Contents lists available at ScienceDirect

The Journal of Systems & Software

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

GT-SimNet: Improving code automatic summarization via multi-modal similarity networks[☆]

Xuejian Gao, Xue Jiang, Qiong Wu, Xiao Wang, Chen Lyu[*], Lei Lyu

Shandong Provincial Key Laboratory for Distributed Computer Software Novel Technology, School of Information Science and Engineering, Shandong Normal University, Jinan 250014, China

A R T I C L E   I N F O

Article history:
Received 15 December 2021
Received in revised form 11 July 2022
Accepted 23 August 2022
Available online 27 August 2022

Keywords:
Code summarization
Local API dependency graph
AST
GCN

A B S T R A C T

Code summarization aims to generate high-quality functional summaries of code snippets to improve the efficiency of program development and maintenance. It is a pressing challenge for code summarization models to capture more comprehensive code knowledge by integrating the feature correlations between the semantics and syntax of the code. In this paper, we propose a multi-modal similarity network based code summarization method: GT-SimNet. It proposes a novel code semantic modelling method based on a local application programming interface (API) dependency graph (Local-ADG), which exhibits an excellent ability to mask irrelevant semantics outside the current code snippet. For code feature fusion, GT-SimNet uses the SimNet network to calculate the correlation coefficients between Local-ADG and abstract syntax tree (AST) nodes and performs fusion under the influence of the correlation coefficients. Finally, it completes the prediction of the target summary by the generator. We conduct extensive experiments to evaluate the performance of GT-SimNet on two language datasets (Java and Solidity). The results show that GT-SimNet achieved BLEU scores of 38.73% and 41.36% on the two datasets, 1.47%∼2.68% higher than the best existing baseline. Importantly, GT-SimNet reduces the BLEU scores by 7.28% after removing Local-ADG. This indicates that Local-ADG is effective for the semantic representation of the code.

1. Introduction

In software development and maintenance, to quickly and precisely locate relevant code, program developers usually use natural language to document the code's functionality (Chen et al., 2021; Ling et al., 2016). However, this task is undoubtedly time-consuming for large code sizes. Automatically generating a natural language description for a code snippet that matches the function of the code, i.e., code summarization, not only reduces software development and maintenance costs but also helps program developers with program comprehension (He, 2019).

Research in recent years has shown that abstract syntax trees (ASTs) can effectively represent code syntax information. There is a significant amount of current work applying ASTs to code summarization tasks. For example, Chen and Wan (2019) improved the Seq2Seq model by proposing the Tree2Seq model applied to the code summarization task. Tree2Seq captures the AST information through an encoder and outputs the summarization using a decoder. Although Tree2Seq can express the syntactic structure of the code, it still encounters difficulties in overcoming the long dependency problems that occur when there are more AST nodes. Shido et al. (2019) proposed the multi-way TreeLSTM model to alleviate this problem with LSTM (Staudemeyer and Morris, 2019). In addition, Alon et al. (2019) proposed the Code2seq model to represent an AST as a collection of paths and output code summarization by selecting the relevant paths using a decoder. Code2seq only samples the AST combined paths for representation without considering the overall AST path information. Hu et al. (2018a) proposed DeepCom, which serializes ASTs with a new AST traversal method, structure-based traversal (SBT), and generates the corresponding code summarization. To further address SBT, Yang et al. (2021b) proposed the MMTrans method. They focus on two AST modalities: SBT sequences and graphs. MMTrans uses two encoders to learn the feature information of the two modalities and utilizes a multi-head attention mechanism to alleviate the long dependency problem. The above work demonstrates the ability of ASTs to effectively characterize code syntactic structures.

While there have been many efforts to improve our code summarization efforts, based on our observation, the existing approaches suffer from the following limitations:

☆ Editor: Dr Earl Barr.
* Corresponding author.

E-mail addresses: xuejiangao@outlook.com (X. Gao), xuejiangcs@gmail.com (X. Jiang), qiongwu2000@gmail.com (Q. Wu), xiao_w0930@163.com (X. Wang), lvchen@sdnu.edu.cn (C. Lyu), lvlei@sdnu.edu.cn (L. Lyu).

https://doi.org/10.1016/j.jss.2022.111495
0164-1212/© 2022 Elsevier Inc. All rights reserved.

- **Limitation 1: The global application programming interface (API) dependency graph (Global-ADG) introduces irrelevant API dependencies that can cause a poor semantic representation of the code.**
Current research efforts have identified that APIs in code play a vital role in the semantic representation of code. Hu et al. (2018b) proposed a method to implement code summarization named TL-CodeSum. To exploit the API information in the code, the model serially encodes and learns the APIs, which improves the model's ability to characterize the code semantic. However, dependencies between APIs are more appropriately expressed as graph structures. In our previous work (Lyu et al., 2021; Xuejian et al., 2021), we explored this and proposed a Global-ADG to model the API call relationships in all code snippets contained in the dataset to complete the characterization of code semantics. We verified the validity of the characterization in an automatic code generation task.
However, the Global-ADG cannot distinguish between the internal and external API dependencies of code snippets. As shown in Fig. 1, the API call dependency internal to code snippet 1 (blue node) can be modelled as a graph, and if a call to code snippet 1 occurs in other code snippets (red node), a globalized representation will construct this relationship. However, by itself, this call relationship is not part of the API call dependency of code snippet 1. We refer to this relationship as external noise, and the introduction of external noise can disrupt the call dependency within the target code snippet. Therefore, the Global-ADG is prone to introducing external irrelevant dependencies when learning semantic representations for a single code snippet. This global characterization approach is inappropriate since the code summary task addresses a local code snippet.

- **Limitation 2: Most existing approaches ignore the feature correlation between code API call dependencies and ASTs.**
Currently, some approaches pay attention to complete code summarization using both the semantic and syntactic structure of the code. Wang et al. (2020) used a control flow graph (CFG), AST and code text information to improve the ability of the model to characterize the code. However, the process of feature fusion uses a concatenation approach that ignores the correlation between code structures. A code is a hybrid structure of semantics and syntax, where semantics depend on different syntactic structures, and syntactic structures are reflected through semantic structures. Therefore, it is necessary to pay attention to the correlation between code syntax and semantics in code feature fusion. Studies have shown that learning the syntax of code using AST and learning the representation of code using ADGs can be beneficial (LeClair et al., 2020; Lin et al., 2021a). However, little work has considered fusing an AST and ADG to learn both syntax and semantics in code. Since the AST and Local-ADG representations are not in one dimension, there is an urgent need for an appropriate mapping or matching method to correlate and effectively fuse the embedded representation of an AST with the embedded representation of a Local-ADG.

In this paper, we propose a novel approach for improving code summarization that fuses local API dependency **G**raphs and AS**T**s via **Sim**ilarity **Net**works (GT-SimNet). To address the above limitations, we propose the following solutions with GT-SimNet.

- **Solution 1:** To solve the problem of global representation in **Limitation 1**, we propose a code semantic representation based on a local API call dependency graph (Local-ADG), where the Local-ADG serves only a single code snippet and can mask external code information (hereafter referred to as noise) while preserving the call constraints and call order among the APIs. Fig. 1 shows the comparative relationship between a Global-ADG and a Local-ADG.
For code snippet 1, Global-ADG looks for subgraphs from the Global-ADG based on the API call dependencies. Since external code snippets (e.g., code snippets 2 and 3) make calls to code snippet 1, Global-ADG constructs the relevant API relationships (red nodes). For code snippet 1 itself, such API relations are external noise, which affects the final semantic information expression of the code snippet. Local-ADG constructs API call dependencies on code snippet 1, which can effectively eliminate the threat of external noise and obtain targeted code semantic expressions.

- **Solution 2:** To solve the problem of feature correlation between code API call dependencies and ASTs in **Limitation 2**, we propose a similarity network (SimNet) approach to compute the similarity scores of Local-ADG and AST to fuse the semantic and syntactic knowledge of the code. Specifically, we extract both syntactic and semantic information of the code, the syntactic information of the code is obtained by AST, the semantic information is embodied by Local-ADG, and both structures are expressed by a graph convolutional network (GCN) for embedding. Moreover, in our previous work, we performed an undifferentiated fusion of syntactic and semantic structures of codes, ignoring the correlation between code structures. Therefore, we use a similarity network to note the correlation between code structures. The similarity network calculates the degree of similarity of all nodes of the AST to each node of the Local-ADG and performs the fusion of code syntax and semantic structures based on the similarity scores.

To verify the superiority of GT-SimNet, we conduct the following experiments: (1) GT-SimNet comparison experiments with baselines on two datasets. (2) Ablation experiments to discuss the role of each component of our model. (3) Studies of the effect of code snippets and natural language length on the code summarization model. (4) Human evaluations. The results show that GT-SimNet obtains 41.36%, 28.09% and 38.13% scores on Dataset B (Zhuang et al., 2020) and outperforms the state-of-the-art approaches by approximately 2.68% in terms of BLEU scores. Compared to Global-ADG, Local-ADG shows an improvement of approximately 3.6%. SimNet also produces an impact of 7.73%. The results of the ablation experiments show that the performance of our method is more robust when the code size increases. The scores of our model in human evaluation are concentrated between 7∼9, which indicates that evaluators hold focused opinions on GT-SimNet. GT-SimNet generates stable levels of code summaries. These experiments demonstrate that GT-SimNet has better performance than existing methods. To facilitate replication and future automatic code summarization techniques provide evaluation. We share the source code of GT-SimNet and the code corpus in the GitHub repository.[1]

The main contributions[2] of this paper are as follows:

[1] https://github.com/timesJohn/GT-SimNet.git.

[2] This paper is an important extension of our work originally reported in ICONIP 2021. Unlike the original paper, we innovated the fusion of Local-ADG and AST, and we proposed SimNet to describe the similarity of different modalities of the code and perform the fusion (contribution 2). We added ablation experiments to analyse the capability of each component of GT-SimNet, added case studies and error analysis to comprehensively discuss the effectiveness of GT-SimNet, and performed human evaluations to assess the actual effectiveness of our model.

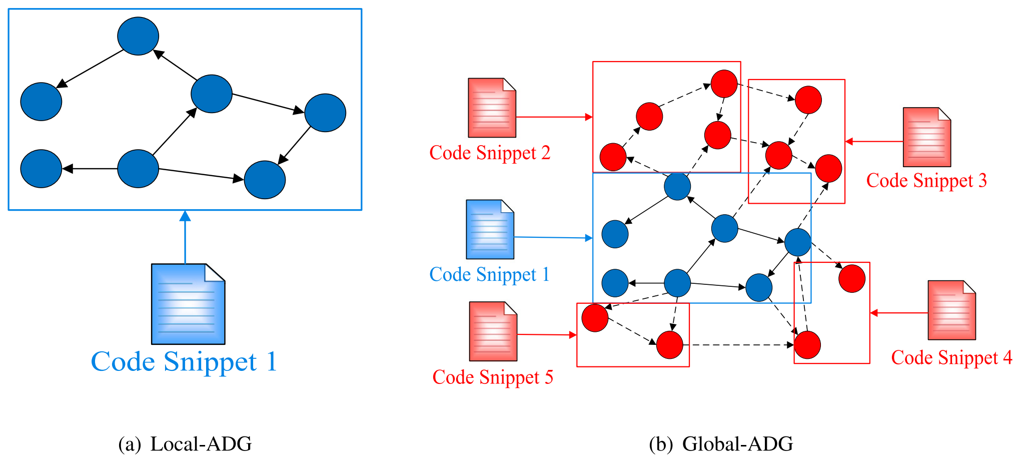


Fig. 1. Global-ADG and Local-ADG comparison results. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

- We propose a new model of code semantic structure, Local-ADG, which focuses on generating API dependencies for each code snippet to enhance code semantic structure representation.
- We present a multi-modal similarity network, SimNet, which performs code multi-modal fusion by measuring the similarity between different modal code embeddings. The characterization of the code structure is more comprehensive and accurate.
- We propose a new multi-modal code summarization method, GT-SimNet, which fuses code representations from the syntactic and semantic perspectives to generate high-quality code summaries.

2. Related work

2.1. Automatic code summarization

The core idea of automatic code summarization is the joint modelling of code and natural language, and the difficulty lies in the "structural gap" problem (Allamanis et al., 2018a). Specifically, unlike serialized natural languages, code has structural complexity. It contains program syntax information and a large number of API call relationships, parameter reference relationships, program framework information, etc. In syntactic structure, codes consist of three basic syntactic structures: sequential, branching, and looping. This implies structural differences between codes and natural languages (McBurney and McMillan, 2015; LeClair et al., 2021).

The earlier automatic code summarization techniques were borrowed from text summarization techniques (Alomari and Stephan, 2022). The core idea of these techniques lies in the extraction of keywords from the code (Desai et al., 2021; Kadar and Syed-Mohamad, 2015). For example, Hill et al. (2009) extracted keywords from the method names of code and comments to compose natural language descriptions of code. In actuality, the information extracted by these techniques represents the surface information of the code; thus, these techniques lack deep abstraction and exploration of the code structure and have certain limitations.

Deep neural networks have shown state-of-the-art performance in feature extraction and abstraction (Marcilio et al., 2020). Code summarization has also found new directions (Fang et al., 2019). For example, CODE-NN (Iyer et al., 2016) models code using LSTM (Staudemeyer and Morris, 2019; Blasi et al., 2021) and attention mechanisms (Luong et al., 2015). The Graph2Seq model (Xu et al., 2018) performs graph modelling for SQL languages and uses graph networks for learning. CoCoSUM (Wang et al., 2021) explores two different contextual contexts of intraclass and interclass relations of codes and proposes a multi-relational graph neural network (MRGNN) to encode and generate natural language summaries via a decoder with a two-layer attention mechanism. The state-of-the-art performance of the Transformer in the field of natural language processing enabled the exploration of applying the model to code summarization tasks. Influenced by this idea, Ahmad et al. (2020) used Transformer for the first time to model code. Specifically, they encoded the code with relative positions. In addition, Yang et al. (2021a) succinctly serialized the AST expression as input into Transformer along with the code token sequence for training. Transformer has powerful feature extraction capabilities and can effectively alleviate the long dependency problem. Combining a better code representation with Transformer may improve the effectiveness of code summarization.

This shows that the deep learning code summarization technique exhibits a much deeper representation of the features of the code. However, code is a hybrid structure of semantic and syntactic interdependencies, and research on the modelling, learning, and fusion of this hybrid structure is in its infancy. Properly modelling the semantic and syntactic structural information of codes and then more effectively extracting and representing them in a joint embedding is still a pressing problem.

2.2. Program representation models

A program representation model is a deep learning model that offers a formal representation of a program and allows program processing (Jiang and McMillan, 2017). In terms of presentation, programs can be expressed as sequences, ASTs, program dependency graphs, CFGs, etc. We classify program representation models into sequence-based program representation models, tree-based program representation models, and graph-based program representation models.

Sequence-based program representation models. Allamanis et al. (2016) proposed convolutional attention networks to express code feature information. The authors used convolutional layers to identify important parts of code sequences and to assign attention weights. The network is widely used on code summarization and function name prediction tasks. Iyer et al. (2016) proposed CODE-NN to represent the code. The model consists of LSTM and attention mechanisms. The authors applied CODE-NN to code summarization and code retrieval tasks. In another work, Hu et al. (2018b) proposed the TL-CodeSum model and applied it to the code summarization task; the TL-CodeSum model draws on code API information to enhance the semantic information about the program. Overall, the sequence-based program representation model achieves good performance to some extent with the help of the Seq2Seq model and CNNs. However, the serialized program representation still does not reflect the complete code structure.

Tree-based program representation models. An AST uses a tree structure to fully express code syntax information. Mou et al. (2014) proposed the tree-based convolutional neural network (TBCNN) model to model the representation of an AST. The authors apply CNNs to ASTs using continuous binomial trees and dynamic trees to adapt ASTs of different shapes. The TBCNN model achieves good performance on code classification and code detection tasks. Sun et al. (2020) proposed the TreeGen model. This model uses a new AST encoder fusing code syntax rules with AST structure information and uses Transformer to alleviate the long-term dependency problem to support code generation tasks. In another work, Zhang et al. (2019) proposed the abstract syntax tree neural network (ASTNN) model. The authors split large ASTs into small statement trees and use statement encoders to learn the embedding of each small statement tree. The model is widely used for program classification and code clone detection tasks. Lin et al. (2021b) proposed an AST splitting method, BASTS, to improve code summarization. BASTS splits the code of a method based on the blocks in the dominator tree of the control flow graph, and generates a split AST for each code split. Each split AST is encoded via Tree-LSTM and generates summary information via a Transformer. Most tree-based program representation models operate on objects as ASTs, enabling adequate representation of code syntax information.

Graph-based program representation models. Representing code as a graph allows one to study the code from another perspective. Xu et al. (2018) proposed the Graph2Seq model to represent code. The authors graphically express the SQL statement invocation relationship and use the graph embedding technique for vector representation. The authors apply the model to the code summarization task. Allamanis et al. (2018b) proposed using gated graph neural networks (GNNs) to represent semantic codes. Good performance is obtained in variable name prediction and variable misuse detection tasks. In another work, Sui et al. (2020) proposed a new code embedding method, Flow2Vec, which first generates an interprocedural value-flow graph for a program and resolves its adjacency matrix. Next, the reachability between two nodes is discussed by matrix multiplication, thus preserving the asymmetric transferability of full program dependencies. Flow2Vec achieves good performance on the task of automatic code summarization. In summary, graph-based program representation models utilize GNNs for graph modelling, enabling deeper exploration of semantic information from code.

This shows that the different forms of program representations facilitate the code structure extraction. Current work is performed to support program analysis downstream tasks by synthesizing multiple program representations. For example, Wan et al. (2019) represented codes as sequences, ASTs and CFGs to reflect program linear and nonlinear features, respectively. The multi-modal attention fusion layer integrates them into a single hybrid representation. The authors named the method as MMAN to support the code retrieval task. In contrast, we model programs as tree structures (ASTs) and graph structures (ADGs) to reflect the syntactic and semantic structure of the code. Importantly, we propose call constraints for the program API to reconstruct the semantic representation of the code. Furthermore, we use similarity networks to extract and fuse program semantic and syntactic similarities, while the MMAN focuses only on the attentional states in a single program representation.

2.3. Graph neural networks

In recent years, the rise of deep neural networks has revolutionized tasks that traditionally rely on manual feature extraction, such as speech recognition (Karmakar et al., 2021) and machine translation. Various end-to-end neural network models (LSTM, CNN, etc.) have been proposed for extracting latent features. Although traditional deep learning shows excellent advantages in extracting features from Euclidean data, it is weak in handling non-Euclidean data (Zhang et al., 2021). As a result, GNNs are proposed for extracting non-Euclidean data features. In this section, we classify GNNs into five types: graph convolutional networks (GCNs), graph attention networks (GATs), graph autoencoders (GAEs), graph generative networks (GGNs), and graph spatial–temporal networks (GSTNs).

GCNs. CNNs exhibit advantages in feature extraction, such as parameter sharing and translational invariance (Krizhevsky et al., 2012). To apply this advantage to the processing of graph data, Bruna et al. proposed GCNs to implement feature processing of graph data (Bruna et al., 2013). Similar to the CNN idea, the core idea of a GCN is to aggregate the edge information of graph data with the node information to generate a new node representation. GCNs demonstrate powerful aggregation capabilities for graph structures in tasks such as graph node classification and edge prediction.

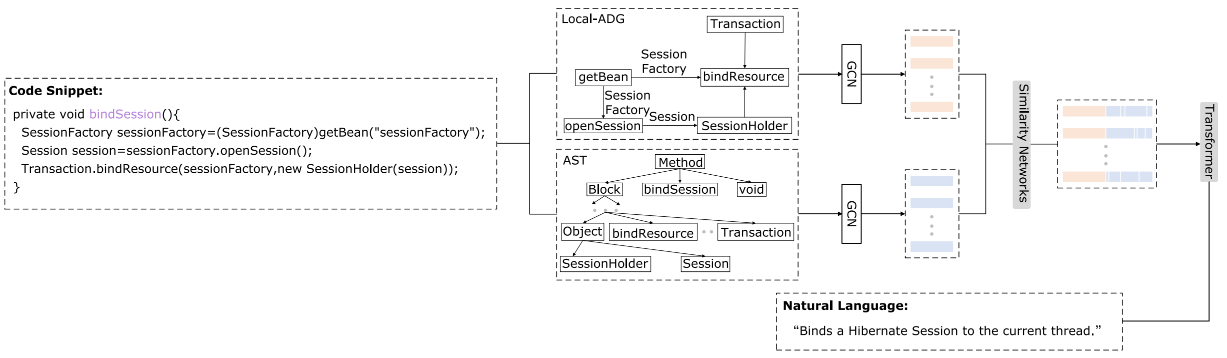
GATs. Attention mechanisms showed the top performance in processing Euclidean data. Thus, the attention of the research community turned to graphs and attention mechanisms. Veličković et al. (2017) proposed the use of GATs to assign different weights to graph nodes to optimize the connectivity relationships between nodes. The core idea of a GAT is to use self-attention to determine the weights of node neighbourhoods. In node classification and graph classification, the GAT reflects superior performance.

GAEs. The important task of a GAE is to represent graph data in a low-dimensional vector space. Kipf and Welling (2016) integrated a GCN into the graph encoder framework to obtain a graph node embedding representation. Finally, the decoder performs graph reconstruction based on this node embedding to compute the loss from the original graph. There are many variant structures of GAEs, such as the adversarially regularized variational graph autoencoder (ARVGA) (Pan et al., 2018), adversarially regularized autoencoder (NetRA) (Yu et al., 2018), structural deep network embedding (SDNE) (Wang et al., 2016), and deep recursive network embedding (DRNE) (Tu et al., 2018).

GGNs. The infrastructure of GGNs includes a generator, discriminator, and reward network. The generator performs discrete sampling of the adjacency matrix A and the feature matrix X. Finally, a GCN is used to optimize the sampling result into a vector representation. The discriminator and reward network output a score of 0$\sim$1 for this vector representation to update the model parameters.

GSTNs. Spatial–temporal networks are networks in which the nodes change over time, such as transportation networks. Therefore, extracting the temporal and spatial information from a spatial–temporal network is a problem that needs to be solved (Yu et al., 2017). Li et al. (2017) proposed the diffusion convolutional recurrent neural network (DCRNN) model to address spatial–temporal network problems. The model uses convolution to capture spatial information and a gated recurrent unit (GRU)-based encoder–decoder to capture temporal information. In addition, there are variants of graph spatial–temporal networks such as spatial–temporal graph convolutional networks (ST-GCNs).

In summary, graph neural networks show advantages in processing non-Euclidean data. In this paper, we use GCNs to aggregate feature information of ASTs and Local-ADGs. GCNs utilize



```
private void bindSession(){
  SessionFactory sessionFactory=(SessionFactory)getBean("sessionFactory");
  Session session=sessionFactory.openSession();
  Transaction.bindResource(sessionFactory,new SessionHolder(session));
}
```

"Binds a Hibernate Session to the current thread."

Fig. 2. Overall framework of our approach. In GT-SimNet, code snippets are represented as Local-ADG and AST to extract code information from multiple perspectives. We use GCNs for embedding the representation of multi-angle code information. GT-SimNets use multi-modal similarity networks to fuse multi-angle code information and output summary information using a summary generator.

the translation invariance of convolution to effectively extract structural information of the graph and embed the information of neighbouring nodes more accurately.

3. Approach

In this section, we present the detailed implementation of GT-SimNet, whose framework is shown in Fig. 2, and the method is divided into three submodules: (1) Multi-Modal code representation. This module aims to mine code information from multiple perspectives and embed code representations of different modalities through GCNs. (2) Multi-Modal similarity network. This module is responsible for the fusion representation of code embeddings in (1). (3) Summary generator. This module feeds the fusion vector into the encoder and outputs a natural language summary at the decoder.

3.1. Multi-modal code representation

The different modal representations of the code can capture more comprehensive information about the code. We use a Local-ADG to extract code semantic knowledge and an AST to capture code syntactic knowledge. A GCN generalizes convolution operators to graph data and plays a vital role in capturing structural dependencies. A GCN is essentially a feature extractor that generates new node representations by combining the features of neighbouring nodes with its own features. We embed a Local-ADG and an AST with GCNs to obtain more information about the code structure.

3.1.1. Local-ADG code representation

Definition 1 (*Local-ADG*)**.** Local-ADG is a graph model that represents the dependencies of API calls inside the target code snippet.

In this paper, APIs refer to methods in code snippets. We focus on the dependency constraints between API methods, mainly including (1) the invocation constraints between each API method (i.e., the API can be invoked when and only when all input parameters of the current API are provided). This can be done to guide the model to correctly learn the invocation relationships between API methods and to determine where the input and output parameters go. (2) The order of invocation between API methods can form a directed acyclic graph (DAG) to facilitate the representation of program semantics in the correct order, and the dependency constraints between API methods can effectively describe the sequential dependencies between program methods and better capture the long-range structural dependencies between methods. Fig. 3 shows an illustrative example of Local-ADG. In this graph, the nodes represent methods, and the edges represent the call relationships between methods. The detailed construction process is as follows.

- **Step 1:** Parse the code snippet into an AST with the help of the built-in syntax parser and extract the method, input and output parameters, function type and other information.
- **Step 2:** Establish the function call relationship based on the extracted information; the process is as follows.
- **Step 2-1:** Assume there exist two methods m_1 and m_2 (two nodes are represented in Fig. 3). Then, if a call relationship occurs between m_1 and m_2, an edge can be established between the two methods.
- **Step 2-2:** If node m_1 has an output parameter A and an edge relationship is established with node m_2, then an edge with label A may exist between the two methods.
- **Step 2-3:** Node m_2 can be called, and its function can be executed after input A is available. Input type A is provided by node m_1; then, there exists an edge pointing m_1 to m_2 with label A.
- **Step 3:** Based on the above steps, the API call dependency graph of the target code snippet is constructed and stored in the adjacency matrix.

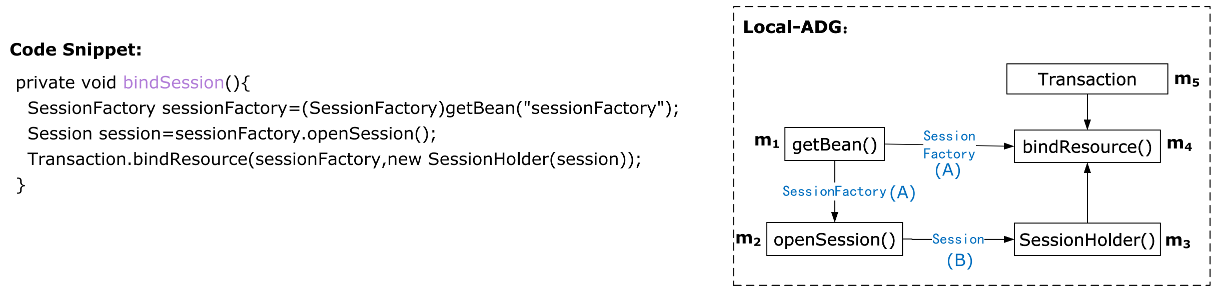
Compared to Global-ADG, Local-ADG has two main advantages: (1) It can more comprehensively capture the internal API call relationship of the target code snippets and effectively exclude external API call information interference. (2) The construction efficiency of Local-ADG is higher than that of Global-ADG when facing large-scale code data.

We use Flair[3] to vectorize each node of the Local-ADG to construct the initialization vector matrix of the Local-ADG: $S^{(0)} = \{S_1^{(0)}, S_2^{(0)}, \dots, S_V^{(0)}\}$, where V is the number of nodes contained in the Local-ADG. We input the adjacency matrix A_{V*V} of the local-ADG into the GCN for node information embedding to generate the Local-ADG embedding.

$$S^{(l+1)} = \delta(D_{\sim}^{-\frac{1}{2}} \cdot A_{\sim} \cdot D_{\sim}^{-\frac{1}{2}} \cdot S^{(l)} \cdot W^{(l)}) \tag{1}$$

where $S^{(l)}$ represents the node embedding vector matrix at the l layer, and each row in the matrix represents the Local-ADG node vector representation after GCN embedding at the l layer, denoted $S^{(l)} = \{S_1^{(l)}, S_2^{(l)}, \dots, S_V^{(l)}\}$, δ is the activation function, $A_{\sim}$ denotes the sum of the adjacency matrix A and the unit matrix E, i.e., $A_{\sim} = A + E$, $D_{\sim}$ denotes the degree matrix of $A_{\sim}$, $S^{(0)}$ is the initial vector representation, and $W^{(l)}$ is the weight of the GCN embedding at the l layer.

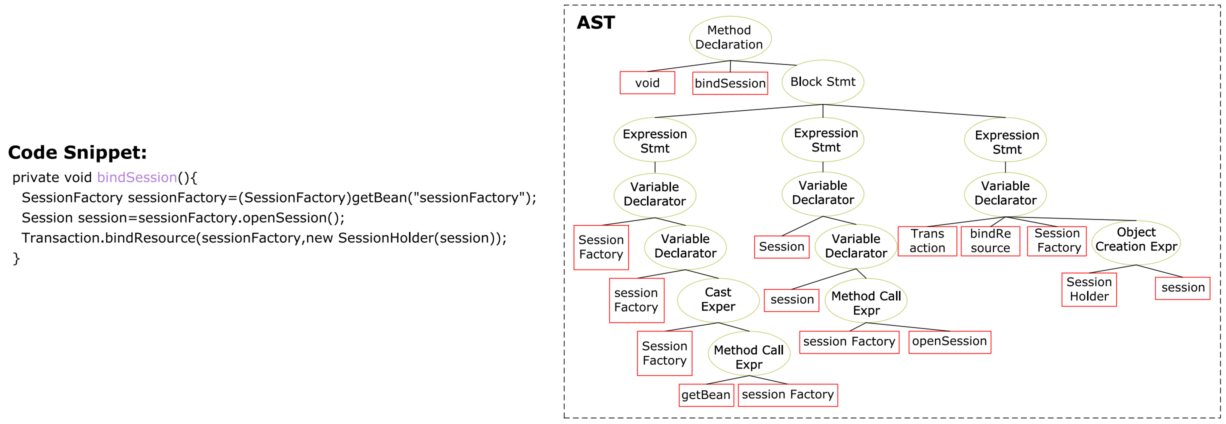
3 https://github.com/flairNLP/flair.



Code Snippet:

```
private void bindSession(){
  SessionFactory sessionFactory=(SessionFactory)getBean("sessionFactory");
  Session session=sessionFactory.openSession();
  Transaction.bindResource(sessionFactory,new SessionHolder(session));
}
```

Fig. 3. Local-ADG representation of the code snippet.



Code Snippet:

```
private void bindSession(){
  SessionFactory sessionFactory=(SessionFactory)getBean("sessionFactory");
  Session session=sessionFactory.openSession();
  Transaction.bindResource(sessionFactory,new SessionHolder(session));
}
```

Fig. 4. AST representation of the code snippet.

3.1.2. AST code representation

Definition 2 (*AST*)**.** AST refers to the tree structure used to express the code syntax structure information.

A structured representation of the syntax of code snippets can improve the accuracy of code summarization models. Thus, in this paper, we use an AST to express the syntactic structure of code snippets. We use the syntax parser to perform lexical analysis and syntactic analysis on the code and to extract the syntactic information of the code snippet to generate the corresponding AST. Fig. 4 shows a code snippet and the corresponding AST.

Compared with other source code representations, e.g., token sequences and CFGs, an AST uses a tree structure for syntactic information representation and provides a more comprehensive exploration of the code syntax structure. The performance of the code summarization model can be further improved by applying an AST to graph embedding methods to aggregate syntactic information.

Similar to the Local-ADG embedding, we represent the initialized vector matrix of the AST as $h^{(0)} = \{h_1^{(0)}, h_2^{(0)}, \ldots, h_m^{(0)}\}$ and use the GCN to obtain the embedding $h^{(l+1)}$ of the AST according to Eq. (1).

3.2. Multi-modal similarity networks

The multi-modal similarity network (SimNet) achieves multi-modal fusion by calculating the similarity score between Local-ADG and AST. As shown in Fig. 5, SimNet is divided into two parts: a similarity metric and a fusion network. The similarity metric obtains the similarity score by calculating the similarity between Local-ADG and AST. The fusion network fuses Local-ADG and AST by the similarity score.

3.2.1. Similarity metric

Cosine similarity measures the magnitude of the difference by calculating the cosine value between two vectors (Luo et al., 2018). In this paper, we use the cosine similarity to perform similarity calculations on the Local-ADG and AST. Specifically, we use the cosine similarity calculation to represent the similarity of all nodes in the AST to each node of the Local-ADG. The equation is as follows:

$$C_i^j = \frac{S_i \cdot h_j}{\|S_i\|_2 \cdot \|h_j\|_2} \tag{2}$$

where C_i^j represents the similarity scores of all nodes in AST to each Local-ADG node. Taking Local-ADG node S_1 as an example, we need to calculate the similarity scores of $(h_1 \sim h_{10})$ to S_1, that is, $(C_1^1, \ldots, C_1^j, \ldots, C_1^{10})$, $\|S_i\|_2$ and $\|h_j\|_2$ refer to the modulus of the vector.

3.2.2. Fusion network

C_i^j reflects the similarity of all AST nodes to each Local-ADG node, and the higher the similarity is, the higher the importance to the Local-ADG node. We aim to achieve the fusion of different modal program embeddings by computing the similarity between AST and Local-ADG, rather than simply adding them together. Specifically, for AST nodes $(h_1, \ldots, h_j, \ldots, h_m)$, we calculate the similarity of Local-ADG nodes (S_i) to all AST nodes by Eq. (2) and generate similarity coefficients $(C_i^1, \ldots, C_i^j, \ldots, C_i^m)$, where each similarity coefficient corresponds to an AST node. The higher the similarity coefficient is, the more important it is for S_i (e.g., C_i^1 is 0.7 and C_i^j is 0.2, which indicates that C_i^1 is more important for S_i).

We define Z_i as the sum of the weighted output of the hidden state of all AST nodes and the corresponding Local-ADG node (S_i),

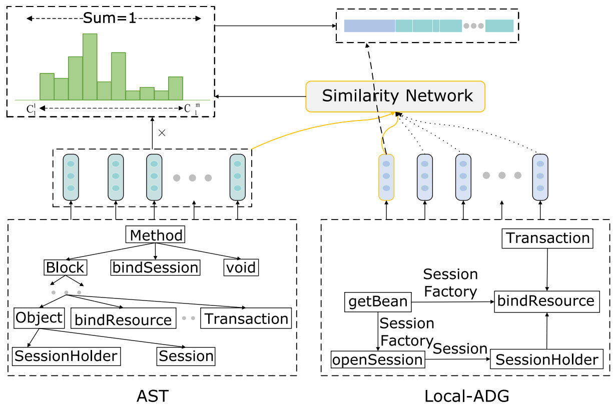


Fig. 5. Multi-Modal similarity network. In SimNet, we first calculate the similarity scores of all AST nodes to each Local-ADG node using cosine similarity, and then we input the similarity scores to the softmax layer for normalization and finally match the normalized similarity scores with AST nodes and complete the fusion with Local-ADG nodes.

and the equation is expressed as

$$Z_i = \sum_{j=1}^{T_x} \alpha_i^j h_j + S_i \tag{3}$$

where Z_i represents the Local-ADG and AST fusion vector, h_j is the AST node vector, and S_i is the node vector corresponding to the Local-ADG. α_i^j is the weight under each hidden state h_j and is calculated as follows.

$$\alpha_i^j = softmax(C_i^j) \tag{4}$$

where C_i^j is the similarity coefficient calculated by Eq. (2), which we normalize to obtain the weight of the AST node hidden state h_j.

The final generated fusion vector Z_i contains the weighted AST node with the corresponding Local-ADG node S_i. In this way, we can realize the multi-modal feature fusion expression of the program based on the similarity coefficients, and the characterization of the program is more comprehensive and accurate.

3.3. Summary generator

Transformer has powerful feature extraction capabilities that can effectively alleviate the long dependency problem. Therefore, we input Z_i into Transformer for weight assignment using its multi-headed self-attention mechanism and finally train it in the decoder together with the target summary.

The encoder of Transformer consists of N blocks, and each block contains two parts: the multi-head self-attention mechanism and the feed-forward network. In addition, residual connection and normalization are added after each part.

$$A^n = LayerNorm(\tilde{A} + FFN(\tilde{A})) \tag{5}$$

$$\tilde{A} = LayerNorm(A^{n-1} + MultiAttn(A^{n-1}, A^{n-1}, A^{n-1})) \tag{6}$$

$$FFN(\tilde{A}) = max(0, \tilde{A}W_1 + b_1)W_2 + b_2 \tag{7}$$

where $A^{(n-1)}$ is the output of the previous encoder block. Notably, $A^0 = Z_i$. FFN contains two fully connected layers, W_1, W_2, b_1, b_2 are the weights and biases of FFN, and $MultiAttn$ denotes the multi-head attention mechanism, which contains multiple self-attention heads and is calculated as follows.

$$MultiAttn(Q, K, V) = Concat(head_1, \dots, head_h)W^O \tag{8}$$

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \tag{9}$$

$$Attention(\tilde{Q}, \tilde{K}, \tilde{V}) = softmax(\frac{\tilde{Q}\tilde{K}^T}{\sqrt{d_k}}\tilde{V}) \tag{10}$$

where Q, K and V are from the previous block $A^{(n-1)}$. i is the number of self-attention heads; $\sqrt{d_k}$ is the square root of a head dimension; and W^O, W^Q, W^K and W^V are learnable parameter matrices.

The construction of the decoder is basically the same as that of the encoder. In particular, the decoder's input is the natural language encoding g and the output vector of the encoder, and the output is the probability distribution of the output words corresponding to position j. In addition, Q of the self-attention mechanism is a linear transformation of the output from the previous position, and K and V are linear transformations of the output vector from the encoder.

Before inputting the natural language encoding g, Transformer first adds the positional code to g to determine the input order. The equation is as follows:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}}) \tag{11}$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}}) \tag{12}$$

where pos is the position index. If the length of x is L, then $pos = 0, 1, 2, \dots, L-1$. d_{model} is the vector dimension; if $d_{model} = 512$, $2i$ represents an even dimension, and $2i+1$ represents an odd dimension, i.e., $(2i \le d_{model}, 2i+1 \le d_{model})$.

In the decoding process, for time t, we obtain the output at time t by relying on the output before time t; therefore, we mask the information after t. We solve this problem by applying an upper triangular matrix (with all upper triangular values of 1 and lower triangular and diagonal values of 0) on the decoding ends.

Note that: we do not perform positional encoding in the Transformer encoder. This is because there are no sequential relationships between the embedding vectors that we obtain using the GCN. This is different from a traditional Transformer.

4. Experimental setup

In this section, we describe the datasets, evaluation metrics, baselines and parameter settings used in the experimental evaluation.

4.1. Dataset

We use two publicly available language datasets (Java and Solidity) for code summarization experiments. Dataset A is a collection of 10 open source Java projects cloned from GitHub by Allamanis et al. (2016). Dataset B follows the code corpus collected by Zhuang et al. (2020) from Etherscan.io. Specifically, Dataset B contains mainly 40932 Ethereum Smart Contracts (ESC) code written by Solidity. Solidity[4] is an object-oriented programming language for implementing ESC. Solidity is a statically typed language with a syntactic structure similar to JavaScript. Dataset B contains mainly containing normal methods, modifiers and fallback methods. Normal methods include constructors and other functional methods. Modifiers are used to change the behaviours of functions declaratively. Fallback methods are executed on a call to the contract. Both code corpora have been widely used for automatic code summarization tasks. To standardize the data format, we processed Dataset A and Dataset B as follows.

[4] https://docs.soliditylang.org/en/latest/

Table 1
Statistics of code length and summary length for Dataset A and Dataset B.

Statistics for code length						
	Avg.	Mode	Median	≤100	≤150	≤200
A	99.94	16	65	81.63%	88.56%	96.33%
B	62.14	14	52	80.77%	89.18%	94.56%
Statistics for Comment Length						
	Avg.	Mode	Median	≤10	≤20	≤30
A	8.86	8	13	75.50%	86.79%	95.45%
B	10.04	8	10	57.18%	96.53%	99.46%

To select target natural language summaries in Dataset A, we extract the description between "/*" and "*/" in each Java file, whereas for Dataset B, we follow the method used by Zhuang et al. (2020). Specifically, comments in smart contracts are usually stored in the order of @notice, @dev, @return, "//" and "/*/". Therefore, we prioritize extracting the text information after @notice; if there is no @notice tag, we extract the information under @dev tag, and so on. Moreover, based on previous work, we extract the first sentence of the text as the target natural language summary. Finally, we remove those data with fewer than 4 words of comments because these comments are not enough to express the code information, which is noise data.

To obtain valid <code, summary> pairs, we filtered low-quality data. First, code snippets that were empty or contained only one character were excluded from the final dataset. Summaries containing a large number of incorrect words are also filtered. Second, we look for overridden methods by approximate static analysis, checking inheritance relationships and @Override annotations. Overridden methods are removed because overridden methods are highly repetitive. We clean up the setter, getter, and test methods because they serve a single function and generate a more fixed summary. Table 1 shows the statistics of code length versus annotation length for the two datasets after data cleaning.

After data cleaning, we find that the average length of Java code and annotations in Dataset A is 99.94 and 8.86, respectively, and more than 95% of the annotations do not exceed 30 characters; 96% of the Java methods do not exceed 200 characters. The average lengths of Solidity code and comments in Dataset B are 62.14 and 10.04, respectively, and more than 99% of comments do not exceed 30 characters; 94% of Solidity methods do not exceed 200 characters. We remove the comments of more than 200 tokens of Solidity methods and summaries of more than 30 characters. Dataset A[5] yields 69,708 <code, summary> pairs and Dataset B[6] obtains 347,410 data samples.

We use the k-fold cross-validation method to evaluate the performance of the proposed method. Specifically, we divide the dataset into 10 parts and use 10% of them each time as the validation set and the remaining 90% as the training set. Finally, the results after 10 executions are averaged.

4.2. Parameter setting

We implement our model with PyTorch. In the training process, the maximum number of nodes for the AST and Local-ADG is set to 400. In addition, the number of encoder layers $N = 6$ for the Transformer, $h = 8$ for the self-attention header, and the numbers of layers for the decoder and encoder are set to the same value. We debug the source code and choose the optimal *d_model* and *n_layers*. GT-SimNet utilizes two layers of GCNs with a fixed hidden state of 256 dimensions. The word embedding dimension is 256-dimensional. The training batch size is set to $\{2048, 4096, 8129\}$, and the number of epochs is set to 50. The initial learning rate is set to 1.0 and decayed using the rate of 0.99. GT-SimNet clips the gradients norm by 5. We use dropout strategy during the training process and $P_{drop} = 0.8$. GT-SimNet utilizes Glorot initialization to randomly initialize all parameters and uses the Adam optimizer to optimize our experiments.

4.3. Metrics

GT-SimNet uses BLEU, METEOR, and ROUGE to assess the validity of the experiment. The criteria for calculating each metric in the experiment are shown below.

BLEU: BLEU is the first experimental evaluation criterion we used (Papineni et al., 2002); this criterion is used to compare the degree of match between the generated natural language and the reference natural language n-gram. Depending on the value of n, BLEU can be classified as BLEU-1/2/3/4. A higher BLEU score indicates a closer match between the generated natural language and the reference natural language. In this paper, we use BLEU-4 as our evaluation metric. The following equation calculates the metric:

$$BLEU = BP \times exp(\sum_{n=1}^{N} w_n \log P_n) \tag{13}$$

where w_n denotes the weight parameter. $P_n = \frac{\sum_{n-gram\in c} count(n-gram)}{\sum_{n-gram'\in c'} count(n-gram')}$ is the ratio between the generated comment and ground truth at the n-word phrase (n-gram). BP is the penalty and is calculated as follows:

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

where c is the generated natural language length and r is the reference natural language length.

METEOR: METEOR was proposed by Banerjee and Lavie (2005) and was inspired by the recall parameter. Specifically, unlike BLEU, METEOR considers the accuracy and recall of the entire corpus. It utilizes knowledge sources such as WordNet for synonym matching. The final result of METEOR is an average of the generated natural language and the reference natural language in terms of accuracy and recall, and this parameter is calculated as follows.

$$METEOR = (1 - \gamma \cdot frag^{\beta}) \cdot \frac{P \ \cdot R}{\alpha \cdot P + (1-\alpha) \cdot R} \tag{14}$$

where P and R are the precision and recall of the unigram, respectively, $frag = \frac{ch}{m}$ is a fragmentation score, ch is the number of tokens, and m is the matched token number. α, β, and γ are three penalty parameters and take values of 0.9, 0.3 and 0.5, respectively.

ROUGE-N: To fully evaluate the performance of the different models, we also use the value of ROUGE (Lin, 2004) to measure the "similarity" between the generated natural language and the reference natural language. According to different calculations, ROUGE can be classified as ROUGE-N/L/W/S. In this paper, we use ROUGE-N. ROUGE-N is based on the recall on an n-gram. For any n, we calculate the total number of n-grams in all reference natural languages and determine how many of the same n-grams exist in the generated natural language. The formula is expressed as:

$$ROUGE - N = \frac{\sum_{s\in RefSummaries} \sum_{gram_n \in s} Count_{match(gram_n)}}{\sum_{s\in RefSummaries} \sum_{gram_n \in s} Count_{gram_n}} \tag{15}$$

[5] https://drive.google.com/drive/folders/14AmfMHoYCyF4ETFlQVHwEKwLlaRobTZB?usp=sharing.

[6] https://drive.google.com/drive/folders/1WPnv0n8BTJHSb9LtYNKrmSncNv6yKzWX?usp=sharing.

where the denominator indicates the total number of n-grams in the summaries of the reference, and the numerator is the number of n-grams shared between the reference summaries and the generated summaries.

Note that: BLEU, METEOR, and ROUGE-N take values in the range of [0, 1] and that to visualize the experimental results, we present them in the form of percentages.

4.4. Baselines

We compare GT-SimNet to the following baseline. The hyper-parameters for each baseline follow the relevant settings in the original paper or code.

CODE-NN (Iyer et al., 2016) is the first data-driven neural network code summarization model proposed by Iyer et al. and represents serialized program representation. It uses recurrent neural networks (RNN) and LSTM to capture source code tokens to generate natural language summaries, and the authors have conducted code summarization experiments on the C# and SQL languages. CODE-NN initializes all parameters between -0.35 and 0.35, and the learning rate is defined as 0.5. In addition, in CODE-NN, the LSTM, token and digest embedding dimensions are unified to 400, the network is trained for 80 epochs, the beam is 10 during decoding, and the maximum summary length is 20 words.

Funcom (LeClair et al., 2019) combines two types of information from the source code: code text information and AST syntax information. Funcom has designed two encoders and a decoder. It uses a GRU for recursive embedding and focuses on the code text and output summary words by an attention mechanism. A separate attention mechanism attends the summary words to parts of the AST. The vectors of each attention mechanism are then connected to create a context vector. The Seq2seq training method was followed. In the training process, Funcom goes through a total of 10 epochs, and the performance is calculated on the validation set after each epoch. Finally, the authors select the best validation performance among the 10 epochs as the final model.

Hybrid-DeepCom (Hu et al., 2020) is a code summarization model based on lexical and syntactical information. Hu et al. extend the study of DeepCom by analogizing the code summarization generation task to a machine translation task. Hybrid-DeepCom combines source code token sequence information with AST sequence information to better generate comments. In addition, Hybrid-DeepCom alleviates the out-of-vocabulary (OOV) problem by dividing identifiers. Hybrid-DeepCom uses a single-layer GRU network with hidden state and word embedding dimensions set to 256 and the learning rate of 0.5. Moreover, Hybrid-DeepCom calculates BLEU scores every 2000 minibatches. The learning rate is decayed using a rate of 0.99. The beam width is set to 5.

Graph2Seq (Xu et al., 2018) is a graph-based code summarization model. Xu et al. perform graph representation of code semantic information, aggregate forward and backwards neighbour information by Graph2Seq, and output summary information at the decoder using the node attention mechanism for weight assignment. Graph2seq uses GloVe to initialize word embeddings with dimensions of 300, and they are trained by the Adam optimizer with mini-batch set to 30. The learning rate during training is 0.001, and a dropout strategy of 0.5 is used. Finally, the hop size of the graph encoder is fixed to 6, and the hidden state of the decoder is 300.

TL-CodeSum (Hu et al., 2018b) is a code summarization model that makes use of API information. It extracts the code's API information, sequentially encodes it using an encoder, and then applies the learned API sequence information and code token information to the code summarization task. Experimental results on Java projects show that TL-CodeSum is effective. For training TL-CodeSum using the SGD algorithm, the batch size is set to 32, the maximum length of the source code and API is 300 and 20, respectively, and the maximum length of the abstract is 30 words. The authors set all GRU network, token and summary dimensions to 128.

DRL+HAN (Wang et al., 2020) uses a type-augmented AST and program control flow for code representation and employs a hierarchical attention mechanism to reflect the hierarchical structure of the code. DRL+HAN feeds code representation information to a deep reinforcement learning (DRL) framework for summary generation. For DRL+HAN, both the LSTM network and the word embedding dimension are set to 512, the mini-batch size is set to 32, and the learning rate is 0.001. Finally, DRL+HAN is trained on the actor–critic network for 10 epochs, and perplexity/reward is calculated every 50 iterations.

AST-Transformer (Tang et al., 2021) is an automatic code summarization model for linear AST encoding. Tang et al. proposed that existing linearized AST algorithms (SBT, POT, PD, etc.) produce excessively long AST sequences and thus create long dependency problems. The AST-Transformer is designed with two relationship matrices to encode ASTs efficiently; the relationship matrices are fed into the Transformer encoder for computation. Finally, the summary information is output in the decoder.

ComFormer (Yang et al., 2021a) is a Transformer model based on hybrid code representation. Yang et al. considered code sequence information and AST syntax information. Furthermore, ComFormer proposes Sim-SBT to serialize AST representations and uses the Byte-BPE algorithm to split identifiers. ComFormer inputs code sequences and ASTs into the Transformer and generates code summaries by the beam search algorithm. For the parameters, ComFormer chooses AdamW as the optimizer and uses the cross-entropy loss function. In addition, the learning rate is set to 0.0005, and training is performed for 30.

SeCNN (Li et al., 2021) is a CNN-based model for automatic code summarization. Li et al. designed two CNNs to capture the features of code tokens and ASTs and generated summary information by LSTM of the code attention mechanism. In particular, SeCNN proposes the ISBT method to traverse the AST. SeCNN uses a 9-layer CNN with a convolutional kernel size of 4×500 and a convolutional step size of 4. The decoder LSTM hidden state and word embedding are unified to 500. In addition, SeCNN chooses the cross-entropy loss function. The initial learning rate is set to 1.0. The learning rate is decayed using the rate of 0.99.

5. Experimental results

In this section, we evaluate the performance of GT-SimNet through comparison experiments with the baselines. In addition, we conduct ablation experiments to evaluate the contribution of each GT-SimNet component. Furthermore, we record the time efficiency of each component of GT-SimNet. Finally, we discuss the effect of different lengths of source code and natural language summary lengths on the effectiveness of GT-SimNet.

5.1. GT-SimNet vs. baselines

Table 2 shows the results of the experimental comparison of GT-SimNet with different baselines. According to the different types of code representations, we classify baselines into single-code representations and hybrid code representations. Overall, the hybrid code representation baseline is better than the single-code representation. Hybrid code representation baselines consider more code information, such as code sequence information,

Table 2
Experimental results of GT-SimNet on Dataset A and Dataset B compared to baselines.

	Method	Datasets A			Datasets B		
		BLEU	METEOR	ROUGE-N	BLEU	METEOR	ROUGE-N
Single code representation	CODE-NN	25.66	16.59	26.08	27.33	18.22	27.77
	Graph2Seq	28.42	17.86	28.12	29.37	19.31	30.58
	TL-CodeSum	33.68	18.96	30.19	32.55	20.47	31.33
	AST-Transformer	36.78	21.02	33.68	37.66	22.93	36.51
Hybrid code representation	Funcom	35.59	20.33	32.44	35.73	24.69	33.76
	DRL+HAN	37.15	21.54	34.83	37.99	25.53	36.24
	Hybrid-DeepCom	36.05	22.88	33.61	36.32	24.11	34.98
	SeCNN	37.01	21.26	35.31	37.79	23.54	36.20
	ComFormer	37.26	22.19	35.68	38.68	25.61	37.37
	GT-SimNet	**38.73**	**24.02**	**36.84**	**41.36**	**28.09**	**38.13**

AST and CFG. Our approach belongs to the group represented by the hybrid code representation baseline.

GT-SimNet achieves a BLEU score, METEOR score and ROUGE-N score of 38.73%, 24.02% and 36.84% on Dataset A and 41.36%, 28.09% and 38.13% on Dataset B, respectively. Specifically, CODE-NN exhibits the worst results for Dataset A, which is not surprising since it understands the code only at the syntactic level. Graph2seq obtains a BLEU score of only 28.42%. Furthermore, we note that Graph2seq operates on the SQL language; however, compared to the Java language, the SQL core function has only 9 verbs, which is close to spoken English, so the difficulty of automatic code summary is less than that of the Java language. TL-CodeSum and Funcom achieve better experimental results. TL-CodeSum generates a source code summary by extracting code API information, while Funcom combines code text information and AST information for the embedded representation of source code. Therefore, these two methods are better than CODE-NN and Graph2seq in extracting code structure. In contrast, Hybrid-DeepCom receives a better score because it combines source code and traversed AST sequences to generate summary information. However, Hybrid-DeepCom uses LSTM to model the whole syntax tree, which is prone to the gradient disappearance problem. In addition, Hybrid-DeepCom applies SBT to capture the code structure information, but the encoded information cannot be fully utilized in the decoding stage. Specifically, DRL+HAN obtains scores of 37.15%, 21.54% and 34.83%. DRL+HAN incorporates information about the control flow of the code, the AST, etc., and thus provides a more comprehensive representation of the code structure. GT-SimNet is significantly higher than DRL+HAN. This is because we utilize Local-ADG to enhance the code semantic information. Moreover, GT-SimNet and DRL+HAN fuse code structures of different modalities in different ways. GT-SimNet relies on the similarity scores calculated from the similarity network for purposeful fusion, effectively eliminating redundant code information. In contrast, DRL+HAN performs stitching fusion, and the amount of code information is too mixed, so the quality of the generated summary is lower than that of GT-SimNet. In addition, since the code or AST is much longer than natural language, the AST-Transformer can help the model catch long-distance meaningful word or node pairs and then learn some characters related to the function of the code. The SeCNN utilizes the powerful convolutional power of a CNN to capture the code features of different modalities and thus scores better than the AST-Transformer. ComFormer performs the best among all the baselines, understandably, because ComFormer implements the method of fusion of different modal code features and uses a transformer for training. From the experimental data, GT-SimNet achieves better performance than the baselines. Because Local-ADG can effectively express the call constraint relationship between programs, GT-SimNet collates code information of different modalities by the similarity network, and the information received by the Transformer encoder contains richer code knowledge, so it works better.

For Dataset B, the baselines perform roughly similarly to Dataset A. From Section 4.1, we find that Dataset B contains 347,410 data pairs whose data volume is larger than that of Dataset A. Therefore, the baselines perform slightly better than on Dataset A. For BLEU, GT-SimNet outperforms the baseline methods by 2.68% to 14.03%; for METEOR by 2.48% to 9.87%; and for ROUGE-N by 0.76% to 10.36%. The comparison results show that the proposed GT-SimNet model extracts more comprehensive knowledge of code features and has excellent performance on different datasets.

Combining the results on both datasets, it is clear that GT-SimNet achieves the best performance in all evaluation metrics. The reason is that GT-SimNet effectively circles the API dependencies closely related to code snippets based on the Local-ADG, which shields the traditional ADG from containing external noise and thus can more accurately characterize the code features of a single snippet.

In summary, GT-SimNet achieves the best performance on two datasets, which validates the advantages of GT-SimNet for code summarization tasks.

5.2. Ablation experiments

We further conduct ablation experiments to verify the effects of the different components of the model. Data for the ablation experiments are provided by Dataset A. Fig. 6 shows the experimental results of the ablation experiments conducted on GT-SimNet. The right side of the graph represents the BLEU score after a component has been eliminated from GT-SimNet, and the left side represents the difference between before and after this elimination.

Analysis of code structure. We perform the ablation analysis one at a time using one of the semantic and syntactic structures, with the other conditions of the model held constant. We observe that both ADG and AST contribute to the model performance. For example, the BLEU score drops by approximately 7% after removing the ADG, which indicates the importance of the ADG to the model. In addition, we replace Local-ADG with Global-ADG, and we find that the BLEU score is reduced by approximately 4%, which is because Global-ADG constructs the ADG of the target code snippet from the perspective of the dataset and mixes a considerable amount of semantic feature information that is not part of the target code snippet. Local-ADG focuses only on the internal semantic information of the target code snippet and thus has a purer reflection of the code semantics.

Analysis of SimNet. We use splicing instead of the SimNet fusion network. We observe that the BLEU score of GT-SimNet decreases without the SimNet fusion network, which indicates that SimNet is effective in fusing the semantic and syntactic structures of the codes. Moreover, SimNet serves as a fusion of

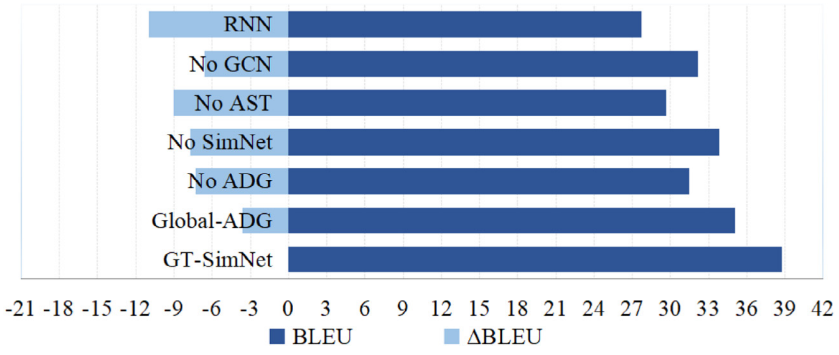


Fig. 6. Experimental results of ablation experiments.

the Local-ADG and AST. Codes are hybrid structures of semantics and syntax. Semantics rely on different syntactic structures, and syntactic structures are reflected by semantic structures. SimNet can effectively reflect the semantic and syntactic code correlations and fuse them effectively. Therefore, SimNet is important for the overall characterization of the code.

Analysis of other factors. As shown in Fig. 6, we observe that the ablation result of the GCN has a greater impact on GT-SimNet, which is not difficult to explain; the GCN has a powerful convolutional ability to aggregate our graph structure. Dropping the GCN leads to discrete information in the code structure and cannot represent the overall code's structure. Finally, if Transformer is replaced with a conventional encoder (RNN Sutskever et al., 2014), the performance of GT-SimNet decreases by nearly 11%, which validates the advantage of Transformer in contending with long dependency problems and vector fusion.

We also implemented ablation experiments using Dataset B and found that the overall statistical trend of Dataset B is basically the same as that of Dataset A. This also validates the advantages of each component of GT-SimNet.

5.3. Analyse the time efficiency of each component of GT-SimNet

We also note the efficiency of training each component of GT-SimNet. Specifically, we choose 20 epochs to train AST and Local-ADG separately to observe the time distribution in each epoch. From the results, the average time spent per epoch is 40 min for AST, 43 min for Local-ADG, and 56 min for AST and Local-ADG together. The training time for all components is less than 1 h, which is reasonable in real-world settings.

In addition, we find that the GT-SimNet performance shows an increasing trend as the number of total training epochs increases, and then the performance stabilizes after 50 epochs. Therefore, we choose 50 as the total number of epochs in this paper.

5.4. Influence of different lengths of source code and natural language summary length on GT-SimNet

Figs. 7 and 8 show the experimental comparison results of GT-SimNet with the baselines at different source code lengths and natural language lengths, respectively. We find that GT-SimNet and the baselines show a trend of low scores under three evaluation metrics as the source code lengths and natural language lengths increase because as the lengths of the code and natural language increase, the sizes of the sequences, trees, and graphs increase. Due to the limitations of the model feature extraction capabilities and vector dimensionality, the code feature information is lost to a certain extent, and prediction becomes more difficult.

In Fig. 7, we can observe that as the code lengths increase, the performance of GT-SimNet is relatively robust compared to the rest of the baselines. For BLEU, GT-SimNet outperforms the baselines by 3%∼16% when the code length is 90. CODE-NN performs the worst of all baselines. Similarly, Hybrid-DeepCom shows a decrease as the code length increases. This is because the AST size grows rapidly with increasing source code length and its modelling capabilities are not strong, leading to the loss of more code information. DRL +HAN achieves better results, and we speculate that this occurs because it uses tokens, an AST, and control flow for code feature representation, and the code feature loss rate is relatively low as the source code length increases due to the complementary relationship formed by multi-modal features. In contrast, GT-SimNet obtains better experimental results. Because the DRL+HAN approach uses a splicing method to fuse tokens, an AST, and control flow, the correlation between the different modal code features is not considered. When the generated code size increases, the code features are cluttered. GT-SimNet uses the SimNet network to identify the code syntax and semantic feature correlations and fuse them effectively. In addition, we exploit the call relationships between APIs to characterize the overall skeleton of the code snippet, allowing GT-SimNet to deal with longer source code.

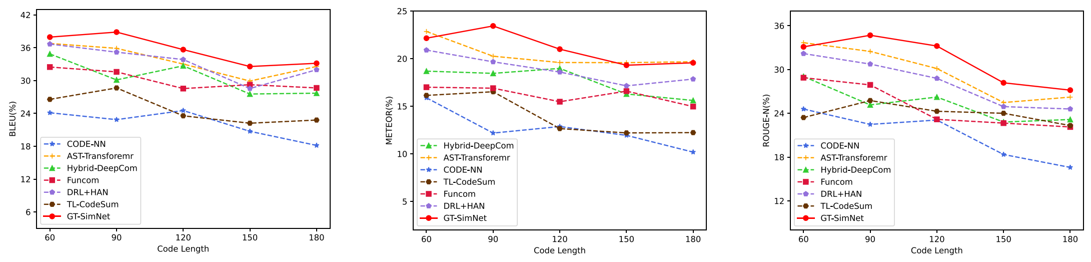
For different lengths of natural language summaries, GT-SimNet achieves the best performance, as shown in Fig. 8. We find that CODE-NN performs best at summary length of 10. DRL +HAN achieves the best results at a summary length of 15. The AST-Transformer performs the best when the summary length increases, which may be due to the use of the Transformer as the summary output model. Similarly, GT-SimNet also makes use of the Transformer so that it can handle longer summaries.

In summary, the red curve (GT-SimNet) is higher than the other curves, as shown in Figs. 7 and 8, indicating that as the code length and natural language summarization length vary, GT-SimNet yields results superior to those of the baselines.

5.5. Case study

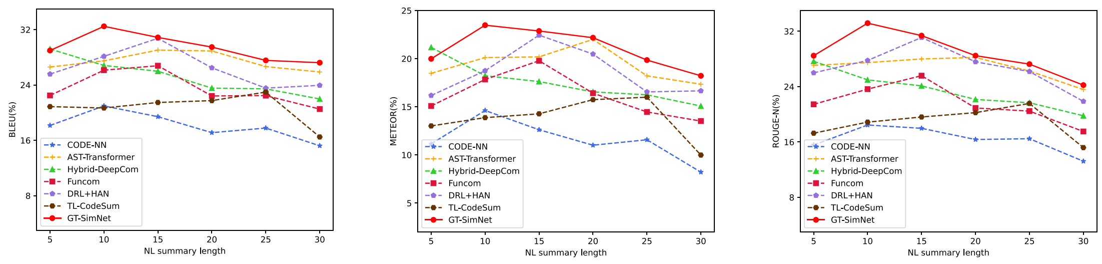
To qualitatively analyse our model, we select two examples from datasets and show the summarization generated by several baselines, as shown in Fig. 9. For more cases, please see Appendix.

Case 1 is a Java code snippet selected from Dataset A. GT-SimNet generates a summary for this code as *"Converts the AST to ASCII and generates that string representation"*. This is closest to the ground truth, which shows that GT-SimNet can generate semantically similar natural language summaries from the Local-ADG. The CODE-NN model is a variation in the machine translation approach. Its summary is actually a translation of key code statements and does not achieve the desired effect. TL-CodeSum performs code summarization with the API information, but the representation of code syntax information is incomplete, while both Hybrid-DeepCom and AST-Transformer deviate from the ground truth.



(a) BLEU (b) METEOR (c) ROUGE-N

Fig. 7. Experimental comparison results of GT-SimNet and baselines at different source code lengths. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)



(a) BLEU (b) METEOR (c) ROUGE-N

Fig. 8. Experimental comparison results of GT-SimNet and baselines at different natural language summary lengths. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Case 1

```
public String showAsString(AST ast, String header) {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    PrintStream ps = new PrintStream( baos );
    ps.println( header );
    showAst( ast, ps );
    ps.flush();
    return new String( baos.toByteArray() );}
```

Case 2

```
public Writer append (CharSequence csq, int start,
int end) throws IOException {
    if (null == csq) {
        write(TOKEN_NULL.substring(start, end));}
    else {
        write(csq.subSequence(start, end).toString());}
    return this;}
```

Ground Truth	Renders the AST into 'ASCII art' form and returns that string representation.	Appends a subsequence of the character sequence to the address.
CODE-NN	Returns the AST to a new baos string representation.	Writes the subsequence of csq to the beginning and the end of the table.
Hybrid-DeepCom	Generate a string representation of the AST by ASCII.	Writes a subsequence of csq to a sequence.
TL-CodeSum	Change the AST string to ASCII and return the representation.	Move the subsequence of csq sequence to the table.
AST-Transformer	Generate ASCII of AST and use string to represent AST.	Adds the subsequence of csq to the target.
GT-SimNet	Converts the AST to ASCII and generates that string representation.	Writes a subsequence of the character sequence to the address.

Fig. 9. Comparison of the generated results of GT-SimNet and several baselines.

Case 2 is an example from Dataset B, and GT-SimNet generates *"Writes a subsequence of the character sequence to the address"*. The expression is basically the same as the ground truth. In contrast, the other baselines all read out *"csq"*, but this term is not meaningful; the results of GT-SimNet show that it correctly understands the token in the code.

5.6. Error analysis

To analyse the reasons for the errors, we randomly select 100 failure cases from the two datasets. We analyse these cases and find two types of error causes:

- **Natural language understanding errors.** These errors are reflected in grammatical errors and deficiencies as well as semantic ambiguity. We evaluate the code snippets corresponding to these failures. We find that they do not follow the common conventions for naming identifiers and function names, resulting in unclear or confusing semantics, which can cause failures. Therefore, the effective interpretation of identifiers is still challenging, and we will focus on this issue in our future research.
- **Out-of-vocabulary errors.** The OOV problem refers to words that do not appear in the existing training corpus; data processing of corpus sets usually generates fixed-size word lists, which makes it impossible to generate words that do not appear in the word lists. Traditional solutions include identifying words outside the word list using the <UNK> sign, increasing the word list size, or ignoring them. Subword-based approaches use FastText and BPE tricks to solve the OOV problem by using word splitting or BPE segmentation. A possible solution is to copy OOV words into the output via a pointer network.

6. Human evaluation

To evaluate the effectiveness of the generated natural language summaries in practical applications, we conduct manual

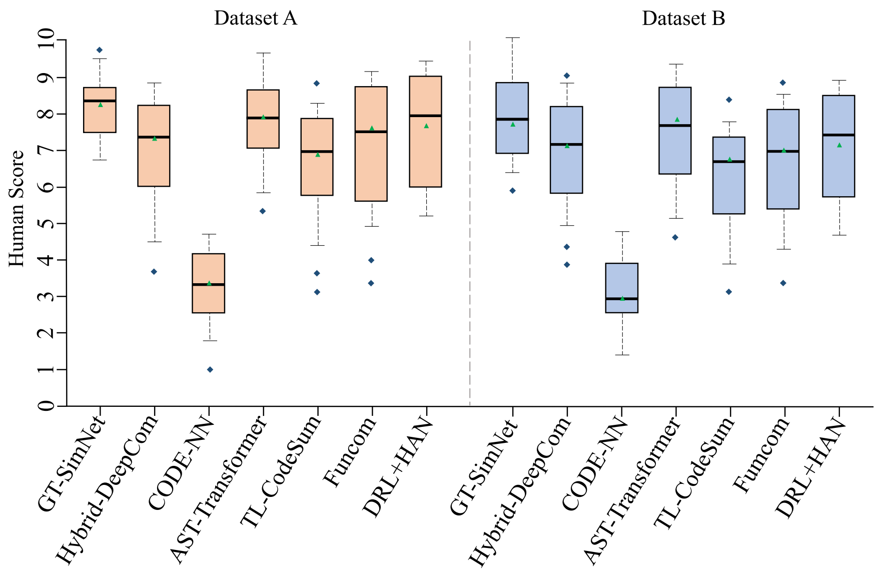


Fig. 10. Human evaluation results of GT-SimNet vs. baseline.

experiments to evaluate the quality of the natural language summaries. We invite 10 Ph.D. students and 5 programmers to participate in these manual experiments. Of these participants, the 10 Ph.D. students are from Shandong Normal University, majoring in computer science, and the five programmers are from Inspur Software Company. The hired evaluators all have more than 8 years of experience in programming. To eliminate as much bias as possible, none of the evaluators hired were emotionally or profitably related to us. During the evaluation process, we shield the evaluators from baseline information. The evaluators will not know which baseline the summary falls under.

6.1. Evaluation process

We divide the 15 evaluators into 5 groups, each having 1 programmer and 2 Ph.D. students. We randomly select 60 and 120 code snippets from Dataset A and Dataset B, respectively. Each evaluator scores the natural language summaries generated by the different code summarization methods. The scoring range is 0∼10. A score of 0 points indicates that the generated natural language summary is not related to the corresponding code snippet. In contrast, 10 points indicates that the generated natural language summary fully reflects the code functionality and has a complete and smooth semantic structure. To demonstrate the experiment's validity, real natural language summaries with corresponding code snippets are provided to each evaluator. Evaluators combine specific code snippets with the ground truth for scoring. In addition, we encourage communication within the group and the use of the internet to look up relevant content, but communication between each group is not allowed.

6.2. Results

For each natural language summary, we take the average score of each group of evaluators as the final evaluation result. The scores of the five groups of evaluators are collated and analysed. We draw box-line plots to represent the manual assessment results, as shown in Fig. 10.

Fig. 10 shows the GT-SimNet and baseline evaluations. Overall, the median line of Dataset A is higher than that of the Dataset B box plot, which indicates that natural language descriptions of higher average quality are generated for Dataset A. In the Dataset A box plot, the mean and median GT-SimNet scores are higher than the baselines, and the scores are more concentrated. This indicates that the evaluators' opinions on GT-SimNet are more concentrated and that the level of generated code summaries is more stable. Compared to GT-SimNet, the box plot scores of Funcom are unevenly distributed, which indicates that the level of summaries generated for Funcom is uneven. CODE-NN has the lowest distribution of box plot scores and is concentrated from 2∼5, which demonstrates the poor natural language summary of CODE-NN in the eyes of the evaluators.

In the Dataset B box plot, we find that GT-SimNet is slightly lower than the average score of the AST-Transformer and that the median line is in a flat state. The CODE-NN method still achieves a low score. In addition, GT-SimNet generates a lower percentage of low-quality summaries than Hybrid-DeepCom. We find that the Dataset A box plots of DRL+HAN do not change much from the Dataset B box plots. Finally, the TL-CodeSum results are slightly better in the Dataset A box plot than in the Dataset B box plot. This is probably due to the tighter API relationship in Dataset A. In summary, GT-SimNet generates the highest quality natural language descriptions on average in the eyes of the evaluators.

Specifically, we find that the box plots of each method show different view distributions and significant height differences, and we analyse the factors affecting these height differences. On the one hand, the self-defined method names in the code have some randomness, which affects the output of the natural language summary to some extent. On the other hand, due to the ambiguity of natural language semantics, the natural language results generated by the model can be interpreted differently by evaluators.

We also consider the final GT-SimNet and baseline scores and find that the p values of GT-SimNet are less than the significance level of 0.04 compared to baselines according to the Wilcoxon signed-rank test at the 95% confidence level. Therefore, we believe that the quality of the natural language generated by GT-SimNet is higher than those of the other comparison methods in human evaluation.

To assess the consistency of the manual experiments, we aggregated the kappa coefficients of five groups of evaluators. The kappa coefficients of the five groups of evaluators are calculated to be distributed in the range of 0.66 to 0.74. The scores rated by the evaluators were broadly consistent according to the classification of kappa coefficients. This can be used for experimental analysis.

ID	Example	

1	**Code**	`void assignToBlock(BlockId blk){`
		`     internalLock.writeLock().lock();`
		`     try{`
		`          flush();`
		`          this.blk=blk;`
		`          contents.read(blk);`
		`          pins=0;`
		`          lastLsn=LogSeqNum.readFromPage(contents,LAST_LSN_OFFSET);}`
		`     finally{`
		`          internalLock.writeLock().unlock();}}`

	Comments	**Ground Truth:** Reads the contents of the specified block into the buffer's page. If the buffer was dirty, then the contents of the previous page are first written to disk. **AST-Transformer:** Check if the buffer is dirty, write the content of the specified block to the buffer if it is dirty, otherwise write the content of the previous page to the disk. **SeCNN:** Writes the contents of the specified block to the buffer, or to disk if the buffer is dirty. **GT-SimNet:** Read the content of the specified block to the buffer page, if the buffer is dirty write the content of the previous page to disk first.

2	**Code**	`function playerMakeBet(uint minRollLimit, uint maxRollLimit,`
		` bytes32 diceRollHash, uint8 v, bytes32 r, bytes32 s) public payable`
		` gameIsActive betIsValid(msg.value, minRollLimit, maxRollLimit) {`
		`  if (playerBetDiceRollHash[diceRollHash] != 0x0`
		`   || diceRollHash == 0x0) throw;`
		`  tempBetHash = sha256(diceRollHash, byte(minRollLimit),`
		`   byte(maxRollLimit), msg.sender);`
		`  if (casino != ecrecover(tempBetHash, v, r, s)) throw;`
		`  . . .`
		`  playerProfifit[diceRollHash] = getProfifit(msg.value, tempFullprofifit);`
		`  if (playerProfifit[diceRollHash] > maxProfifit) throw;`
		`  . . .`
		`  LogBet(diceRollHash, playerAddress[diceRollHash],`
		`   playerProfifit[diceRollHash], playerToJackpot[diceRollHash],`
		`   playerBetValue[diceRollHash], playerMinRollLimit[diceRollHash],`
		`   playerMaxRollLimit[diceRollHash]);`
		`}`

	Comments	**Ground Truth:** Public function player submit bet only if game is active bet is valid. **AST-Transformer:** Players can submit bets only when the game is active. **SeCNN:** Check if the game is running and submit a bet. **GT-SimNet:** The bets submitted by the function player are valid only while the game is running.

3	**Code**	`public static String signature(Constructor<?> constructor) {`
		`    StringBuilder builder = new StringBuilder();`
		`    visitFormalTypeParameters(builder, constructor.getTypeParameters());`
		`    visitParameters(builder, constructor.getGenericParameterTypes());`
		`    builder.append("V");`
		`    visitExceptions(builder, constructor.getGenericExceptionTypes());`
		`    return builder.toString();`
		`}`

	Comments	**Ground Truth:** Generates the signature for the given constructor. **AST-Transformer:** Return the constructor signature for the given constructor. **SeCNN:** Generate markers for the constructor. **GT-SimNet:** Generate a signature for the given constructor.

7. Threats to validity

The following factors pose a threat to the validity of the model.

Multi-View evaluation metrics. The accurate assessment of the quality of the generation of automatic code summarization is an open problem. Most existing code summarization methods measure the similarity between statements through machine translation and text summarization evaluation metrics (e.g., BLEU). This paper uses BLEU, METEOR, and ROUGE to comprehensively evaluate the similarity between the generated summaries and the reference summaries. In the future, we plan to evaluate the summarization generation performance from more perspectives.

Quality of the dataset. The quality and representativeness of the dataset poses a major validity threat. In this paper, the datasets we use may contain some mismatched summaries. In the future, we will build a better-quality parallel corpus. In addition, we conduct code summarization experiments using two large datasets, but the generalization to other programming languages and the application of larger datasets are not tested; however, the results on the existing datasets in both programming languages demonstrate the validity of our model. In the future, we plan to extend our model to study larger datasets and multiple types of programming languages.

Deviation of the code summarization model. Parameter biases that arise when we operate on other code summarization models can also pose validity threats. The parameters in the source code of these models may not be suitable for our dataset. We perform an iterative parametric tuning exercise. This effect is reduced to a minimum level.

Data splitting strategy. The splitting strategy of the dataset has a significant impact on the evaluation of the code summarization model. It is known that many studies exist that divide the dataset arbitrarily (Tao et al., 2021). Generally, in the data splitting strategy, we can divide each programming language's dataset for training (80%), validation (10%), and testing (10%) in chronological order. In addition, k-fold cross-validation is also a common method of data segmentation (used in this paper). In the future, we can find various ways to split datasets to alleviate this threat.

8. Conclusion

In this paper, we proposed a new code summarization method, GT-SimNet, to generate natural language descriptions of target code snippets, which innovatively proposes a Local-ADG semantic structure model and jointly models it with AST. In the fusion model, we proposed a multi-modal similarity network to compute the similarity of two code structures and fused them. We conducted code summarization experiments on two datasets (Java dataset and Solidity dataset), and the results show that our model outperforms existing methods and yields better performance on longer code snippets.

While GT-SimNet's performance is better than that of the existing baselines, there are still areas for improvement. In the future, we will further improve the accuracy of the model and apply it to multiple types of datasets. In addition, we plan to visualize the code structure and attempt to train jointly with the code generation task to extend and improve our model.

CRediT authorship contribution statement

Xuejian Gao: Methodology, Writing – original draft. **Xue Jiang:** Conceptualization, Data curation. **Qiong Wu:** Formal analysis, Investigation. **Xiao Wang:** Investigation, Visualization. **Chen Lyu:** Supervision, Writing – review & editing. **Lei Lyu:** Validation, Resources.

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.

Acknowledgement

This work is financially supported by the Natural Science Foundation of Shandong Province, China (ZR2021MF059, ZR2019MF071), the National Natural Science Foundation of China (61602286, 61976127), the Special Project on Innovative Methods (2020IM020100), the Postdoctoral Innovation Program of Shandong Province (202103014) and the China Postdoctoral Science Foundation (2021M691978).

Appendix. More examples of comments generated by GT-SimNet in case study

References

Ahmad, Wasi Uddin, 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.

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

Allamanis, Miltiadis, Brockschmidt, Marc, Khademi, Mahmoud, 2018b. Learning to represent programs with graphs. In: International Conference on Learning Representations. ICLR.

Allamanis, Miltiadis, Peng, Hao, Sutton, Charles, 2016. A convolutional attention network for extreme summarization of source code. In: Proceedings of the 33nd International Conference on Machine Learning, (ICML). Vol. 48. pp. 2091–2100.

Alomari, H.W., Stephan, M., 2022. Clone detection through srcclone: A program slicing based approach. J. Syst. Softw. 184, 111115.

Alon, Uri, Brody, Shaked, Levy, Omer, Yahav, Eran, 2019. code2seq: Generating sequences from structured representations of code. In: International Conference on Learning Representations.

Banerjee, Satanjeev, Lavie, Alon, 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In: Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization. pp. 65–72.

Blasi, A., Stulova, N., Gorla, A., Nierstrasz, O., 2021. Replicomment: identifying clones in code comments. J. Syst. Softw. 182, 111069.

Bruna, Joan, Zaremba, Wojciech, Szlam, Arthur, LeCun, Yann, 2013. Spectral networks and locally connected networks on graphs. In: International Conference on Learning Representations.

Chen, Minghao, Wan, Xiaojun, 2019. Neural comment generation for source code with auxiliary code classification task. In: Asia-Pacific Software Engineering Conference. pp. 522–529.

Chen, Qiuyuan, Xia, Xin, Hu, Han, Lo, David, Li, Shanping, 2021. Why my code summarization model does not work: Code comment improvement with category prediction. ACM Trans. Softw. Eng. Methodol. 30 (2), 25:1–25:29.

Desai, Utkarsh, Sridhara, Giriprasad, Tamilselvam, Srikanth, 2021. Advances in code summarization. In: International Conference on Software Engineering (ICSE). pp. 330–331.

Fang, Liu, Ge, Li, Xing, Hu, Zhi, Jin, 2019. Program comprehension based on deep learning. J. Comput. Res. Dev. 56 (8), 1605.

He, Hao, 2019. Understanding source code comments at large-scale. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE). pp. 1217–1219.

Hill, Emily, Pollock, Lori, Vijay-Shanker, K., 2009. Automatically capturing source code context of nl-queries for software maintenance and reuse. In: International Conference on Software Engineering (ICSE). pp. 232–242.

Hu, Xing, Li, Ge, Xia, Xin, Lo, David, Jin, Zhi, 2018. Deep code comment generation. In: Proceedings of the 26th Conference on Program Comprehension (ICPC). pp. 200–20010.

Hu, Xing, Li, Ge, Xia, Xin, Lo, David, Lu, Shuai, Jin, Zhi, 2018. Summarizing source code with transferred api knowledge. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI). pp. 2269–2275.

Hu, X., Li, G., Xia, X., et al., 2020. Deep code comment generation with hybrid lexical and syntactical information. Empir. Softw. Eng. 25 (3), 2179–2217.

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.

Jiang, Siyuan, McMillan, Collin, 2017. Towards automatic generation of short summaries of commits. In: Proceedings of the 25th Conference on Program Comprehension (ICPC). pp. 320–323.

Kadar, Rozita, Syed-Mohamad, Sharifah Mashita, 2015. Semantic-based extraction approach for generating source code summary towards program comprehension. In: Malaysian Software Engineering Conference (MySEC). pp. 129–134.

Karmakar, Priyabrata, Teng, Shyh Wei, Lu, Guojun, 2021. Thank you for attention: A survey on attention-based artificial neural networks for automatic speech recognition. arXiv preprint arXiv:2102.07259.

Kipf, Thomas N., Welling, Max, 2016. Variational graph auto-encoders. arXiv preprint arXiv:1611.07308.

Krizhevsky, Alex, Sutskever, Ilya, Hinton, Geoffrey E., 2012. Imagenet classification with deep convolutional neural networks. Adv. Neural Inf. Process. Syst. 25, 1097–1105.

LeClair, A., Bansal, A., McMillan, C., 2021. Ensemble models for neural source code summarization of subroutines. In: International Conference on Software Maintenance and Evolution. pp. 286–297.

LeClair, Alexander, Haque, Sakib, Wu, Lingfei, McMillan, Collin, 2020. Improved code summarization via a graph neural network. In: Proceedings of the 28th Conference on Program Comprehension (ICPC). pp. 184–195.

LeClair, A., Jiang, S., McMillan, C., 2019. A neural model for generating natural language summaries of program subroutines. In: International Conference on Software Engineering (ICSE). pp. 795–806.

Li, Z., Wu, Y., Peng, B., et al., 2021. SeCNN: A semantic CNN parser for code comment generation. J. Syst. Softw. 181, 111036.

Li, Yaguang, Yu, Rose, Shahabi, Cyrus, Liu, Yan, 2017. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence. pp. 3634–3640.

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

Lin, Chen, Ouyang, Zhichao, Zhuang, Junqing, Chen, Jianqiang, Li, Hui, Wu, Rongxin, 2021a. Improving code summarization with block-wise abstract syntax tree splitting. In: Proceedings of the 29th Conference on Program Comprehension (ICPC). pp. 184–195.

Lin, C., Ouyang, Z., Zhuang, J., et al., 2021b. Improving code summarization with block-wise abstract syntax tree splitting. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC). IEEE, pp. 184–195.

Ling, Wang, Grefenstette, Edward, Hermann, Karl Moritz, Kočisk'y, Tomáš, Senior, Andrew, Wang, Fumin, Blunsom, Phil, 2016. Latent predictor networks for code generation. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics..

Luo, C., Zhan, J., Xue, X., Lei, W., Rui, R., Qiang, Y., 2018. Cosine normalization: Using cosine similarity instead of dot product in neural networks. Artif. Neural Netw. Mach. Learn. 11139, 382–391.

Luong, Minh-Thang, Pham, Hieu, Manning, Christopher D., 2015. Effective approaches to attention-based neural machine translation. In: Proceedings of the 2015 Conference on Empirical Methods in NaturalLanguage Processing, (EMNLP). pp. 1412–1421.

Lyu, Chen, Wang, Ruyun, Zhang, Hongyu, Zhang, Hanwen, Hu, Songlin, 2021. Embedding API dependency graph for neural code generation. Empir. Softw. Eng 26 (4), 1–51.

Marcilio, D., Furia, C.A., Bonifacio, R., Pinto, G., 2020. Spongebugs: Automatically generating fix suggestions in response to static code analysis warnings. J. Syst. Softw. 168, 110671.

McBurney, Paul W., McMillan, Collin, 2015. Automatic source code summarization of context for java methods. IEEE Trans. Software Eng. 42 (2), 103–119.

Mou, Lili, Li, Ge, Jin, Zhi, Zhang, Lu, Wang, Tao, 2014. TBCNN: A tree-based convolutional neural network for programming language processing. arXiv preprint arXiv:1409.5718.

Pan, Shirui, Hu, Ruiqi, Long, Guodong, Jiang, Jing, Yao, Lina, Zhang, Chengqi, 2018. Adversarially regularized graph autoencoder for graph embedding. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, (IJCAI). pp. 2609–2615.

Papineni, Kishore, Roukos, Salim, Ward, Todd, Zhu, Wei-Jing, 2002. Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. pp. 311–318.

Shido, Yusuke, Kobayashi, Yasuaki, Yamamoto, Akihiro, Miyamoto, Atsushi, Matsumura, Tadayuki, 2019. Automatic source code summarization with extended tree-LSTM. In: International Joint Conference on Neural Networks. pp. 1–8.

Staudemeyer, Ralf C., Morris, Eric Rothstein, 2019. Understanding LSTM–a tutorial into long short-term memory recurrent neural networks. arXiv preprint arXiv:1909.09586.

Sui, Y., Cheng, X., Zhang, G., et al., 2020. Flow2Vec: value-flow-based precise code embedding. In: Proceedings of the ACM on Programming Languages (OOPSLA). pp. 1–27.

Sun, Zeyu, Zhu, Qihao, Xiong, Yingfei, Sun, Yican, Mou, Lili, Zhang, Lu, 2020. Treegen: A tree-based transformer architecture for code generation. In: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 34. pp. 8984-8991. (05).

Sutskever, Ilya, Vinyals, Oriol, Le, Quoc V., 2014. Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems Advances in Neural Information Processing Systems. pp. 3104–3112.

Tang, Z., Li, C., Ge, J., et al., 2021. AST-transformer: Encoding abstract syntax trees efficiently for code summarization. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 1193–1195.

Tao, W., Wang, Y., Shi, E., et al., 2021. On the evaluation of commit message generation models: An experimental study. In: IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 126–136.

Tu, Ke, Cui, Peng, Wang, Xiao, Yu, Philip S., Zhu, Wenwu, 2018. Deep recursive network embedding with regular equivalence. In: Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 2357–2366.

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

Wan, Y., Shu, J., Sui, Y., et al., 2019. Multi-modal attention network learning for semantic source code retrieval. In: 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 13–25.

Wang, Daixin, Cui, Peng, Zhu, Wenwu, 2016. Structural deep network embedding. In: Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 1225–1234.

Wang, Y., Shi, E., Du, L., et al., 2021. CoCoSum: Contextual code summarization with multi-relational graph neural network. arXiv preprint arXiv:2107.01933.

Wang, Wenhua, Zhang, Yuqun, Sui, Yulei, Wan, Yao, Zhao, Zhou, Wu, Jian, Yu, Philip, Xu, Guandong, 2020. Reinforcement-learning-guided source code summarization via hierarchical attention. IEEE Trans. Softw. Eng..

Xu, Kun, Wu, Lingfei, Wang, Zhiguo, Feng, Yansong, Sheinin, Vadim, 2018. Sql-to-text generation with graph-to-sequence model. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, (EMNLP). pp. 931–936.

Xuejian, G., Xue, J., Qiong, W., Xiao, W., Chen, L., Lei, L., 2021. Multi-modal code summarization fusing local API dependency graph and AST. In: International Conference on Neural Information Processing (ICONIP). pp. 290–297.

Yang, G., Chen, X., Cao, J., et al., 2021a. Comformer: Code comment generation via transformer and fusion method-based hybrid code representation. In: 2021 8th International Conference on Dependable Systems and their Applications (DSA). IEEE, pp. 30–41.

Yang, Z., Keung, J., Yu, X., et al., 2021b. A multi-modal transformer-based code summarization approach for smart contracts. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC). IEEE, pp. 1–12.

Yu, Bing, Yin, Haoteng, Zhu, Zhanxing, 2017. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. pp. 3634–3640.

Yu, Wenchao, Zheng, Cheng, Cheng, Wei, Aggarwal, Charu C., Song, Dongjin, Zong, Bo, Chen, Haifeng, Wang, Wei, 2018. Learning deep network representations with adversarially regularized autoencoders. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. pp. 2663–2671.

Zhang, J., Wang, X., Zhang, H., et al., 2019. A novel neural source code representation based on abstract syntax tree. In: Proceedings of the 41st International Conference on Software Engineering, (ICSE). pp. 783–794.

Zhang, Z., Wang, X., Zhu, W., 2021. Automated machine learning on graphs: a survey. In: Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence. pp. 4704–4712.

Zhuang, Y., Liu, Z., Qian, P., et al., 2020. Smart contract vulnerability detection using graph neural network. In: Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence (IJCAI). pp. 3283–3290.

Xuejian Gao is currently studying for a master's degree at the School of Information Science and Engineering, Shandong Normal University, Jinan, China, under the guidance of Chen Lyu. His research interests include natural language processing, code analysis and artificial intelligence

Xue Jiang is with the School of Information Science and Engineering, Shandong Normal University, Jinan, China, under the guidance of Chen Lyu. Her research interests include program comprehension, automatic program summarization and component based software development.

Qiong Wu is with the School of Information Science and Engineering, Shandong Normal University, Jinan, China, under the guidance of Chen Lyu. Her research interests include code analysis and artificial intelligence.

Xiao Wang is currently studying for a master's degree at the School of Information Science and Engineering, Shandong Normal University, Jinan, China, under the guidance of Chen Lyu. Her research interests include natural language processing, code analysis and artificial intelligence.

Chen Lyu received his Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2015. He is currently an associate professor with the School of Information Science and Engineering, Shandong Normal University, Jinan, China. His research interests include program comprehension, software maintenance and evolution, and source code summarization.

Lei Lyu received his Ph.D. degree in computer application technology from the University of Chinese Academy of Science in 2013. He is currently an associate professor with the School of Information Science and Engineering, Shandong Normal University, Jinan, China. His current research interests include software engineering and Programming languages, including automated software analysis and software evolution.