Heterogeneous Directed Hypergraph Neural Network over abstract syntax tree (AST) for Code Classification

Guang Yang*, Tiancheng Jin*, Liang Dou*†(✉)
* School of Computer Science and Technology, East China Normal University, Shanghai, China
† NPPA Key Laboratory of Publishing Integration Development, ECNUP, Shanghai, China
51215901104@stu.ecnu.edu.cn, 51184506019@stu.ecnu.edu.cn, ldou@cs.ecnu.edu.cn

***Abstract*—Code classification is a difficult issue in program understanding and automatic coding. Due to the elusive syntax and complicated semantics in programs, most existing studies use techniques based on abstract syntax tree (AST) and graph neural network (GNN) to create code representations for code classification. These techniques utilize the structure and semantic information of the code, but they only take into account pairwise associations and neglect the high-order correlations that already exist between nodes in the AST, which may result in the loss of code structural information. On the other hand, while a general hypergraph can encode high-order data correlations, it is homogeneous and undirected which will result in a lack of semantic and structural information such as node types, edge types, and directions between child nodes and parent nodes when modeling AST. In this study, we propose to represent AST as a heterogeneous directed hypergraph (HDHG) and process the graph by heterogeneous directed hypergraph neural network (HDHGN) for code classification. Our method improves code understanding and can represent high-order data correlations beyond paired interactions. We assess heterogeneous directed hypergraph neural network (HDHGN) on public datasets of Python and Java programs. Our method outperforms previous AST-based and GNN-based methods, which demonstrates the capability of our model.**

***Index Terms*—hypergraph, heterogeneous graph, code classification, graph neural networks, program understanding**

I. Introduction

With the advancement of modern computer software, how to learn from vast open-source code repositories to enhance software development has become an essential research topic. In recent years, source code processing, which tries to help computers automatically comprehend and analyze source code, has received a lot of attention. Several works have been suggested including code classification [1]–[6], method name prediction [3] [4] [7] [8], code summarization [3] [9] [10] and code clone detection [5] [11] [12], etc.

Due to the improvement of machine learning technology, particularly deep learning, more and more work has employed deep learning for code classification. Currently, there are two main categories of code classification methods: AST-based and GNN-based. To take advantage of the semantic and structural information of the source code, several studies adopt AST when learning code representations [1] [5] [7] [8]. Some research uses graph neural networks (GNN) to create code representations for code categorization to a better understanding of the structure of code based on AST [3] [4] [13] [14].

```
a, b, c = input()
print(["No", "Yes"][a==c])
```

Fig. 1. An example of the code snippet. The program reads three inputs a, b, and c in turn, if a equals c, "Yes" will be output, otherwise, "No" will be output.

Although these AST-based and graph-based techniques employ the structural information of source code and demonstrate their effectiveness, there is a problem that they only take into the pairwise relationships and ignore the possible high-order correlations between AST nodes. For example, when code is parsed into an AST, each parent AST node has child AST nodes belonging to various fields or called attributes. A parent node may have several child nodes under the same field, and these nodes have high-order correlations with one another. Fig. 1 depicts an example of a python code snippet. The corresponding AST generated by the official python ast module[1] is illustrated in Fig. 2(a). As we can see, the "Module" is the root node of the AST. It has two child nodes, "Assign" and "Expr" which belong to the field named "body." When modeling the correlations between the three nodes, previous approaches only consider the pairwise relationships, i.e., the pair of "Module" and "Assign" and the pair of "Module" and "Expr," as demonstrated in Fig. 3(a). The high-order data correlation that "Assign" and "Expr" both belong to the "body" of "Module" as shown in Fig. 3(b) is dismissed may result in the loss of code structural information.

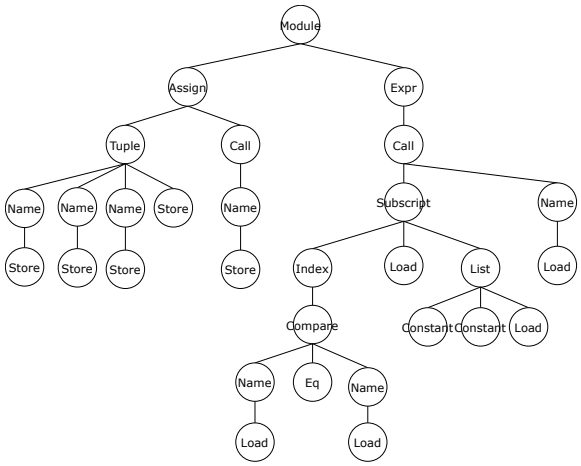
In recent years, hypergraph, which can encode high-order data correlations, has drawn a lot of interest. Considering the outstanding performance of hypergraph in graph classification [15] [16], we present hypergraph into code classification. On the other hand, a general hypergraph is homogeneous and undirected, i.e., it only has one type of node and one type of edge, and its hyperedge is undirected. If we represent the AST with a general hypergraph, it will result in lack of semantic

DOI reference number: 10.18293/SEKE2023-136

[1]https://docs.python.org/3/library/ast.html.

```
Module(body=[
  Assign(targets=[
    Tuple(elts=[
      Name(id='a',ctx =Store()),
      Name(id='b', ctx=Store()),
      Name(id='c',ctx =Store()),
    ],ctx =Store()),
  ], value=Call(func=Name(id='input',ctx =Load()), args=[], keywords=[])),
  Expr(value=Call(func=Name(id='print',ctx =Load()),args =[
    Subscript(value=List(elts=[
      Constant(value='No', kind=None),
      Constant(value='Yes', kind=None),
    ],ctx =Load()), slice=Index(value=Compare(left=Name(id='a', ctx=Load()), ops=[
      Eq(),
    ], comparators=[
      Name(id='c',ctx =Load()),
    ])),ctx =Load()),
  ], keywords=[])),
])
```

(a) The AST created by offical python module.



(b) The illustration of the AST.

Fig. 2. The AST of code snippet in Fig. 1. We use the official python module to print the AST to depict the details in Fig. 2(a). We draw the illustration of the AST in Fig. 2(b) to demonstrate the parent-child relationship between AST nodes.

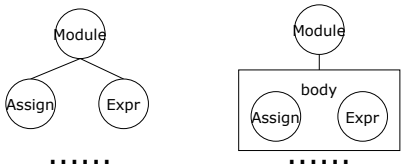
and structural information such as field names and directions. As illustrated in Fig. 4(a), a typical hypergraph does not have the field name and the direction to show who is the parent node and who is the child node.

To tackle the above problems, we suggest a heterogeneous directed hypergraph (HDHG) to model the AST and a HD-HGN for code classification. First, we propose to use heterogeneous directed hyperedge to show the relationship of AST nodes, the example of Fig. 3 is shown in Fig. 4(b). Second, we combine deep learning techniques from hypergraph neural networks and heterogeneous graph neural networks to create the HDHGN, and we also add operations to process directed hyperedge.

We evaluate our method on public datasets Python800 and Java250 [17]. Our model gets 97% in accuracy on Python800 and 96% on Java250 which outperforms previous state-of-the-art AST-based and GNN-based work. Our study demonstrates the utility of HDHG and HDHGN for code classification.

The main contributions of this paper are:

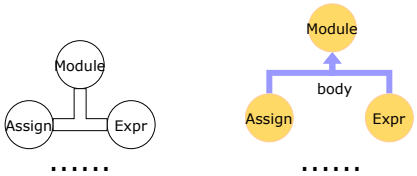
- We propose an HDHG to depict AST.
- We propose an HDHGN to generate vector representations for code classification.
- We assess our model on public datasets and compare it with previous SOTA AST-based and graph-based methods.



(a) The pairwise relationships.

(b) The high-order data correlation.

Fig. 3. The pairwise connections and high-order data correlation between three AST nodes. We ignore other AST nodes in the figure.



(a) The general hypergraph.

(b) The HDHG.

Fig. 4. Comparison of using general hypergraph and HDHG to model the relationships between three nodes of Fig. 3. The difference is that hyperedges in HDHG have direction and type.

II. Related Work

Code classification is to classify codes based on their functions. Different from natural language, code has structural information. As a result, several works adopt AST by various techniques. Mou et al. [1] is one of the first works to suggest a Tree-Based Convolutional Neural Network (TBCNN) in code classification. Alon et al. propose code2seq [7] and code2vec [8] to deconstruct code to a collection of paths in its AST. J. Zhang et al. [5] propose a novel neural called ASTNN for source code representation for code classification and clone detection. N. D. Q. Bui et al. [18] propose a novel method named TreeCaps by fusing capsule networks with TBCNN in code classification.

With the popularity of GNN, more works apply kinds of GNN in code classification based on AST to strengthen the comprehension of code structures. M. Allamanis et al. [13] first construct graphs from source code by adding edges like control flow and data flow to AST and employing a gated graph neural network (GGNN) to process program graphs. V. Hellendoorn et al. [19] propose a model called GREAT based

on the transformer architecture by extracting global relational information from code graphs. D. Vagavolu et al. [3] propose an approach that can extract and use program features from multiple code graphs. M. Lu et al. [6] improved GGNN in program classification. T. Long [14] proposes a multi-view graph program representation method that combines both data flow and control flow as multiple views and applies GNN to process. W. Wang et al. [4] propose to leverage heterogeneous graphs to show code based on previous work and adopt heterogeneous GNN to process.

III. Preliminary

In this section, we introduce some fundamental background ideas, such as hypergraph, heterogeneous graph, and AST.

A. Hypergraph

In an ordinary graph, an edge can only be connected with two vertices. Different from general graphs, the edge of a hypergraph [20] can link any number of vertices. Formally, a hypergraph H is a pair $H = (V, E)$ where V is a set of elements called nodes or vertices, and E is a set of non-empty subsets of V called hyperedges or links.

A directed hypergraph [21] is a hypergraph with directed hyperedges. A directed hyperedge or hyperarc is an ordered pair, $E = (X, Y)$, of (possibly empty) disjoint subsets of vertices; X is the tail of E while Y is its head. A backward hyperarc, or simply B-arc, is a hyperarc $E = (X, Y)$ with $|Y| = 1$. A forward hyperarc, or simply F-arc, is a hyperarc $E = (X, Y)$ with $|X| = 1$. A hypergraph whose hyperarcs are B-arcs is known as a B-graph (or B-hypergraph). A hypergraph whose hyperarcs are F-arcs is known as an F-graph or F-hypergraph.

In our study, since the child node of AST points to the parent node and the child node has only one parent node, our HDHG is a B-hypergraph.

B. Heterogeneous Graph

A heterogeneous graph [22] is a graph consisting of multiple types of entities or nodes and multiple types of links or edges. A heterogeneous graph is represented as $G = (V, E)$, consisting of an entity set V and a link set E. A heterogeneous graph is also correlated with a node type mapping function $\phi : V \to A$ and a link type mapping function $\psi : E \to R$. A and R represent the sets of predefined object types and link types, where $|A| + |R| > 2$.

C. Abstract Syntax Tree

The AST represents the source code's abstract syntax structure. The code compiler will parse the code into an AST through the program syntax and semantic analysis. Each node on the tree represents a structure in the source code and belongs to different AST node types. Each AST node has zero, one, or several fields that can be thought of as the node's attributes. Each field may have none, one, or a list of objects such as AST node, number, and string. If one AST node contains a field with a different AST node, and the latter is equivalent to the former's child AST node.

IV. Methodology

We first convert the code snippet into an AST and construct an HDHG based on it, then put it into our HDHGN. We combine the vector representations for code categorization once we get the network's node's vector representation. The overview of our model is demonstrated in Fig. 5.

A. Heterogeneous Directed Hypergraph

We parse the code snippet into an AST with a code compiler, then we develop the HDHG based on the AST. We set the node of AST as the "AST" node and the identifier of AST as the "identifier" node in HDHG. We set the value of the "AST" node as its AST node type name, set the value of the "identifier" node as its content, and treat them as two different types of nodes. The field is configured as a directed hyper edge. If one node has a field including another node, the latter node belongs to the tail of the field hyperedge, the former is the head of the field hyperedge. We designated the field name as the type of hyper edge. The illustration of the HDHG of AST in Fig. 2 is shown in Fig. 6.

B. Heterogeneous Directed Hypergraph Neural Network

1) Definition: We let a HDHG $G = (N, E)$, which includes a node set $N = \{n_1,\ n_2, \ldots, n_{|N|}\}$ and a directed hyperedge set $E = \{e_1, e_2, \ldots, e_{|E|}\}$. Each node $n = (\mu, x)$, where μ represents node type and x is the value of the node. Each directed hyperedge $e = (\rho, S(e), T(e))$, ρ represents edge type, $S(e) = \{n_1, \ldots, n_{|S(e)|}\} \subseteq N$ is the tail nodes of hyperedge e, $T(e) \in N$ is the head node of hyperedge e, they show the direction of the hyperedge e is $S(e)$ to $T(e)$.

2) Feature initialization: According to the value x and the category μ of node n, we obtain embedding vector $d_n \in \mathbb{R}^{C_1}$ by embedding function as (1), where C_1 is the dimension size of the embedding vector.

$$d_n = Embed_\mu(x) \tag{1}$$

To put embedding vectors of various types into the same vector space, we make a linear projection to obtain the initial feature vector $h_n^0 \in \mathbb{R}^{C_2}$ of node n based on the corresponding node type μ as (2), where C_2 is the dimension size of feature vector and hidden vector.

$$h_n^0 = W_\mu d_n + b_\mu \tag{2}$$

We also obtained embedding vector $d_e \in \mathbb{R}^{C_2}$ of hyperedge e according to the edge type ρ as (3).

$$d_e = Embed_{edge}(\rho) \tag{3}$$

3) Heterogeneous Directed Hypergraph Convolution Layer: Our model updates each node vector in one heterogeneous directed hypergraph convolution (HDHGConv) layer. We refer to the framework of two-stage message passing of hypergraph neural network [15] which has two steps: aggregating messages from nodes to hyperedges and aggregating messages from hyperedges to nodes. Contrarily, we add operations that add heterogeneous information and direction information.

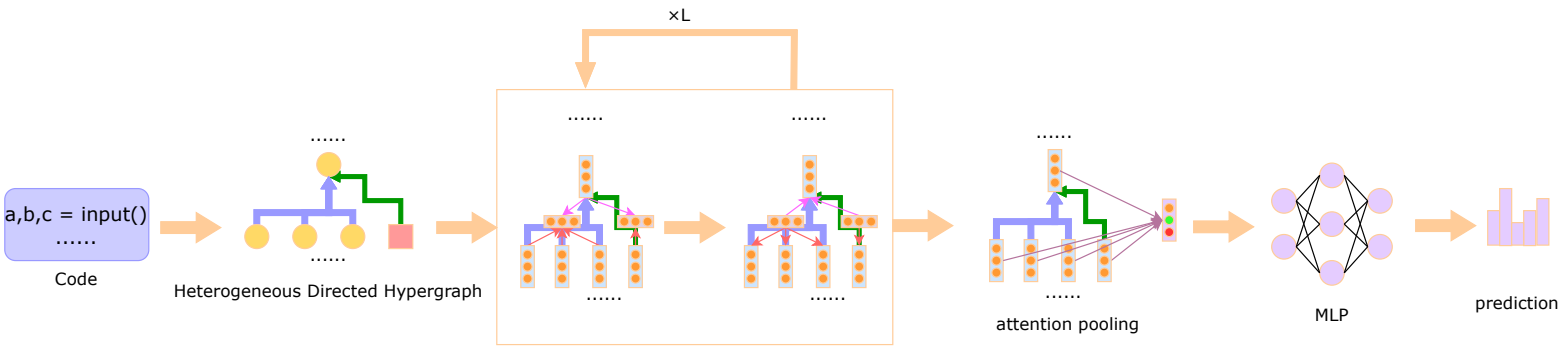


Fig. 5. Overview of the process.

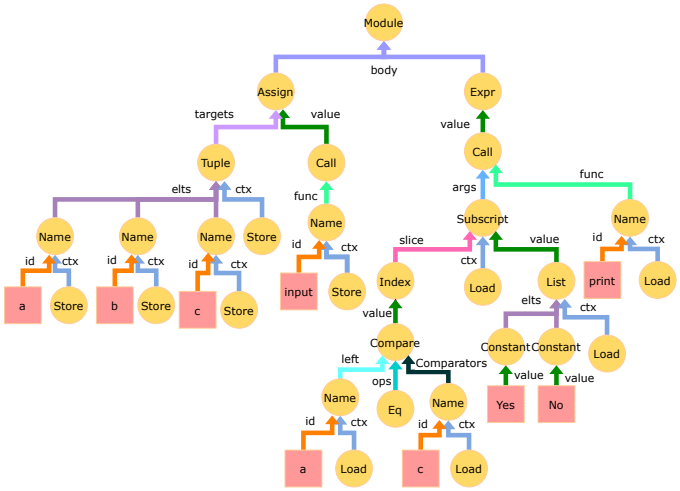


Fig. 6. The HDHG of AST in Fig. 2. The circular node is the "AST" node, and the square node is the "identifier" node. The edge can connect multiple nodes, the node connected with an arrow is the head node of the edge, and the node connected without the arrow is the tail node. Various hues correspond to various edge kinds. The node and edge both display the node value and edge type.

Aggregating messages from nodes to hyperedges: First, the hidden vector h_n^{l-1} of each node n is multiplied by the head matrix or tail matrix to get the message vector $m_{n\_e}^l$ from node n to hyperedge e as (4), where $l = 1, 2, \ldots, L$ indicate layer number, L is the total number of layers.

$$m_{n\_e}^l = \begin{cases} W_{head}^l h_n^{l-1} + b_{head}^l \, if \; n \in S(e) \\ W_{tail}^l h_n^{l-1} + b_{tail}^l \, if \; n = T(e) \end{cases} \tag{4}$$

Directed hyperedge e gathers message from their tail nodes and head node by transformer attention mechanism [23]. For each message, the attention score is formulated as (5), where d_e is the edge type vector.

$$\alpha_{n\_e}^l = Softmax\left(\frac{\left(W_{q1}^l d_e\right)^T W_{k1}^l m_{n\_e}^l}{\sqrt{C_2}}\right) \tag{5}$$

We obtain the vector o_e^l of directed hyperedge e as (6).

$$o_e^l = \sum_{n \in S(e) \; or \; n = T(e)} \alpha_{n\_e}^l W_{v1}^l m_{n\_e}^l \tag{6}$$

Then we add edge type vector to o_e^l as (7), where z_e^l is formed as (8).

$$q_e^l = o_e^l + z_e^l \tag{7}$$

$$z_e^l = W_z^l d_e + b_z^l \tag{8}$$

Aggregating messages from hyperedges to nodes: For each directed hyperedge e whose head node or tail node is n, the q_j^l will be linear projected by (9) to get message $m_{e\_n}^l$ which will be sent to n.

$$m_{e\_n}^l = \begin{cases} W_{to\_head}^l q_e^l + b_{to\_head}^l \, if \; n \in S(e) \\ W_{to\_tail}^l q_e^l + b_{to\_tail}^l \, if \; n = T(e) \end{cases} \tag{9}$$

Same as before, we aggregate messages to get v_n^l by the transformer attention mechanism.

$$\alpha_{e\_n}^l = Softmax\left(\frac{\left(W_{q2}^l h_n^{l-1}\right)^T W_{k2}^l m_{e\_n}^l}{\sqrt{C_2}}\right) \tag{10}$$

$$v_n^l = \sum_{T(e)=n \; or \; n \in S(e)} \alpha_{e\_n}^l W_{v2}^l m_{e\_n}^l \tag{11}$$

Last, we update hidden vector h_n^l of node n by (12), where σ is the elu activation function and GraphNorm is graph normalization [24].

$$h_n^l = \sigma\left(GraphNorm(W_{u1}^l v_n^l + W_{u2}^l h_n^{l-1} + b_u^l)\right) \tag{12}$$

The W above are all the weight matrix, and the b above are all bias vectors, which will be learned by training. All of the attention mechanisms mentioned above use multiple heads.

C. Classification

When we obtain the node hidden vectors $h_1^L, h_2^L, \ldots, h_{|N|}^L$ from the last layer, we utilize attention pooling to aggregate the information of each node to obtain vector representation r as (13)(14), where $g \in \mathbb{R}^{C_2}$ is a learnable vector.

$$\alpha_n = Softmax\left(g^T h_n^L\right) \tag{13}$$

$$r = \sum_{n \in N} \alpha_n h_n^L \tag{14}$$

To obtain the final classification prediction, we use an MLP, which is expressed as (15).

$$pred = Softmax\left(MLP(r)\right) \tag{15}$$

The attention mechanism above is also multi-head attention. We employ the standard cross-entropy loss function for the training.

V. Evaluation

We implement code by torch_geometric[2]. Our implementation is available on https://github.com/qiankunmu/HDHGN.

A. Datasets

We use Python800 and Java250 to train and assess our model. The two public datasets are from Project CodeNet [17] which are obtained from downloading submissions from two online judge websites: AIZU Online Judge and AtCoder. The code snippets are classified by the problem. The statistics of the datasets are depicted in Table I. To be clear, the AST node type means the AST type such as "Module" and "Assign," different from node types in HDHG, i.e., "AST" and "identifier." The edge type means the field name or called attribute in AST. We randomly split the dataset into the training set, validation set, and test set by 6:2:2.

B. Baselines

We compare our model with AST-based and GNN-based techniques which acquire the best performance in code classification including TBCNN [1], TreeCaps [18], GGNN [13], GREAT [19] and HPG+HGT [4]. TBCNN used a tree-based convolutional neural network to extract features from AST. A model called TreeCaps combines TBCNN and capsule networks. By adding edges like control flow and data flow to AST, the gated graph neural network (GGNN) processes graphs from source code. GREAT is a model extracting global relational information from code graphs based on the transformer architecture. A technique known as HPG+HGT uses a heterogeneous graph transformer to describe code as a heterogeneous graph. We also trained a GCN [25] and a GIN [26] in an experiment to compare.

[2]https://pytorch-geometric.readthedocs.io/en/latest/index.html

TABLE I
STATISTICS OF THE DATASETS

	Python800	Java250
Size	240000	75000
Labels	800	250
Avg. node	202.05	216.43
Avg. edge	187.25	198.51
AST node types	93	58
Edge types	58	61
Language	Python	Java

TABLE II
ACCURACY OF MODELS IN CODE CLASSIFICATION (IN %).

	Models	Python800 mean	Python800 sd	Java250 mean	Java250 sd
AST-based	TBCNN	91.16	±0.10	90.47	±0.10
	TreeCaps	90.33	±0.11	91.38	±0.11
GNN-based	GCN	91.94	±0.10	90.01	±0.10
	GIN	93.23	±0.10	90.72	±0.10
	GGNN	90.13	±0.11	89.97	±0.11
	GREAT	93.33	±0.12	93.17	±0.11
	HPG+HGT[4]	94.99	-	93.95	-
Ours	HDHGN	**97.87**	±0.10	**96.42**	±0.10

TABLE III
ACCURACY OF ABLATION STUDY ON PYTHON800 (IN %).

Variant	mean	sd
HDHGN	**97.87**	±0.10
- hyperedge	94.79	±0.11
- heterogeneous information	95.23	±0.11
- direction	95.49	±0.10

C. Experiment settings

We use a parser from the official Python 3.8 ast library and javalang library[3] to parse the code snippets into ASTs. The embedding vectors are produced by random initialization and learned via training. Our model's layer number was set to four. The hidden vector dimension size and embedding vector dimension size were both set to 128. We use a narrow multi-head attention [23] mechanism and set the number of heads to eight. We employed Adam optimizer with the learning rate of 5×10^{-5} to train our model. We set the dropout rate to 0.2. We optimized the hyper-parameters of other baselines for the validation set's greatest performance. The models were trained for 100 epochs and we saved the models which perform best in validation set.

D. Results

We use the performance of the model on the test set as the outcome. We select classification accuracy as the metric. We calculate the mean accuracy and standard deviation after five iterations of the experiment. The results are depicted in Table II. Our HDHGN outperforms other methods in both datasets. In Python800, our HDHGN is 2.88% higher than the best baseline. In Java250, our model outperforms baseline models by at least 2.47%. This demonstrates that our model utilizes the semantic and structural features of code AST more effectively than previous approaches.

E. Ablation study

We perform some ablation studies of our HDHGN on Python800. We take into account three variants as below.

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

[4]We use the outcomes that were reported in their research because the paper did not make their code publicly available.

1) - hyperedge: We eliminate hyperedges from our model, leaving only paired edges, or normal edges, in the graph. A few regular edges will develop from the initial hyperedge.

2) - heterogeneous information: We eliminate heterogeneous information from our model, which entails treating identifier nodes and AST nodes as a single type of node in the graph and eliminating the information about edge types.

3) - direction: We remove direction information in our model, this means that the hyperedge is not directed hyperedge, it does not differentiate the head nodes and tail nodes.

We also repeat the experiment five times and compute the mean accuracy and standard deviation. The outcomes are depicted in Table III. Removing hyperedge make the result decrease by 3.08%. This demonstrates that high-order data correlations between AST nodes in code are indeed useful for comprehending programs. The removal of heterogeneous information reduces the result by 2.64%. Heterogeneous information often contains a lot of semantic information, which is helpful for program understanding. Removing direction caused a drop of 2.38% on the result. The direction of the graph can help enhance the model and get structural information by indicating whether the nodes connected by hyperedges are parent nodes or child nodes. The above outcomes demonstrate that our model can obtain a better understanding of AST structure and acquire more precise results in code classification after considering high-order data correlations, heterogeneous information, and direction information.

VI. Conclusion

In this study, we propose an HDHGN for code classification. To possibly encode high-order data correlations between nodes in AST, we introduce the use of hypergraphs. Due to the general hypergraph being homogeneous and undirected which will result in a lack of semantic and structural information, we propose to represent AST as a heterogeneous directed hypergraph. We create an HDHGN accordingly to utilize high-order data correlation, heterogeneous information and direction information better than previous methods. We test our model using open Python and Java datasets, and we compare the results to the baselines developed using the SOTA AST and GNN. The experiment demonstrates that our HDHGN outperforms the baselines. Further ablation study describes that the HDHGN enhances the performance of code classification.

Presently, the hypergraph we produce is large and contains many nodes and edges. Future research will focus on ways to scale down hypergraphs for modeling AST and enhance the current hypergraph model to make it more effective at classifying codes.

Acknowledgment

This work was supported by the Open Research Fund of NPPA Key Laboratory of Publishing Integration Development, ECNUP.

References

[1] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *AAAI*, 2016.

[2] S. Gilda, "Source code classification using neural networks," *2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pp. 1–6, 2017.

[3] D. Vagavolu, K. C. Swarna, and S. Chimalakonda, "A mocktail of source code representations," in *ASE*, 2021.

[4] W. Wang, K. Zhang, G. Li, and Z. Jin, "Learning to represent programs with heterogeneous graphs," *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, pp. 378–389, 2022.

[5] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 783–794, 2019.

[6] M. Lu, Y. Wang, D. Tan, and L. Zhao, "Student program classification using gated graph attention neural network," *IEEE Access*, vol. 9, pp. 87857–87868, 2021.

[7] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *ICLR*, 2019.

[8] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1 – 29, 2019.

[9] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pp. 200–20010, 2018.

[10] S. Liu, Y. Chen, X. Xie, J. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid gnn," in *ICLR*, 2021.

[11] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," *29th International Conference on Software Engineering (ICSE'07)*, pp. 96–105, 2007.

[12] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *ASE*, 2016.

[13] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *ICLR*, 2018.

[14] T. Long, Y. Xie, X. Chen, W. Zhang, Q. Cao, and Y. Yu, "Multi-view graph representation for programming language processing: An investigation into algorithm detection," in *AAAI*, 2022.

[15] J. Huang and J. Yang, "Unignn: a unified framework for graph and hypergraph neural networks," in *IJCAI*, 2021.

[16] Y. Feng, H. You, Z. Zhang, R. Ji, and Y. Gao, "Hypergraph neural networks," in *AAAI Conference on Artificial Intelligence*, 2018.

[17] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," 2021.

[18] N. D. Q. Bui, Y. Yu, and L. Jiang, "Treecaps: Tree-based capsule networks for source code processing," in *AAAI*, 2021.

[19] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, "Global relational models of source code," in *ICLR*, 2020.

[20] C. Berge, "Graphs and hypergraphs," 1973.

[21] G. Gallo, G. Longo, and S. Pallottino, "Directed hypergraphs and applications," *Discret. Appl. Math.*, vol. 42, pp. 177–201, 1993.

[22] Y. Sun and J. Han, "Mining heterogeneous information networks: Principles and methodologies," in *Mining Heterogeneous Information Networks: Principles and Methodologies*, 2012.

[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*, pp. 5998–6008, 2017.

[24] T. Cai, S. Luo, K. Xu, D. He, T.-Y. Liu, and L. Wang, "Graphnorm: A principled approach to accelerating graph neural network training," in *ICML*, 2021.

[25] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *ICLR*, 2017.

[26] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?," in *ICLR*, 2019.