composed of an embedding layer and BiGRU, and the other one uses multiple graph attention layers jointed with a BiGRU. We demonstrate that stacking graph attention layers enables the model to extract useful node features, which is beneficial to source code summarization. Experiments over three public datasets show that GSCS outperforms the state-of-the-art approaches by a large margin.

For future work, we plan to extend our approach and experiments to other programming languages. Further studies are required to improve the performance of code summarization based on, for example, an effective fusion of local and global structures of code snippets.

CRediT authorship contribution statement

Yu Zhou: Devised the project and conceived the main conceptual framework, Wrote the manuscript, Provided critical feedback and helped shape the research, Analysis and manuscript, Supervised the project. **Juanjuan Shen:** Devised the project and conceived the main conceptual framework, Carried out the implementation and experiments, Wrote the manuscript, Provided critical feedback and helped shape the research, Analysis and manuscript. **Xiaoqing Zhang:** Carried out the implementation and experiments, Provided critical feedback and helped shape the research, Analysis and manuscript. **Wenhua Yang:** Wrote the manuscript, Provided critical feedback and helped shape the research, Analysis and manuscript. **Tingting Han:** Wrote the manuscript, Provided critical feedback and helped shape the research, Analysis and manuscript. **Taolue Chen:** Devised the project and conceived the main conceptual framework, Wrote the manuscript, Provided critical feedback and helped shape the research, Analysis and manuscript, Supervised the project.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (NSFC, No. 61972197), the Natural Science Foundation of Jiangsu Province, China (No. BK20201292), the Collaborative Innovation Center of Novel Software Technology and Industrialization, China, and the Qing Lan Project, China. T. Chen is partially supported by Birkbeck BEI School Project (EFFECT), China and NSFC, China grant (No. 61872340, No. 62072309).

References

Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K., 2020. A transformer-based approach for source code summarization. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5–10, 2020. pp. 4998–5007.

Allamanis, M., Peng, H., Sutton, C., 2016. A convolutional attention network for extreme summarization of source code. In: Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19–24, 2016. pp. 2091–2100.

Binkley, D.W., 2007. Source code analysis: A road map. In: International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23–25, 2007, Minneapolis, MN, USA. pp. 104–119.

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 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. pp. 1724–1734.

Chung, J., Gülçehre, Ç., Cho, K., Bengio, Y., 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. CoRR arXiv:1412.3555.

Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., Koschke, R., 2009. A systematic survey of program comprehension through dynamic analysis. IEEE Trans. Software Eng. 35 (5), 684–702.

Denkowski, M.J., Lavie, A., 2014. Meteor universal: Language specific translation evaluation for any target language. In: Proceedings of the Ninth Workshop on Statistical Machine Translation, WMT@ACL 2014, June 26–27, 2014, Baltimore, Maryland, USA. pp. 376–380.

Duan, J., Zhao, H., Qin, W., Qiu, M., Liu, M., 2020. News text classification based on MLCNN and BiGRU hybrid neural network. In: 3rd International Conference on Smart BlockChain, SmartBlock 2020, Zhengzhou, China, October 23–25, 2020. pp. 137–142.

Fernandes, P., Allamanis, M., Brockschmidt, M., 2018. Structured neural summarization. CoRR arXiv:1811.01824.

Haiduc, S., Aponte, J., Marcus, A., 2010. Supporting program comprehension with source code summarization. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1–8 May 2010. pp. 223–226.

Han, Y., Liu, M., Jing, W., 2020. Aspect-level drug reviews sentiment analysis based on double BiGRU and knowledge transfer. IEEE Access 8, 21314–21325.

He, D., Xia, Y., Qin, T., Wang, L., Yu, N., Liu, T.-Y., Ma, W.-Y., 2016. Dual learning for machine translation.

Hellendoorn, V.J., Devanbu, P.T., 2017. Are deep neural networks the best choice for modeling source code? In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4–8, 2017. pp. 763–773.

Hill, E., Pollock, L.L., Vijay-Shanker, K., 2009. Automatically capturing source code context of NL-queries for software maintenance and reuse. In: 31st International Conference on Software Engineering, ICSE 2009, May 16–24, 2009, Vancouver, Canada, Proceedings. pp. 232–242.

Hindle, A., Barr, E.T., Gabel, M., Su, Z., Devanbu, P.T., 2016. On the naturalness of software. Commun. ACM 59 (5), 122–131.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2018. Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27–28, 2018. pp. 200–210.

Husain, H., Wu, H., Gazit, T., Allamanis, M., Brockschmidt, M., 2019. Codesearchnet challenge: Evaluating the state of semantic code search. CoRR arXiv:1909.09436.

Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L., 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.

Kingma, D.P., Ba, J., 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. URL http://arxiv.org/abs/1412.6980.

Kipf, T.N., Welling, M., 2017. Semi-supervised classification with graph convolutional networks. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24–26, 2017, Conference Track Proceedings.

LeClair, A., Haque, S., Wu, L., McMillan, C., 2020. Improved code summarization via a graph neural network. In: ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13–15, 2020. pp. 184–195.

LeClair, A., Jiang, S., McMillan, C., 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. pp. 795–806.

Leclair, A., Mcmillan, C., 2019. Recommendations for datasets for source code summarization.

Lin, C.-Y., 2004. ROUGE: A package for automatic evaluation of summaries. In: Text Summarization Branches Out.

Luong, T., Pham, H., Manning, C.D., 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. pp. 1412–1421.

Lv, J., Du, J., Zhou, N., Xue, Z., 2020. BERT-BIGRU-CRF: a novel entity relationship extraction model. In: 2020 IEEE International Conference on Knowledge Graph, ICKG 2020, Online, August 9–11, 2020. pp. 157–164.

Maletic, J.I., Marcus, A., 2001. Supporting program comprehension using semantic and structural information. In: Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on.

Ottenstein, K.J., Ottenstein, L.M., 1984. The program dependence graph in a software development environment. In: Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, USA, April 23–25, 1984. pp. 177–184.

Papineni, K., Roukos, S., Ward, T., Zhu, W.-J., 2002. Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics. Association for Computational Linguistics, pp. 311–318.

Schuster, M., Paliwal, K.K., 1997. Bidirectional recurrent neural networks. IEEE Trans. Signal Process. 45 (11), 2673–2681.

Shido, Y., Kobayashi, Y., Yamamoto, A., Miyamoto, A., Matsumura, T., 2019. Automatic source code summarization with extended tree-LSTM. In: International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14–19, 2019. pp. 1–8.

Song, X., Sun, H., Wang, X., Yan, J., 2019. A survey of automatic generation of source code comments: Algorithms and techniques. IEEE Access 7, 111411–111428.

Sridhara, G., Hill, E., Muppaneni, D., Pollock, L.L., Vijay-Shanker, K., 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. pp. 43–52.

Sridhara, G., Pollock, L.L., Vijay-Shanker, K., 2011. Generating parameter comments and integrating with method summaries. In: The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, on, Canada, June 22–24, 2011. pp. 71–80.

Stapleton, S., Gambhir, Y., LeClair, A., Eberhart, Z., Weimer, W., Leach, K., Huang, Y., 2020. A human study of comprehension and code summarization. In: ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13–15, 2020. pp. 2–13.

Sutskever, I., Vinyals, O., Le, Q.V., 2014. Sequence to sequence learning with neural networks. In: NIPS.

Takang, A.A., Grubb, P.A., Macredie, R.D., 1996. The effects of comments and identifier names on program comprehensibility: an experimental investigation. J. Program. Lang. 4 (3), 143–167.

Tenny, T., 1988. Program readability: Procedures versus comments. IEEE Trans. Software Eng. 14 (9), 1271–1279.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I., 2017. Attention is all you need. In: Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4–9, 2017, Long Beach, CA, USA. pp. 5998–6008.

Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., Bengio, Y., 2017. Graph attention networks. CoRR arXiv:1710.10903.

Wei, B., Li, G., Xia, X., Fu, Z., Jin, Z., 2019. Code generation as a dual task of code summarization. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8–14, 2019, Vancouver, BC, Canada. pp. 6559–6569.

Wilcoxon, F., 2010. Some rapid approximate statistical procedures. Ann. New York Acad. Sci. 52 (The Place of Statistical Methods in Biological and Chemical Experimentation), 808–814.

Wong, E., Yang, J., Tan, L., 2013. Autocomment: Mining question and answer sites for automatic comment generation. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11–15, 2013. pp. 562–567.

Woodfield, S.N., Dunsmore, H.E., Shen, V.Y., 1981. The effect of modularization and comments on program comprehension. In: ICSE '81. IEEE Press, pp. 215–223.

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A.E., Li, S., 2018. Measuring program comprehension: a large-scale field study with professionals. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. p. 584.

Zhang, J., Wang, X., Zhang, H., Sun, H., Liu, X., 2020. Retrieval-based neural source code summarization. pp. 1385–1397. http://dx.doi.org/10.1145/3377811.3380383.

Zhou, Y., Yan, X., Yang, W., Chen, T., Huang, Z., 2019. Augmenting java method comments generation with context information based on neural networks. J. Syst. Softw. 156 (Oct.), 328–340.

Zügner, D., Kirschstein, T., Catasta, M., Leskovec, J., Günnemann, S., 2021. Language-agnostic representation learning of source code from structure and context. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3–7, 2021.

Yu Zhou is currently a full professor in the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics (NUAA). He received his B.Sc. degree in 2004 and Ph.D. degree in 2009, both in Computer Science from Nanjing University China. Before joining NUAA in 2011, he conducted PostDoc research on software engineering at Politechnico di Milano, Italy. From 2015–2016, he visited the SEAL lab at University of Zurich Switzerland, where he is also an adjunct researcher. His research interests mainly include software evolution analysis, mining software repositories, software architecture, and reliability analysis. He has been supported by several national research programs in China.

Juanjuan Shen received her B.Sc. degree in Computer Science, from Nanjing University of Finance and Economics, China. She is currently a M.Sc. student in the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics. Her research interests include software evolution analysis, artificial intelligence, and mining software repositories.

Xiaoqing Zhang received her B.Sc. degree in Software Engineering, from Nanjing University of Information, Science and Technology, China. She is currently a M.Sc. student in the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics. Her research interests include deep learning, and mining software repositories.

Wenhua Yang is currently an assistant professor in the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics. He obtained his Ph.D. degree in computer science and technology from Nanjing University in 2017. His research interests mainly include self-adaptive & cyber physical systems, empirical software engineering.

Tingting Han received the Bachelor and Master degrees from Nanjing University, China, both in Computer Science. She joined a bilateral Ph.D. program and acquired a Ph.D. degree from the RWTH Aachen University (DE) and University of Twente (NL). After that she was a research assistant at the University of Oxford and currently she is a Senior Lecturer at the Department of Computer Science, Birkbeck, University of London. Her research interests are mainly in formal verification and synthesis, program analysis, as well as stochastic modeling and machine learning in various applications, such as biometric verification, autonomous driving, etc.

Taolue Chen received the Bachelor and Master degrees from Nanjing University, China, both in Computer Science. He was a junior researcher (OiO) at the CWI and acquired the Ph.D. degree from the Vrije Universiteit Amsterdam, The Netherlands. He is currently a Lecturer at the Department of Computer Science, Birkbeck, University of London. He was a research assistant at the University of Oxford. His research interests are mainly in software engineering including formal verification and synthesis, program analysis, as well as stochastic modeling and machine learning in software engineering. He has co-authored over 100 peer-reviewed journal and conference papers, and has served as a technical program committee member for various international conferences.