The Journal of Systems & Software 203 (2023) 111746

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Learning a holistic and comprehensive code representation for code summarization☆

Kaiyuan Yang [a], Junfeng Wang [a,*], Zihua Song [b]

[a] College of Computer Science, Sichuan University, Chengdu 610065, China
[b] School of Cyber Science and Engineering, Sichuan University, Chengdu 610207, China

ARTICLE INFO

Article history:
Received 3 November 2022
Received in revised form 14 March 2023
Accepted 4 May 2023
Available online 10 May 2023

Keywords:
Code summarization
API
Deep learning
Program comprehension

ABSTRACT

Code summarization is the task of describing the function of code snippets in natural language, which benefits program comprehension and boosts software productivity. Despite lots of effort made by previous studies, existing models are not comprehensive enough to represent code, and yet the literature does not consider to model source code with API usage from a holistic perspective. To this end, this paper proposes a novel multi-modal code summarization approach called HCCS (Learning a **H**olistic and **C**omprehensive code representation for **C**ode **S**ummarization). We first design a neural network based on the graph attention mechanism to encode API Context Graph (ACG), which highlights holistic information of source code. Then, a multi-modal framework with a tree encoder for Abstract Syntax Tree (AST) and a code encoder for code tokens is incorporated to learn a more comprehensive code representation. Afterwards, we propose a fusing layer to integrate the encodings, which are then passed to a joint-decoder to generate summaries. The experimental results show that HCCS achieves better performance than the state of the arts (i.e., HCCS scores 9.5% higher in terms of BLEU metric and 11.46% in terms of BERTScore). As a result, HCCS is an effective approach to generate high-quality summaries.

1. Introduction

Software projects grow rapidly in scale and complexity, which impedes software maintenance. Software comprehension is a prerequisite for code reuse and software maintenance (Levy and Feitelson, 2021). According to the statistics, developers spend 58% of their time on program understanding in the entire software development life cycle (Xia et al., 2017). As such, code comments are crucial for learning the intent of code snippets. For instance, a comment such as "Returns the maximum value of two numbers" (shown in Fig. 1) assists developers in digesting this code snippet, which effectively promotes the productivity of software development. However, manually maintaining comments for code snippets is labor-intensive and time-consuming (Nazar et al., 2016a). Comments often miss and tend to be outdated as the software projects iterate. Hence, automatically generating code comments referred to as "code summarization" has raised lots of interest, and becomes a vital problem in the software engineering area (Mastropaolo et al., 2021).

Previous studies about code summarization mainly depend on manually-template (Mou et al., 2014) or information retrieval (IR) (Wei et al., 2020). Recently, data-driven strategies based on neural networks are favored with the open-source platform accumulating numerous codes (Nazar et al., 2016b; Iyer et al., 2016). Most of the state of the arts extend an encoder–decoder framework (Sutskever et al., 2014; Cho et al., 2014) and rely on massive code–summary pairs to perform training. In this framework, the encoder extracts various information from source code and then yields an intermediate representation for decoding.

```
0:    // Returns the maximum value of two numbers
1:    public static Number max(Number a,Number b){
2:        if (isFloatingPoint(a) || isFloatingPoint(b)){
3:            return Math.max(a.doubleValue(),b.doubleValue());
4:        }
5:        else{
6:            return Math.max(a.longValue(),b.longValue());
7:        }
8:    }
```

Fig. 1. The example of a code snippet.

The main difference of current encoder-to-decoder approaches is how to learn code representations. Previous studies have used various modalities to represent code, such as sequence, tree, and graph, each focusing on a different aspect of code information (LeClair et al., 2020, 2021). Source code is first viewed as

☆ Editor: Burak Turhan.
* Corresponding author.
E-mail address: wangjf@scu.edu.cn (J. Wang).

https://doi.org/10.1016/j.jss.2023.111746
0164-1212/

pure text, then feeding an encoder–decoder model to generate summaries (Iyer et al., 2016). After that, some works leverage Application Program Interface (API) knowledge into the model (Hu et al., 2018a). Indeed, code is highly structured and follows certain programming syntax rules which is different from natural language text. In light of this, diverse heterogeneous forms of the Abstract Syntax Tree (AST) are utilized to absorb code structure and grammar information (Hu et al., 2018b; Shido et al., 2019; Tang et al., 2022). Meanwhile, the literature begins to recognize that graph neural networks (GNN) can tease out the deep semantics of source code in a graph form (LeClair et al., 2020; Zhou et al., 2022). Additionally, pre-train models could support code representation (Wan et al., 2022), which serves as a basis for downstream tasks such as code completion (Wang and Li, 2021), code search (Wan et al., 2019), code vulnerability detection, and code summarization (LeClair et al., 2020). Recently, code representations tend to become more complex. The works (Yang et al., 2021; LeClair et al., 2021; Haque et al., 2020) which employ multiple encoders have been proven effective to learn a more comprehensive code representation. Hence, multi-modal approaches are showing promise and continue to advance the state of the arts.

Despite lots of effort made by prior studies, the following challenges have been not well resolved in this regard:

Challenge 1. Existing approaches lack a holistic view to model source code with API usage. Previous studies (Hu et al., 2018a; Shahbazi et al., 2021) have acknowledged that API usage knowledge correlates highly with the functionality of the source code. Nevertheless, this vital knowledge source might not be fully exploited while treating code as pure text (Iyer et al., 2016) or extracting plain API sequences merely (Hu et al., 2018a). Besides, limited to the sequence's maximum length, the models have natural inferiority in capturing long-range dependencies between correlated but far-away API usage (Yang et al., 2021; Li et al., 2021). The API calls in the code snippets are often accompanied with control units, which include holistic reasoning of the whole code structure (Chen et al., 2021). Therefore, it can promote the model's ability to characterize code semantics if we model source code with API usage from a holistic view.

Challenge 2. Most existing models are not competent enough to comprehensively comprehend code semantics. The diverse information sources challenge the existing models to learn a comprehensive code representation. Indeed, the function of a program often manifests in different aspects of the code information, including nonstructural features (i.e., code tokens and APIs) and other structural features (i.e., syntax structure and control flow) (Zhou et al., 2020; Jiang et al., 2022; Cheng et al., 2021). Unfortunately, most of the encoders just receive shallow textual code features and it is a struggle for the encoder to capture various types of information from source code (LeClair et al., 2021). An alternative way is to ensemble multiple encoders. Yet, how to integrate features in the ensemble models remains a fundamental problem. The naive fusion ways to integrate three features, namely, by concatenating or summing might miss potential clues.

In this paper, we propose a novel approach for code summarization called HCCS (Learning a **H**olistic and **C**omprehensive code representation for **C**ode **S**ummarization). To tackle the above challenges, we propose the following solutions with HCCS:

Solution 1. We first represent source codes by API usage in a control flow graph called API Context Graph (ACG). ACG can be viewed as the skeleton of the programs, which provides a joint overview of semantic and structural information, as shown in Fig. 2. In particular, we take a step forward and design a neural network based on the graph attention mechanism (graph encoder) to encode ACG, aiming to learn a holistic code representation. We take two technologies to increase the expressive capability of the attention modules: (1) we compute the inner product of the features between the anchor node and neighbor nodes instead of linear combinations to extract high-level information. (2) The graph attention module incorporates a novel structural encoding technique which injects local and global structure information.

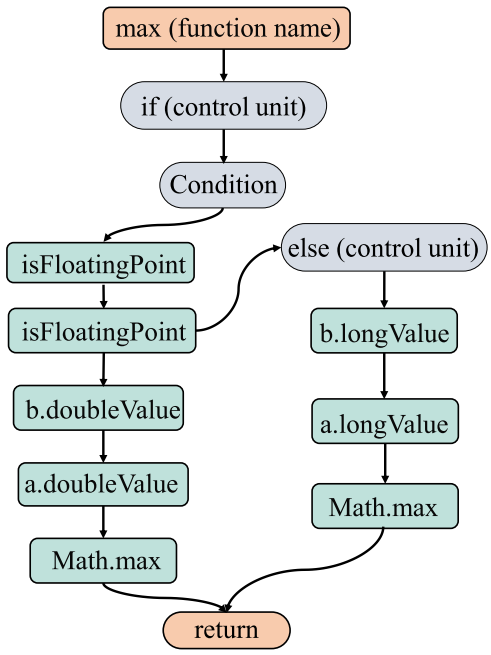

Fig. 2. The example of an API Context Graph.

Solution 2. To address Challenge 2, we exploit a multi-modal architecture network to learn a comprehensive code representation. This challenge motivates us to extract more code features from its multiple views, such as ACGs, ASTs and code tokens. ACGs characterize holistic code information (integrating API usage and control-flows); ASTs could depict syntax syntactic of a program and code tokens directly preserve shallow textual information. Both nonstructural and structural features are fed to our network. Hence, our network extracts condensed and informative features to learn a comprehensive code representation. Apart from a graph encoder, we not only incorporate a tree encoder to encode the ASTs, but also a token encoder for code tokens. We argue for a complementary relationship among these encoders, which can facilitate a deep semantic understanding of the source code. To integrate three encodings, we design a novel fusing layer, which makes significant progress on the results.

The main idea of HCCS is inspired by the following practice. To digest code semantics, software participants first scan the overall program (function name, API calls, control flow unit, etc.), then gather more comprehensive code information (syntax, identifiers semantics, etc.). The combined use of diverse code information to learn a holistic and comprehensive code representation is the main characteristic of our approach. To our best knowledge, we use ACG to model source code from a holistic perspective for the first time, which enhances code representation. To extract high-level code information from ACGs, we furthermore design a graph attention module (qk-GAT) with structural encoding, which facilitates the final performance. We proposed a novel fusion way to integrate three encodings and conduct a series of ablation studies to identify the contributions of different components.

Our approach is trained and evaluated on two datasets (i.e, JHD Hu et al., 2018a, PFD Barone and Sennrich, 2017) during

evaluation. Previous works (Hu et al., 2022; Haque et al., 2022; Roy et al., 2021) have pointed to the need for specialized automated evaluation metrics, which correlates more closely to human judgment. Therefore, apart from traditional evaluation metrics (such as BLEU Papineni et al., 2001, METEOR Denkowski and Lavie, 2014, and ROUGE Lin, 2004), we introduce another robust metric called BERTScore (Zhang* et al., 2020) to serve as the measurement. The experimental results clearly indicate that HCCS outperforms the baselines. For instance, HCCS makes a 9.5% improvement over the baselines in terms of BLEU metric and the BERTScore achieves 11.46%. As such, HCCS is an effective approach to generate high-quality summaries.

Contributions. The contributions of this paper are stated below:

- For effective code summarization, we model source code with API usage from a holistic perspective to feed an improved graph attention network, thus enhancing code representation.
- In this paper, we propose a novel multi-modal code summarization method, HCCS, which could fuse features from multiple encoders to generate high-quality summaries.
- We conduct empirical experiments to verify the effectiveness of our proposed approach. This paper additionally introduces another robust evaluation metric to serve as complementary to traditional metrics (i.e., BLEU, METEOR, and ROUGE).

Organization. The reminder of our paper is structured as follows: Section 2 introduces the related work. In Section 3, we present the background concerning fundamental concepts. Section 4 elaborates technical details of our approach. We report our experimental studies and results in Section 5. Finally, we conclude this paper in Section 6.

2. Related work

2.1. Deep code representation

How to represent source codes affects its downstream work such as code completion, code search, code vulnerability detection, and code summarization. Deep neural networks have illustrated considerable promise in code representation recently.

Wang and Li (2021) model a flattened token sequence of a partial AST in a pre-order way, which has proven to advance the code completion task. Sui et al. (2020) present Flow2Vec, a value-flow-based code embedding approach which aims to capture interprocedural program dependence. They argue for a context-sensitive code representation by preserving calling context information of a program, which could boost the performance of code summarization task. In the work (Wan et al., 2019), unstructured and structured features of source code are simultaneously exploited to develop a comprehensive multi-modal representation for effective semantic code search. They adopted multi-modal encoders, with one LSTM for code tokens, a Tree-LSTM for the AST, and a Gated Graph Neural Network (GGNN) for the CFG. Cheng et al. (2021) also embed both the unstructured (code tokens) and structured (control-and data dependence) code information of a program in a compact low-dimensional code representation for code vulnerability detection. For effective API recommendation, Chen et al. (2021) proposed a novel code representation called API Context Graph Network, which combines API usage and textual information as a whole. PTMs (Hadi et al., 2022) introduced pre-trained models to assist in API sequence embedding, which contribute to a significant boost for the API learning task. Recently, Wan et al. (2022) conduct a thorough structural analysis of pre-train models for source code and suggest incorporating code syntax structure for better code representations.

2.2. Code summarization

Deep learning models have shown remarkable success in code summarization, which are capable of representing complex features. This paper focuses on whether code representations can comprehensively capture code semantics. As such, we list the related work from the aspect of code representation modalities such as sequence, API, AST, GNN, and ensemble models.

Iyer et al. (2016) first applied the Neural Machine Translation (NMT) framework to code summarization. The proposed model CODE-NN treats the source codes as plain sequences, which are transported to a long short-term memory network (LSTM) to generate comments. After that, Hu et al. (2018a) transferred API sequence knowledge to an encoder–decoder model. They argue that the API sequence information directly affects the code functions. To accommodate extra API information, the work (Shahbazi et al., 2021) also added the related API documents to the encoder. However, the author states that this approach fails to facilitate the results if there are a large number of API calls in the code snippet.

To explore the tree structure of source code, Shido et al. (2019) proposed an extension of Tree-LSTM models. Tree-LSTM can retain the original structure of the Abstract Syntax Tree (AST), which can better capture the structural information. Hu et al. (2018b) proposed the DeepCom, which learns latent code representations fusing structural and semantic information from linearized ASTs. To obtain unambiguous AST sequences, they propose a novel Structure-Based Traversal (SBT) method to traverse the AST.

LeClair et al. (2020) proposed a novel neural model architecture that leverages code sequences and AST. In this work, a convGNN is utilized to encode ASTs. Different weights should be assigned while aggregating information from neighbor nodes. To this end, Zhou et al. (2022) proposed GSCS, which utilizes a graph attention network (GAT) to represent ASTs.

LeClair et al. (2021) believe that code representations directly affect the results of code summarization methods. As such, they combine different code representations to explore how to integrate various baseline methods to promote results. Zhou et al. (2020) combined a code sequence encoder and an AST-based encoder. In this work, they adopted a switch network which learns adaptive weight vectors to combine different vector representations from two encoders. Based on similar ideas, Yang et al. (2021) proposed a multi-modal code summarization approach called MMTrans. Specifically, MMTrans learns the code representation from two heterogeneous forms of ASTs, Namely, SBT sequences and graphs. Two encoders aim to extract global and local semantic information, respectively. Wang et al. (2022a) employed GAT and code pre-trained models to learn code mixture representations. They extract control-flow-related specific statements from code snippets, fusing them with the AST to learn more latent information from source code.

In addition, Wan et al. (2018) incorporate AST with a Tree-LSTM network as well as sequential tokens with an LSTM network into a deep reinforcement learning (DRL) framework. Wang et al. (2022b) extend this work and inject hybrid representations (including code sequences, type-augmented ASTs and program control flows) to the model. Specifically, this approach assigns weights to tokens and statements, which could reflect the hierarchical code structure.

3. Background

3.1. Encoder–decoder framework

NMT is capable of translating a source sentence to a target based on deep learning (Sutskever et al., 2014). Previous studies

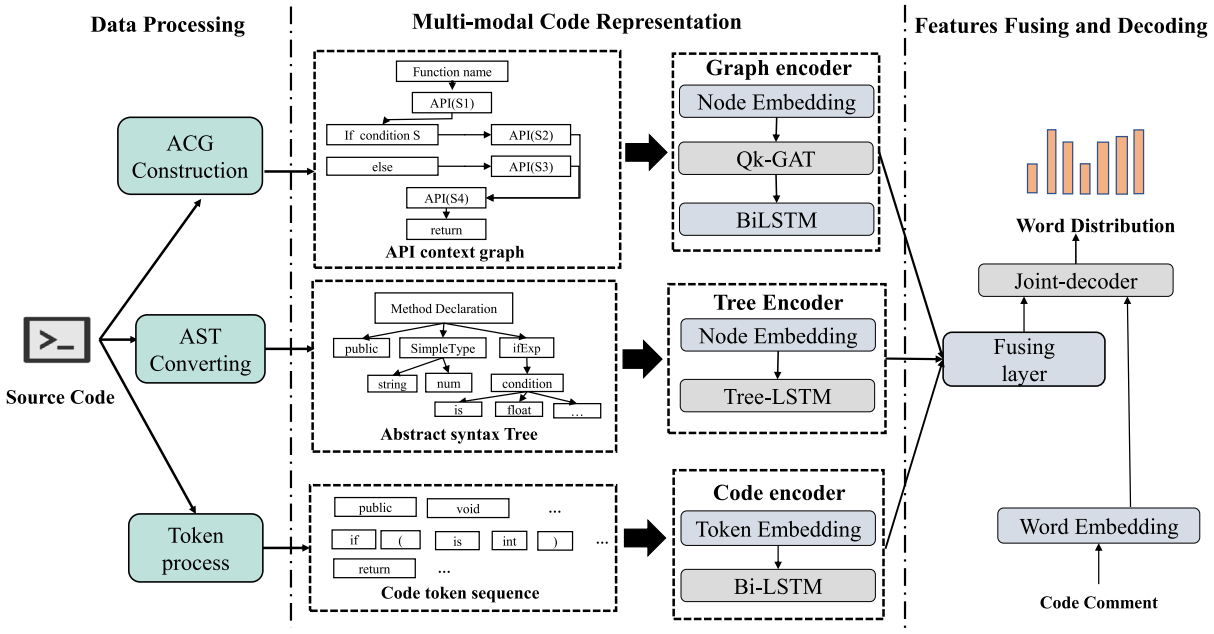

Fig. 3. The overview of HCCS.

have applied NMT framework to the code summarization task, in which we translate the source code to a natural language description (Zhou et al., 2020). Apart from an encoder and a decoder, this framework also tends to adopt the attention mechanism. **Encoder.** The encoder converts the input sequence into a continuous contextual representation. **Attention.** Attention mechanism can assist the encoder in selecting different parts from input sequence. It yields a contextual vector from the encodings which drives the decoding. **Decoder.** The decoder composes each target token from the contexts at each time step, which is usually performed by unidirectional Recurrent Neural Networks.

3.2. Graph attention network

Graph attention networks (Veličković et al., 2017) is an effective attention-based model which successfully adapts attention mechanisms to graph-structured data. The authors presented the use of GATs by stacking layers in which nodes could integrate information from their neighborhoods. The main idea of a GAT is to specify different weights to graph nodes, which distinguishes each neighbor node's contribution to the anchor node. Furthermore, Pan et al. (2022) proposed PLAM, a flexible attention-based module, which is a generalization of GAT with the query–key mechanism. In this work, they project several graph measurements to the feature space called structural encoding, which could inject local and global information.

3.3. Tree-LSTM

Tree-LSTM (Tai et al., 2015) reflects superior performance in handling long-term dependencies for tree-structured input (i.e, AST). Tree-LSTM unit contains multiple forget gates while a standard LSTM unit only has one. As such, Tree-LSTM updates the state of current node from multiple child units. The work (Shido et al., 2019) has applied N-Ary Tree-LSTM to encode ASTs. According to the work (Tai et al., 2015), the N-ary Tree-LSTM unit can be updated as follows:

$$
\begin{aligned}
i &= \sigma\left(W_i x + \textstyle\sum_{l=1}^{L} U_{il} hl + b_i\right) \\
f_l &= \sigma\left(W_f x + U_{fl} hl + b_f\right), \quad l = 1, 2, \ldots, L \\
o &= \sigma\left(W_o x + \textstyle\sum_{l=1}^{L} U_{ol} hl + b_o\right) \\
u &= \tanh\left(W_u x + \textstyle\sum_{l=1}^{L} U_{ul} hl + b_u\right) \\
c &= i \odot u + \textstyle\sum_{l=1}^{L} f_l \odot c_l \\
h &= o \odot \tanh(c)
\end{aligned}
\tag{1}
$$

Here, i, f_l, and o represent the input gate, the forget gate for the lth child and the output gate, σ denotes the sigmoid function, and $\odot$ means elementwise multiplication. $W, U,$ *and* b are trainable weight matrices and bias vectors. The Tree-LSTM recursively computes the embedding sequence from the leaf to the root nodes. Naturally, the output of the root node can be treated as the final state.

4. The proposed approach

4.1. Architecture

Fig. 3 depicts the overall architecture of the proposed approach called HCCS, which is a typical encoder–decoder based code summarization approach, consisting of three parts:

Data processing. Given a code snippet, HCCS first spends some effort on data processing to represent it via three modalities such as ACG, AST, and code tokens. We systematically construct ACG, which can represent overall program logic. Then, we convert source code to AST and split code text into code (sub)tokens, which characterize syntactic structure and textual features of a program.

Multi-modal representation. We adopted a multi-modal strategy in the encoder–decoder framework, with a graph encoder for ACG, a tree encoder for AST, and a token encoder for code tokens. The purpose of ensemble encoders is to learn a holistic and comprehensive code representation.

Features fusion and decoding. We integrate the intermediate features with a novel fusing layer, which are then fed to a joint-decoder to generate summaries.

We will elaborate technical details of each part in the following sections.

4.2. Data processing

ACG construction. Given a code snippet, HCCS first builds ACG from source code systematically. We create nodes and edges after scanning through each statement of source code. What is new is that ACG focuses on statements related to API calls and control units (e.g. if, while, for, switch, etc.). We take the code snippet shown in Fig. 1 as an example, as described below:

- **step 1:** The function name "max" is extracted to serve as the initial node of ACG, which preserves the core semantics of the program. Then, the statements containing a return statement are mapped to a node as the end of the program.
- **step 2:** We traverse code statements to create nodes in ACG.
- **step 2.1:** If the statement acts as an expression including some control unit, HCCS creates a control unit node according to its type, as well as several other nodes (i.e., condition and body nods) together with edges connecting them. For example, when it comes to an "if" statement as shown in Fig. 2, Three nodes such as an "If" node, and a "Condition" node are established. Then HCCS introduces two edges, one from the If node to the Condition node, and the other from the Condition node to the body statements.
- **step 2.2:** If the statement contains a method call, an ACG node assigned the value of the called method name should be constructed. For example, we create a node with the method name "b.doublevalue". It is worth noting that if the parameters of method call are also method calls, HCCS first creates nodes for the parameters in order from right to left.
- **step 3:** Finally, we systematically analyze the control dependencies between the nodes, then add the corresponding edges. Take the "if" control unit of Fig. 2 as an example again. We introduce an edge between the "Condition" node and the "Else" node owing to condition branching. Afterwards, another edge is added from the If node to the "Condition" node for sequential execution.

AST converting. HCCS parses source code in the dataset JHD into AST by Javalang[1] and filters out those failed to convert. For the Python dataset (PFD), we use ast[2] lib. After obtaining the original AST, we split partial leaf nodes with compound identifiers, which benefits the tree encoder by learning more explicit information. Namely, we not only tokenize the nodes into atomic tokens according to their name styles (camelCase, snake_case), but also preserve the tree structure. Based on these single tokens and the type field of AST, HCCS builds an extensive vocabulary, prepared for word embedding.

Tokens preprocess. HCCS also needs to extract the lexical information of source code. A simple way is to treat it as a code tokens. We first obtain raw tokens by tokenizing code with a lexer and then split the identifiers into subtokens, similar to deal with AST nodes. To keep a reasonable vocabulary, HCCS replaces special symbols (Non-integer, MD5, hash values, etc.) in code snippets with an anonymous token <Num>. Note that the word out of the vocabulary will be replaced with the token <Unknown>. Practically, the token encoder and tree encode share the same vocabulary because the words in code tokens are always somewhere in AST nodes.

[1] https://github.com/c2nes/javalang

[2] https://docs.python.org/2/library/ast.html

4.3. Multi-modal code representation

Graph encoder. The graph encoder converts ACGs to the contextual features, consisting of a node embedding layer, a graph attention module, and a Bi-directional LSTM (BiLSTM) network. Each ACG node with the API name or a control unit is embedded into a vector via a node embedding layer, then treated as the node's initial state in the attention module. Here, we exploit a tokenizer to split API names into slices, averaging their corresponding embeddings to get the embedding vector. As such, the node embedding layer could simultaneously leverage APIs' textual and sequences knowledge into the model (Hadi et al., 2022). We store nodes' embeddings into a matrix, referred to as $I^{ACG} = \{I_1, I_2, \ldots, I_n\}$, where the parameter n represents the number of ACG nodes. Afterwards, the nodes embeddings are projected into an intermediate vector via a graph attention module (qk-GAT). The detailed calculating process describes below:

Similar to the traditional self-attention mechanism, we first transform the input into two matrices, which indicate a query space and a key space, respectively. Formally, the query matrix and the key matrix are calculated below.

$$f\left(I^{(ACG)}\right) = I^{(ACG)}W_q, g\left(I^{(ACG)}\right) = I^{(ACG)}W_k \tag{2}$$

Afterwards, we apply dot product between the query vectors and key vectors to acquire the attention matrix α_{ij}. Additionally, we also plug a structural encoding technique (Pan et al., 2022) into the attention module. Formally,

$$\alpha_{ij} = \begin{cases} \text{softmax}_j\left(s_{ij}\right) & \text{if } \widetilde{B}_{ij} = 1 \\ 0, & \text{otherwise,} \end{cases} \tag{3}$$

$$s_{ij} = f\left(I_i\right) \odot g\left(I_j\right)^T + c(i) \odot g\left(I_j\right)^T \tag{4}$$

with $\widetilde{B_{ij}} = B + E_N$, E_N being an Elemental matrix and $c(i)$ projecting several graph measurements into the feature space. Specifically, α_{ij} serves as the attention coefficient for j_{th} node when synthesizing the i_{th} node. The output of the i_{th} node via an attention module is calculated as follows:

$$Out_i^{ATT} = \text{ReLU}\left(\left(\sum_{j\in\{i\cup\mathcal{N}(i)\}} \alpha_{ij}I_j\right)W_v\right) \tag{5}$$

Here, W_v is a learnable matrix, and afterwards all the symbols with W_* refer to learnable network parameters without a special statement. Apart from a query and a key matrix defined in the standard self-attention mechanism, $I_{ACG}W$ can be viewed as the value matrix. Intuitively, the graph attention module produces a weighted summation for each node. Then, we store the final node representations in another matrix $Out^{ATT} = \{Out_1, Out_2, \ldots, Out_n\}$. Fig. 4 depicts a demonstration of graph attention module.

Finally, we adopt another BiLSTM network to hook the graph attention module, which leverages global information into models. The output of the graph encoder can be formulated as below:

$$Out_i^{ACG}, h^{ACG} = \text{BiLSTM}(Out_{i-1}^{ACG}, Out_i^{ATT}) \tag{6}$$

Tree encoder. The tree encoder employs a Tree-LSTM network to encode ASTs, which has proven effectiveness in the works (Wan et al., 2018; Shido et al., 2019). Based on atomic tokens of AST nodes, we can obtain initial tree-structure embeddings I^{AST}. The node values are split into atomic tokens, then passed into the node embedding layer. Aiming at the split AST, Tree-LSTM could effectively handle this tree-structure data to yield intermediate representations. It recursively updates the representation of each node from the bottom. Fig. 5 shows the

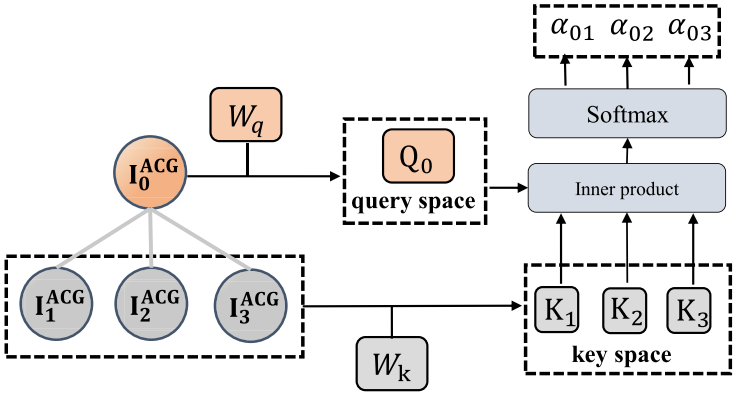

Fig. 4. A demonstration of graph attention module.

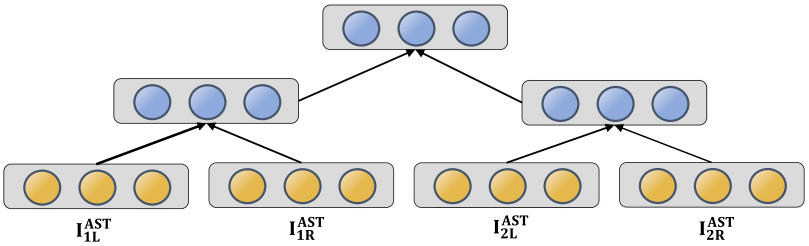

Fig. 5. An illustration of the tree-LSTM network.

structure of Tree-LSTM. Eq. (7) summarizes the overall of the tree encoder formally.

$$Out_i^{AST}, h^{AST} = \text{Tree-LSTM}(I_{iL}^{AST}, I_{iR}^{AST}) \tag{7}$$

Code encoder. The code encoder treats the code (sub)tokens as input and adopts an RNN model to produce tokens features. Similarly, (sub)tokens are first converted to a vector I^{TOKEN} via a learnable embedding layer. Then we employ a BiLSTM network to encode the vector. This process can be formulated as:

$$Out_i^{TOKEN}, h^{TOKEN} = \text{BiLSTM}(Out_{i-1}^{TOKEN}, I_i^{TOKEN}) \tag{8}$$

4.4. Features fusion and decoding

Features fusion layer. After obtaining code encodings from the aspect of each modality, the fusion layer integrates the aforementioned features into a single joint vector. Since the decoder tends to notice different parts of input sequences, the self-attention mechanism which can assign different weights to the sequence is introduced to guide decoding. Given the intermediate features Out^{ACG}, we compute the distinct graph context vector by the following equation:

$$C_t^{ACG} = \sum_{j=1}^{n} \beta_{tj}^{ACG} Out_j^{ACG} \tag{9}$$

where β_{tj} is another attention matrix, representing the weight of C_t on Out_j (Bahdanau et al., 2016). Similarly, we could obtain another two context vectors C_t^{AST} and C_t^{TOKEN} according to Eq. (9).

The key insight into the fusing layer is how to combine the three contexts dynamically. In previous works (Wan et al., 2019; Hu et al., 2018a), the contexts can be simply concatenated or summed by linear transformation when synthesizing multiple features. Here, we integrate three contexts as the following steps. Given the code and tree context first, we compute a weighted average C^{TC} (Eq. (10)). The probability vector $\hat{p}$ defines the contribution of two contexts during each decoding time step (Eq. (11)), which is dependent on the final state (h^{TOKEN}) of the code encoder.

$$C_t^{TC} = \tanh(\hat{p} C_t^{AST} + (1 - \hat{p}) C_t^{TOKEN} \tag{10}$$

$$\hat{p} = \delta\left(W_p h^{TOKEN} + b\right) \tag{11}$$

As mentioned in Section 4.2 The graph context is modeled from a holistic perspective, leading to its feature space varying from the other two vectors. In light of this, we directly concatenate the graph context and the intermediate fusing context C_t^{TC}, which directly feeds the decoder high-level code information. Namely, $C_t^{FUSE} = [C_t^{ACG}; C_t^{TC}]$.

Joint-decoder. The decoder undertakes the task to generate the target sequence $Y = \{Y_1, Y_2, \ldots, Y_T\}$. The initial state of the decoder is an average of the code encoder and the tree encoder' backward final state, namely $h^{INIT} = \frac{1}{2}(h^{AST} + h^{TOKEN})$. During the teach-forcing train stage, the embedded reference summaries S_t are joint with the condensed code features C_t^{FUSE}, which are passed to the joint-decoder (LSTM). The following equation presents this process formally.

$$C_t = \text{LSTM}([C_t^{FUSE}; S_t)], h^{INIT}) \tag{12}$$

$$\hat{Y} = \text{softmax}\left(W_s \tanh\left(W_t\left[C_t; S_t\right]\right)\right) \tag{13}$$

According to the probability distribution $\hat{Y}$, the token with the highest probability in the target vocabulary is selected.

5. Evaluation

This section provides a detailed description of the experimental setup and comprehensive results to evaluate the performance of HCCS. For reproducibility, the program is now available in a repository.[3]

[3] https://github.com/codeMining-tech/codeSum-HCSS.

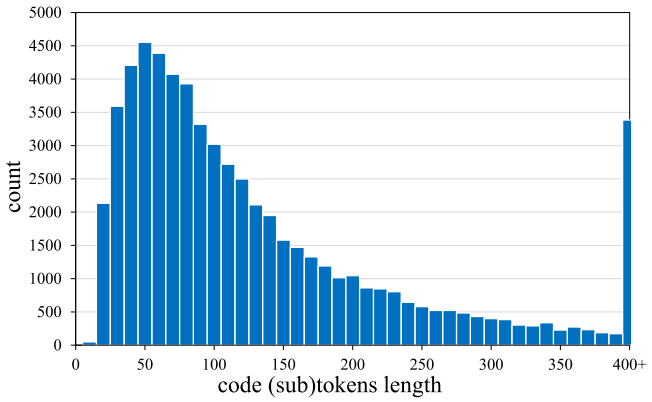

Fig. 6. The distribution of tokens length.

Table 1
Comparison of the overall performance between HCCS and baselines on the JHD dataset.

	Traditional metrics			BERTScore		
	BLEU	Metetor	Rouge	P	R	F1
Code-NN	28.85	21.32	33.46	35.52	35.66	35.59
tree-Lstm	22.63	16.54	27.71	33.36	31.13	32.21
deepCom	31.52	28.64	35.21	46.52	44.12	45.29
Tl-codeSum	40.98	29.95	53.86	55.98	57.95	56.92
NeuralCodeSum	34.14	30.62	48.45	42.58	40.19	41.35
ConvGNN	37.70	29.21	51.35	57.70	58.21	57.95
GCN+Transformer	36.14	28.15	52.59	55.31	57.41	56.12
DRL+HAN	37.25	28.67	50.88	54.44	56.82	56.63
HCCS	**41.65**	**31.43**	**54.85**	**65.17**	**63.32**	**64.23**

5.1. Dataset

Two public datasets containing numerous code–summary pairs are used to evaluate HCCS. One is a Java dataset collected by Hu et al. (Wan et al., 2018; Hu et al., 2018a). The other (Barone and Sennrich, 2017) based on Python programming language is well-established, adopted in many previous works (Wan et al., 2018; Choi et al., 2021). We spend some extra effort on data processing to avoid noise hindering the training process. To tackle the adverse effects of code duplication (via copy-and-paste code) (Allamanis, 2019), we have carefully deduplicated two datasets to avoid data leakage from the training set. Moreover, The samples with no method calls are removed from the original dataset since HCCS needs to extract API method call information to construct ACG. Additionally, the code sample should be truncated when its length exceeds the maximum limit.

At last, we get 69708 samples from Hu's Java language dataset (called JHD) and get 86525 samples from the Python Function dataset (called PFD). The average lengths of codes and comments in the JHD Dataset are 199.07 and 10.12. For the PFD dataset, the responding values are 172.87 and 9.45. Fig. 6 shows the distribution of code length in the JHD datasets. We discuss the impact of this parameter in Section 5.6. After shuffling the pairs randomly, we divide both datasets into 8:1:1 for training, validation, and testing.

5.2. Evaluation metrics

We use four metrics to evaluate the performance of code summarization approaches. BLEU (Papineni et al., 2001), ROUGE-L (Lin, 2004), and METEOR (Denkowski and Lavie, 2014) are popular in the literature to measure the quality of generated summaries. Apart from the traditional metrics popular in NMT areas, we also employ another robust evaluation metric, **BERTScore** (Zhang* et al., 2020). All the metrics are expressed in percentages, and a higher value indicates better performance. A score of 100% means the candidate summary is the same as the reference.

BLEU. BLEU is relied on the n-gram precision referred to as machine translation tasks. Specifically, it calculates the geometric mean of the n-gram matching precision scores reference sentences and imposes a brevity penalty for short predictions. In particular, we choose sentence level BLEU-4 with a smoothing function (Chen and Cherry, 2014).

METEOR. METEOR is a recall-oriented metric, which is an improvement of BLEU. Apart from considering uni-gram matching precision, it calculates the recall scores and applies a synonym matching mechanism.

ROUGE-L. ROUGE-L can be also regarded as a recall value, which computes the length of the longest common subsequence between the generated summary and the references.

BERTScore. Instead of string matching, BERTScore is relied on pre-trained BERT contextual embeddings, correlating highly with human judgment. Given a summary and its reference, we compute the precision, recall, and F1 scores.

5.3. Implementation details

During data processing, code tokens and ACG nodes retain respective vocabulary with sizes set to 50000 and 20000 and limit their maximum length to 400, and 20, respectively. In our model, the hidden size of LSTM and tree-LSTM is set to 512. We adopt 128-dimensional embedding layers for both encoders and decoders. During the training stage, the batch size is set to 32 and the training epoch is 50. Adam (Kingma and Ba, 2017) optimizes the model parameters with an initial learning rate of 0.002. We use beam search to find the best target sequence during decoding. The beam size and length penalties are set to 4 and 0.4, respectively.

5.4. Comparison with baselines

To verify the effectiveness of HCCS, we choose the eight state of the arts as baselines:

- **CodeNN** (Iyer et al., 2016) utilizes LSTM with an attention mechanism to generate summaries from pure code sequences, which first introduces neural machine translation to code summarization.
- **TreeLstm** (Shido et al., 2019) directly feeds AST into a tree-Lstm network to capture more latent information from source codes.
- **TL-codeSum** (Hu et al., 2018a) learns API sequence knowledge in advance and leverages the API contexts to the model.
- **DeepCom** (Hu et al., 2018b) proposes a novel structure-Based traversal to get linearized AST sequences, which are input to the standard Seq2Seq model.
- **NeuralCodeSum** (Ahmad et al., 2020) exploits a transformer model with a self-attention mechanism which could alleviate the long-range dependencies problem.
- **ConvGNN** (LeClair et al., 2020) exploits a GNN-based encoder to model AST, combined with an RNN-based encoder to process code tokens.
- **GCN+Transformer** (Choi et al., 2021) converts the graph-convolutioned ASTs into sequences and handles them with transformer layers.
- **DRL+HAN** (Zhou et al., 2022) injects hybrid representations (including code sequences, type-augmented ASTs and program control flows) into a DRL model.

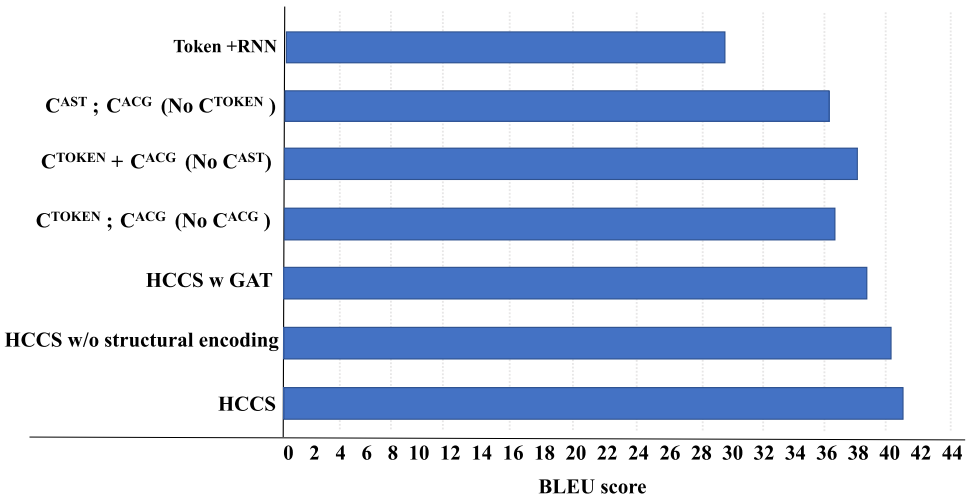

Fig. 7. BLEU scores of different encodings fusion.

Table 2
Comparison of the overall performance between HCCS and baselines on PHD dataset.

	Traditional metrics			t-test values		
	BLEU	Metetor	Rouge	Standard deviation	p-value	Significance
Code-NN	16.92	09.15	35.82	0.85	<0.001	+
tree-Lstm	14.82	09.04	32.12	1.08	<0.001	+
deepCom	20.78	12.98	37.35	0.50	<0.001	+
NeuralCodeSum	24.02	12.40	42.54	1.02	<0.001	+
ConvGNN	32.28	22.43	48.49	0.43	0.0028	+
Transformer+GNN	31.59	20.08	45.97	0.86	0.0063	+
HCCS (ours)	**33.54**	**23.41**	**51.40**	0.40		

The overall performance. The main results of HCCS against the prior baselines are reported in Table 1. From the table, we can observe an evident outcome that HCCS achieves the optimal results on the three traditional metrics. HCCS achieves 9.5%, 7.1%, and 6.4% improvements over other baselines (ConvGNN) in terms of the three metrics on the JHD dataset, respectively. To make the results more convincing, we also report BERTScore results for different models. Our approach scores 11.46%, 8.1%, 9.8% higher than ConvGNN in terms of the precision, recall, and F1 scores on the JHD dataset. The results demonstrate that the summaries generated by HCCS are highly similar to the ground-truth in semantic consistency. The BERTScore is factually capable of differing the performance of the code summarization approaches. It is apparent from this table that Code-NN and tree-Lstm perform poorly on both datasets. Both two approaches adopt a single encoder, which makes it not comprehensive to explore the code semantics. Meanwhile, the average length of input code tokens has increased. It may be that we remove the code samples without method calls from original datasets as illustrated in Section 5.1. The limitation of the maximum sequence length might exacerbate the long-range dependencies problem. In light of the above factors, Code-NN and tree-Lstm do not produce outstanding results. All the aforementioned results indicate the effectiveness of multi-model code representation of our approach.

The stability of our approach. The results achieved by a stable approach for several independent runs should be close enough to each other. Our purpose is to verify the stability of our approach. To this end, we conduct statistical tests (t-test) on the outcomes of the dataset PFD to see if the performance gain is statistically significant. The t-test assists in checking the presence of a significant difference in favor of our approach with respect to the other baselines. We execute our proposed approach and other baselines 5 times by randomly shuffling the dataset. The detailed results (mean BLEU scores) with statistical tests are represented in Table 2 and "+" symbol denotes that the result is significantly in favor of HCCS compared to others.

From the table, we can also observe that our approach is superior to the baselines. For the PFD dataset, the corresponding boosts are 3.5%, 4.2%, and 5.7%, which indicates that our approach can be also adapted to Python programming language for generating high-quality summaries. Besides, all the p-values are smaller than some significance level (0.05 in our tests) which shows the final results of HCCS are statistically significant with respect to the other baselines. Additionally, another finding in the table is that our approach holds a minimum standard deviation. Overall, the proposed approach achieves similar results during each run, which verifies its stability.

5.5. Ablation study

We conduct a series of experiments to identify the impact of different components of HCCS on the final performance. HCCS incorporates three encoders, producing three intermediate features, (graph context, tree context, and code context). This section first performs the ablation analysis to verify the contributions of each encoder. Considering that the graph encoder consists of two phases (structural encoding and qk-GAT), we further explore their impact, respectively. We just eliminate or replace the corresponding component to carry out the ablation experiment with the other components of the model constant. The detailed experimental results are shown in Fig. 7. The symbols and operators appearing in the figure or table (in Section 5.6) are described as follows: C^{ACG} stands for graph context; C^{AST} stands for tree context; C^{TOKEN} stands for code context; ";" represents the connection of two context features, and "+" represents the weighted addition of two features.

Analysis of three code features. An obvious outcome can be concluded from the Fig. 7 that all three types of code features

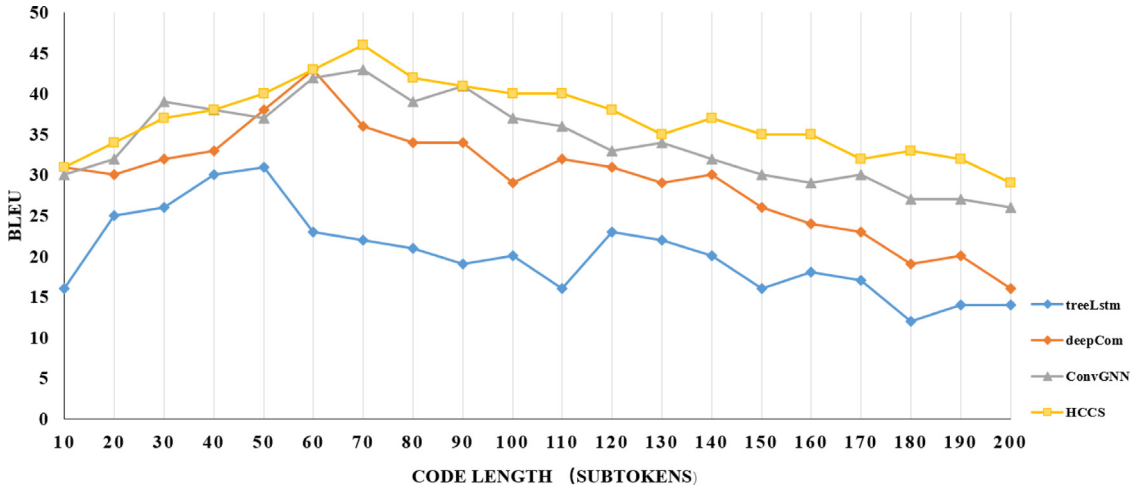

Fig. 8. The effect of tokens length.

Table 3
Impact of different fusion strategy.

Fusion strategy	BLEU	Meteor	Rouge
$C^{AST} + C^{TOKEN}$	37.98	29.85	53.14
$C^{ACG}; C^{TOKEN}$	38.16	29.99	53.53
$C^{ACG} + C^{TOKEN}$	37.36	28.54	52.35
$C^{ACG} + C^{AST} + C^{TOKEN}$	40.32	29.95	53.27
$C^{ACG}; C^{AST}; C^{TOKEN}$	40.12	30.03	52.84
$C^{ACG}; (C^{AST} + C^{TOKEN})$ (Ours)	41.65	31.43	54.85

can boost the performance of our approach. An 8.8% drop can be observed from the figure, which verifies the importance of ACGs which integrate API usage and structural features. After removing the tree encoder, the BLEU score decreases by approximately 8.4%. It can be attributed that The ASTs representing the syntactical structure information contribute to the final performance. Remarkably, the drop percent exceeds 13% without the token encoder. This shallow textual information which preserves code sequential features is also indispensable. It is worth mentioning that the three variations of our model outperform the single encoder model (Token+RNN) by a significant magnification. All the results address the effectiveness of the multi-modal code representations because code summarization benefits from extracting condensed and informative features.

Analysis of the graph attention module. We consider another variation of our model to examine the effectiveness of qk-GAT, in which the graph encoder incorporates a standard GAT module instead of qk-GAT. From the figure, the qk-GAT module makes a 6.8% increment, indicating that the graph attention module used in our approach can increase the representation of the graph encoder. Our graph attention module is more competent to extract high-level information from ACGs. Also, the proposed structural encoding technique improves the BLEU scores by injecting global structural information. As a result, the proposed graph attention module promotes the overall representation of source code.

5.6. Impact of different fusion strategy

Our model produces three intermediate features of source code, including graph context, tree context, and code context. This section analyzes the impact on the final results by applying different features fusion strategies. Detailed results are presented in Table 3.

A closer inspection of the table shows that integrating all three features outperforms the other cases remarkably, which validates the advantage of the multi-modal code representations. Another important finding that stands out in the table is that the concatenation way to fuse Graph context $[C^{ACG}; (C^{AST} + C^{TOKEN}]$ shows the optimal performance. There are several possible explanations for this result. First, the feature C^{ACG} learns the overall program logic, its vocabulary varying from the other two. Combining the feature via summation directly might discard important clues while decoding. In addition, the concatenation operation needs more trainable parameters compared with the addition, which might improve the capability of representing source code. Naturally, the combination way $[C^{ACG}; (C^{AST} + C^{TOKEN})]$ produces the best performance.

5.7. Effect of input lengths

The performance of the code summarization model varies while receiving samples with different input lengths. In this section, we want to analyze the prediction accuracy of the code summarization with different input lengths. To this end, we record several models' results (treeLSTM, deepCom, convGNN, and HCCS) in terms of the BLEU metric with different input lengths on the JHD dataset, as depicted in Fig. 8.

From Fig. 8 an upward trend is observed in all the models, as the code length increases. Tiny codes which have less than 20 tokens are not likely to have complete method call logic, even broken. This might explain why the models perform not well at first. When the code lengths are within 50 code length position, GSCS shows similar performance to GSCS. After that, HCCS starts a significant rise at 50 tokens position, maintaining the highest scores until the end, which indicates HCCS could achieve outstanding performance in most cases. HCCS achieve the highest scores at the 70 tokens position, clearly superior to other baselines. The results suggest that HCCS can fully explore the code semantics with multiple encoders.

Another finding that can be seen from the figure is that all the models tend to decrease when the code length exceeds 200 tokens. This result may be explained by the following facts. First, the longer code means more noise which makes it not stable for models to explore valid code semantics. Meanwhile, the models are not capable of identifying ground-truth dependencies between long-distance tokens with code length longer. In particular, when the code length reaches more than 400 tokens position, samples have a number of 3384, occupying partial proportions of datasets. These samples are supposed to be truncated and confined to the maximum code length as mentioned in Section 4.1. HCCS still stands out against other baselines. The results can be attributed

Table 4
Example from JHD for different models.

Source code	example 1 (JHD)

	`public void fling(int startX,int startY,int velocityX,int velocityY,`
	`  int minX,int maxX,int minY,int maxY,int overX,int overY)`
	`{`
	`  if (mFlywheel && !isFinished())`
	`  {`
	`  float oldVelocityX=mScrollerX.mCurrVelocity;`
	`  float oldVelocityY=mScrollerY.mCurrVelocity;`
	`  if (Math.signum(velocityX) == Math.signum(oldVelocityX)`
	`  && Math.signum(velocityY) == Math.signum(oldVelocityY))`
	`  {`
	`  velocityX+=oldVelocityX;`
	`  velocityY+=oldVelocityY;`
	`  }`
	`  }`
	`  mScrollerX.mMode=mScrollerY.mMode=FLING_MODE;`
	`  mScrollerX.fling(startX,velocityX,minX,maxX,overX);`
	`  mScrollerY.fling(startY,velocityY,minY,maxY,overY);`
	`}`

Ground-truth:	start scrolling based on a fling gesture.
Code-NN:	perform the fling.
deepCom:	start scrolling based on fling velocity.
ConvGNN:	start scrolling and change the velocity.
HCCS:	start scrolling based on a fling gesture.

Table 5
Example from CSD for different models.

Source code	example 2 (PHD)

	`def get_bulk_archive(selected_submissions, zip_directory=''):`
	`  zip_file = tempfile.NamedTemporaryFile(...)`
	`  sources = set([i.source.journalist_designation for ...])`
	`  with zipfile.ZipFile(zip_file, 'w') as zip:`
	`  for source in sources:`
	`  submissions = [s for s in selected_submissions if ...]`
	`  for submission in submissions:`
	`  filename = path(submission.source.filesystem_id, submission.filename)`
	`  verify(filename)`
	`  document_number = submission.filename.split('-')[0]`
	`  ...`
	`  return zip_file`

Ground-truth:	generate a zip file from the selected submissions.
Code-NN:	write a zip file.
deepCom:	get a zip file from sources.
ConvGNN:	return a zip file for selected submissions.
HCCS:	write a zip file from the selected submissions.

that HCCS learns a holistic representation from ACG separately. The ACG abandons the noises and preserves pivotal semantic and structural information of source codes, thus yielding high-quality summaries for the codes with massive tokens.

5.8. Case study

The aforementioned experimental results have verified the effectiveness of our approach from the aspect of statistics. To exhibit a visualized results, we provide two cases from the JHD dataset (Table 4) and the PFD dataset (Table 5). Intuitively, the summaries generated by our model correlate highly to ground-truth summaries. In the case of the JHD dataset, the example code represents an interactive operation, namely, "scroll based on a fling gesture". HCCS generates exactly the same statement as the reference. In contrast, the other models differ in more than one token, even failing to express the ground-truth semantics of the code snippet. This can be contributed that multiple encoders of HCCS produce a condensed and informative vector from different perspectives. Therefore, the decoder has more clues to generate high-quality summaries. In the case of the PFD dataset, the code snippet contains much noise information. From the results, HCCS missed only one token "generate", yet preserving semantic equivalence compared with the target summary. Other models can almost not capture important keywords from plenty of redundant tokens. This is because that HCCS extracts an overview logic from ACG, which eases the pressure on the attention mechanism to select every important part. To sum up, our model can learn a holistic and comprehensive representation from source code, which advances the art of states by a significant magnification.

6. Conclusion

Code summarization aims to generate a natural language description for a code snippet, which can facilitate software maintenance activities. This paper innovatively uses ACG to highlight the API calls and related control units, which model source code from a holistic perspective. To learn a comprehensive code representation, we propose a multi-modal approach, with a graph encoder for ACG, a tree encoder for AST, and a code encoder for the code tokens. The fusing encodings are passed to a joint-decoder for generating the summaries. We conduct a series of experiments on two datasets (JHD and PFD) to verify the effectiveness of our approach. Experimental results show that the proposed approach is superior to the existing state of the arts, and can effectively help development participants understand the code. Our future work plans to exploit a pre-train model on data-rich corpora for token and API embedding, which enhances the deep understanding of the code semantics.

CRediT authorship contribution statement

Kaiyuan Yang: Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Junfeng Wang:** Investigation, Validation, Supervision, Writing – review & editing. **Zihua Song:** Methodology, Validation, Data curation, Writing – review & editing.

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 supported in part by the National Key Research and Development Program under Grant 2022YFB3305203; in part by the National Natural Science Foundation of China under Grant U2133208 and Grant 62101368; and in part by the Sichuan Youth Science and Technology Innovation Team under Grant 2022JDTD0014 and in part by the Major Science and Technology Special Project of Sichuan Province under Grant 2022ZDZX0008.

References

Ahmad, Wasi, Chakraborty, Saikat, Ray, Baishakhi, Chang, Kai-Wei, 2020. A transformer-based approach for source code summarization. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 4998–5007. http://dx.doi.org/10.18653/v1/2020.acl-main.449, URL https://aclanthology.org/2020.acl-main.449.

Allamanis, Miltiadis, 2019. The adverse effects of code duplication in machine learning models of code. Onward! 2019, ISBN: 9781450369954, pp. 143–153. http://dx.doi.org/10.1145/3359591.3359735, URL https://doi.org/10.1145/3359591.3359735.

Bahdanau, Dzmitry, Cho, Kyunghyun, Bengio, Yoshua, 2016. Neural Machine Translation by Jointly Learning to Align and Translate. arXiv, http://dx.doi.org/10.48550/arXiv.1409.0473.

Barone, Antonio Valerio Miceli, Sennrich, Rico, 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. arXiv preprint arXiv:1707.02275.

Chen, Boxing, Cherry, Colin, 2014. A systematic comparison of smoothing techniques for sentence-level BLEU. In: Proceedings of the Ninth Workshop on Statistical Machine Translation. pp. 362–367. http://dx.doi.org/10.3115/v1/W14-3346.

Chen, Chi, Peng, Xin, Xing, Zhenchang, Sun, Jun, Wang, Xin, Zhao, Yifan, Zhao, Wenyun, 2021. Holistic combination of structural and textual code information for context based API recommendation. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/TSE.2021.3074309 (ISSN: 0098-5589, 1939-3520, 2326-3881).

Cheng, Xiao, Wang, Haoyu, Hua, Jiayi, Xu, Guoai, Sui, Yulei, 2021. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. (ISSN: 1049-331X) 30 (3), http://dx.doi.org/10.1145/3436877, URL https://doi.org/10.1145/3436877.

Cho, Kyunghyun, van Merriënboer, Bart, Gulcehre, Caglar, Bahdanau, Dzmitry, Bougares, Fethi, Schwenk, Holger, Bengio, Yoshua, 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). pp. 1724–1734. http://dx.doi.org/10.3115/v1/D14-1179.

Choi, YunSeok, Bak, JinYeong, Na, CheolWon, Lee, Jee-Hyong, 2021. Learning sequential and structural information for source code summarization. In: Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021. pp. 2842–2851. http://dx.doi.org/10.18653/v1/2021.findings-acl.251, URL https://aclanthology.org/2021.findings-acl.251.

Denkowski, Michael, Lavie, Alon, 2014. Meteor universal: Language specific translation evaluation for any target language. In: Proceedings of the Ninth Workshop on Statistical Machine Translation. pp. 376–380. http://dx.doi.org/10.3115/v1/W14-3348.

Hadi, Mohammad Abdul, Yusuf, Imam Nur Bani, Thung, Ferdian, Luong, Kien Gia, Lingxiao, Jiang, Fard, Fatemeh H., Lo, David, 2022. On the Effectiveness of Pretrained Models for API Learning. http://dx.doi.org/10.1145/3524610.3527886, arXiv:2204.03498 [cs].

Haque, Sakib, Eberhart, Zachary, Bansal, Aakash, McMillan, Collin, 2022. Semantic similarity metrics for evaluating source code summarization. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension. ICPC '22, ISBN: 9781450392983, pp. 36–47. http://dx.doi.org/10.1145/3524610.3527909, URL https://doi.org/10.1145/3524610.3527909.

Haque, Sakib, LeClair, Alexander, Wu, Lingfei, McMillan, Collin, 2020. Improved Automatic Summarization of Subroutines via Attention to File Context. In: Proceedings of the 17th International Conference on Mining Software Repositories. ISBN: 978-1-4503-7517-7, pp. 300–310. http://dx.doi.org/10.1145/3379597.3387449.

Hu, Xing, Chen, Qiuyuan, Wang, Haoye, Xia, Xin, Lo, David, Zimmermann, Thomas, 2022. Correlating automated and human evaluation of code documentation generation quality. ACM Trans. Softw. Eng. Methodol. (ISSN: 1049-331X) 31 (4), http://dx.doi.org/10.1145/3502853.

Hu, Xing, Li, Ge, Xia, Xin, Lo, David, Jin, Zhi, 2018b. Deep code comment generation. In: 2018 IEEE/ACM 26th International Conference on Program Comprehension. ICPC, pp. 200–20010. http://dx.doi.org/10.1145/3196321.3196334.

Hu, Xing, Li, Ge, Xia, Xin, Lo, David, Lu, Shuai, Jin, Zhi, 2018a. Summarizing source code with transferred API knowledge. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18. pp. 2269–2275. http://dx.doi.org/10.24963/ijcai.2018/314.

Iyer, Srinivasan, Konstas, Ioannis, Cheung, Alvin, Zettlemoyer, Luke, 2016. Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 2073–2083. http://dx.doi.org/10.18653/v1/P16-1195.

Jiang, Yuan, Su, Xiaohong, Treude, Christoph, Wang, Tiantian, 2022. Hierarchical semantic-aware neural code representation. J. Syst. Softw. (ISSN: 0164-1212) 191, 111355. http://dx.doi.org/10.1016/j.jss.2022.111355.

Kingma, Diederik P., Ba, Jimmy, 2017. Adam: A Method for Stochastic Optimization.

LeClair, Alexander, Bansal, Aakash, McMillan, Collin, 2021. Ensemble models for neural source code summarization of subroutines. In: 2021 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 286–297. http://dx.doi.org/10.1109/ICSME52107.2021.00032.

LeClair, Alexander, Haque, Sakib, Wu, Lingfei, McMillan, Collin, 2020. Improved Code Summarization via a Graph Neural Network. In: Proceedings of the 28th International Conference on Program Comprehension. ISBN: 978-1-4503-7958-8, pp. 184–195. http://dx.doi.org/10.1145/3387904.3389268.

Levy, Omer, Feitelson, Dror G., 2021. Understanding large-scale software systems–structure and flows. Empir. Softw. Eng. 26 (3), 1–39. http://dx.doi.org/10.1007/s10664-021-09938-8.

Li, Zheng, Wu, Yonghao, Peng, Bin, Chen, Xiang, Sun, Zeyu, Liu, Yong, Yu, Deli, 2021. Secnn: A semantic CNN parser for code comment generation. J. Syst. Softw. (ISSN: 0164-1212) 181, 111036. http://dx.doi.org/10.1016/j.jss.2021.111036.

Lin, Chin-Yew, 2004. Rouge: A package for automatic evaluation of summaries. In: Text Summarization Branches Out. pp. 74–81.

Mastropaolo, Antonio, Aghajani, Emad, Pascarella, Luca, Bavota, Gabriele, 2021. An empirical study on code comment completion. In: 2021 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 159–170. http://dx.doi.org/10.1109/ICSME52107.2021.00021.

Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z., 2014. Convolutional neural networks over tree structures for programming language processing. In: National Conference on Artificial Intelligence. http://dx.doi.org/10.13140/RG.2.1.2912.2966.

Nazar, Najam, Hu, Yan, Jiang, He, 2016a. Summarizing software artifacts: A literature review. J. Comput. Sci. Tech. 31 (5), 883–909. http://dx.doi.org/10.1007/s11390-016-1671-1 (ISSN: 1000-9000, 1860-4749).

Nazar, Najam, Jiang, He, Gao, Guojun, Zhang, Tao, Li, Xiaochen, Ren, Zhilei, 2016b. Source code fragment summarization with small-scale crowdsourcing based features. Front. Comput. Sci. 10 (3), 504–517. http://dx.doi.org/10.1007/s11704-015-4409-2 (ISSN: 2095-2228, 2095-2236).

Pan, Xuran, Song, Shiji, Chen, Yiming, Wang, Liejun, Huang, Gao, 2022. PLAM: A plug-in module for flexible graph attention learning. Neurocomputing (ISSN: 0925-2312) 480 (C), 76–88. http://dx.doi.org/10.1016/j.neucom.2022.01.045.

Papineni, Kishore, Roukos, Salim, Ward, Todd, Zhu, Wei-Jing, 2001. BLEU: A method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02. p. 311. http://dx.doi.org/10.3115/1073083.1073135.

Roy, Devjeet, Fakhoury, Sarah, Arnaoudova, Venera, 2021. Reassessing automatic evaluation metrics for code summarization tasks. Association for Computing Machinery, ISBN: 9781450385626, pp. 1105–1116. http://dx.doi.org/10.1145/3468264.3468588.

Shahbazi, Ramin, Sharma, Rishab, Fard, Fatemeh H., 2021. API2com: On the improvement of automatically generated code comments using API documentations. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension. ICPC, pp. 411–421. http://dx.doi.org/10.1109/ICPC52881.2021.00049.

Shido, Yusuke, Kobayashi, Yasuaki, Yamamoto, Akihiro, Miyamoto, Atsushi, Matsumura, Tadayuki, 2019. Automatic source code summarization with extended tree-lstm. In: 2019 International Joint Conference on Neural Networks. IJCNN, pp. 1–8. http://dx.doi.org/10.1109/IJCNN.2019.8851751.

Sui, Yulei, Cheng, Xiao, Zhang, Guanqin, Wang, Haoyu, 2020. Flow2Vec: Value-Flow-Based Precise Code Embedding. 4, (OOPSLA), http://dx.doi.org/10.1145/3428301, URL https://doi.org/10.1145/3428301.

Sutskever, Ilya, Vinyals, Oriol, Le, Quoc V., 2014. Sequence to sequence learning with neural networks. Adv. Neural Inf. Process. Syst. 27.

Tai, Kai Sheng, Socher, Richard, Manning, Christopher D., 2015. Improved semantic representations from tree-structured long short-term memory networks. In: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). pp. 1556–1566. http://dx.doi.org/10.3115/v1/P15-1150.

Tang, Ze, Shen, Xiaoyu, Li, Chuanyi, Ge, Jidong, Huang, Liguo, Zhu, Zhelin, Luo, Bin, 2022. AST-trans: Code summarization with efficient tree-structured attention. In: Proceedings of the 44th International Conference on Software Engineering. ISBN: 978-1-4503-9221-1, pp. 150–162. http://dx.doi.org/10.1145/3510003.3510224.

Veličković, Petar, Cucurull, Guillem, Casanova, Arantxa, Romero, Adriana, Lio, Pietro, Bengio, Yoshua, 2017. Graph attention networks. arXiv preprint arXiv:1710.10903.

Wan, Yao, Shu, Jingdong, Sui, Yulei, Xu, Guandong, Zhao, Zhou, Wu, Jian, Yu, Philip, 2019. Multi-modal attention network learning for semantic source code retrieval. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering. ASE, pp. 13–25. http://dx.doi.org/10.1109/ASE.2019.00012.

Wan, Yao, Zhao, Zhou, Yang, Min, Xu, Guandong, Ying, Haochao, Wu, Jian, Yu, Philip S., 2018. Improving automatic source code summarization via deep reinforcement learning. ASE 2018, ISBN: 9781450359375, pp. 397–407. http://dx.doi.org/10.1145/3238147.3238206, URL https://doi.org/10.1145/3238147.3238206.

Wan, Yao, Zhao, Wei, Zhang, Hongyu, Sui, Yulei, Xu, Guandong, Jin, Hai, 2022. What do they capture? - a structural analysis of pre-trained language models for source code. In: 2022 IEEE/ACM 44th International Conference on Software Engineering. ICSE, pp. 2377–2388. http://dx.doi.org/10.1145/3510003.3510050.

Wang, Yu, Dong, Yu, Lu, Xuesong, Zhou, Aoying, 2022a. GypSum: Learning Hybrid Representations for Code Summarization. http://dx.doi.org/10.1145/3524610.3527903.

Wang, Yanlin, Li, Hui, 2021. Code completion by modeling flattened abstract syntax trees as graphs. In: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 35. (16), pp. 14015–14023. http://dx.doi.org/10.1609/aaai.v35i16.17650.

Wang, Wenhua, Zhang, Yuqun, Sui, Yulei, Wan, Yao, Zhao, Zhou, Wu, Jian, Yu, Philip S., Xu, Guandong, 2022b. Reinforcement-learning-guided source code summarization using hierarchical attention. IEEE Trans. Softw. Eng. 48 (1), 102–119. http://dx.doi.org/10.1109/TSE.2020.2979701.

Wei, Bolin, Li, Yongmin, Li, Ge, Xia, Xin, Jin, Zhi, 2020. Retrieve and refine: Exemplar-based neural comment generation. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 349–360. http://dx.doi.org/10.1145/3324884.3416578.

Xia, Xin, Bao, Lingfeng, Lo, David, Xing, Zhenchang, Hassan, Ahmed E, Li, Shanping, 2017. Measuring program comprehension: A large-scale field study with professionals. IEEE Trans. Softw. Eng. 44 (10), 951–976. http://dx.doi.org/10.1109/TSE.2017.2734091.

Yang, Zhen, Keung, Jacky, Yu, Xiao, Gu, Xiaodong, Wei, Zhengyuan, Ma, Xiaoxue, Zhang, Miao, 2021. A Multi-Modal Transformer-based Code Summarization Approach for Smart Contracts. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC). ISBN: 978-1-66541-403-6, pp. 1–12. http://dx.doi.org/10.1109/ICPC52881.2021.00010.

Zhang*, Tianyi, Kishore*, Varsha, Wu*, Felix, Weinberger, Kilian Q., Artzi, Yoav, 2020. Bertscore: Evaluating text generation with BERT. In: International Conference on Learning Representations.

Zhou, Yu, Shen, Juanjuan, Zhang, Xiaoqing, Yang, Wenhua, Han, Tingting, Chen, Taolue, 2022. Automatic source code summarization with graph attention networks. J. Syst. Softw. (ISSN: 01641212) 188, 111257. http://dx.doi.org/10.1016/j.jss.2022.111257.

Zhou, Ziyi, Yu, Huiqun, Fan, Guisheng, 2020. Effective approaches to combining lexical and syntactical information for code summarization. Softw. - Pract. Exp. 50 (12), 2313–2336. http://dx.doi.org/10.1002/spe.2893 (ISSN: 0038-0644, 1097-024X).

Kaiyuan Yang received the B'S degree in Computer Science and Technology from Sichuan University of China, Chengdu in 2020. He is currently a doctoral candidate of Computer Science and Technology, Sichuan University. His recent research interests include software engineering, software analysis, software security and data mining.

Junfeng Wang received the M.S. degree in Computer Application Technology from Chongqing University of Posts and Telecommunications, Chongqing in 2001 and Ph.D. degree in Computer Science from University of Electronic Science and Technology of China, Chengdu in 2004. From July 2004 to August 2006, he held a postdoctoral position in Institute of Software, Chinese Academy of Sciences. From August 2006, Dr. Wang is with the College of Computer Science, Sichuan University as a professor. He serves as an associate editor for several international journals. His recent research interests include network and information security, spatial information networks and data mining.

Zihua Song received the M.S. degree in Computer Science and Technology from Guizhou University of China, Guiyang in 2019. He is currently pursuing the Ph.D. degree at the School of Cyber Science and Engineering, Sichuan University, Chengdu. His recent research interests include software security, software analysis, information security, and data mining.