GSAGE2defect: An Improved Approach to Software Defect Prediction based on Inductive Graph Neural Network

Ju Ma
School of Computer Science and Information Engineering
Hubei University
Wuhan, China
mj_5265@qq.com

Yi-Yang Sun
School of Computer Science and Information Engineering
Hubei University
Wuhan, China
625824202@qq.com

Peng He
School of Cyber Science and Technology
Hubei University
Wuhan, China
penghe@hubu.edu.cn

Zhang-Fan Zeng
School of Computer Science and Information Engineering
Hubei University
Wuhan, China
zeng.zhangfan@hubu.edu.cn

Abstract*—*Graph neural network is an effective deep learning framework for learning graph data. Existing research has introduced different variants of graph neural networks into the field of software defects and has achieved promising results. However, the graph neural network model based on the previous research is essentially transductive, is applied to a single fixed graph, and often ignores the direction and weight of the edges when modeling the network. In practice, software systems are dynamically evolving. Furthermore, in software network modeling, the direction and weight of edges are factors that are worth considering. Based on an inductive graph neural network, we proposed an improved defect prediction method named GSAGE2defect. We first constructed the class dependency network of the program and then used node2vec for embedding learning to automatically obtain the structural features of the network. Then we combined the learned structural features with traditional software code features to initialize the properties of nodes in the class dependency network. Next, we fed the dependency network to GraphSAGE for a deeper class representation. Finally, we evaluated the proposed method based on eight open-source programs and demonstrated that GSAGE2defect achieves an average improvement of 2.09%-26.69% over state-of-the-art methods in terms of F-measure.

Keywords-component; Software defect prediction; Graph neural network; Network embedding; Class dependency network

I. INTRODUCTION

Graph-structured data are ubiquitous nowadays in many domains such as social networks, cybersecurity, and bio- and chemo-informatics [1]. Through the learning of graph data, many tasks in these domains have been successfully solved, such as recommendations in social networks, and drug discovery in biological information networks. In the field of software engineering, software systems can also be abstracted as a Class Dependency Graph (CDG) with classes as nodes and class dependencies as edges. Some researchers have confirmed that learning the CDG structural information by using complex network theory can effectively improve software defect prediction [2-4]. In recent years, researchers have begun to employ deep learning techniques to automatically encode the dependency graph structure into low-dimensional vector spaces to improve downstream software tasks [2].

Many high-performance graph neural networks have been proposed, in which node adjacency information and node attributes are combined to capture structural information well (e.g., [3,5-7]). It is not difficult to find that the Graph Neural Network (GNN) models, based on which previous studies have been carried out, are inherently transductive and have only been applied in settings with a single fixed graph. However, graph structure data in real scenarios are often dynamic. As we know, a successful software system usually undergoes multiple consecutive versions during its life cycle. In other words, the structure of a software system continues to evolve with requirements and other factors. For instance, the fixing of bugs and the updating of functions may result in the addition or deletion of classes. The transductive-based GNNs primarily generate node embeddings on fixed graphs, and for new nodes, they usually require relearning the new graph, even if it is just a small update. This gives rise to high costs. Moreover, representations for unseen nodes or entirely new graphs cannot be quickly generated.

Hamilton et al. [8] proposed a general framework, called GraphSAGE (SAmple and aggreGatE), for inductive node embedding. Instead of training a distinct embedding vector for each node, GraphSAGE trains a set of aggregator functions and generates node embeddings by applying the learned aggregation functions. Based on GraphSAGE ideas, Zhou et al. [10] attempted to learn node deeper representation scores in CDGs for key class identification tasks. Besides, in the proposed method, the authors also considered the influence of the direction and weight of the edges in the Class Dependency Network (CDN).

Inspired by the above-mentioned studies and by considering the direction and weight of edges for nodes' feature vector learning, we also attempted to apply an inductive GNN model to CDN and then used them for defect prediction. The main contributions are summarized as follows:

DOI reference number: 10.18293/SEKE23-068

- We introduce an inductive GNN model, called GraphSAGE, to learn the features of CDN nodes effectively.
- We proposed a new method named GSAGE2defect, which uses the network embedding technique node2vec to initialize the node attributes of CDNs, and uses the GraphSAGE model to further implement feature extraction to improve defect prediction.
- We validated the effectiveness of GSAGE2defect based on eight open-source projects, and the results indicated that the proposed method can improve defect prediction performances.

The remainder of this paper is organized as follows: The related work is introduced in Section 2. The proposed method is detailed in Section 3. The experimental setup and results analysis are presented in Section 4. The advantages and shortcomings of our work are discussed in Section 5. Finally, the conclusion and prospects are drawn in Section 6.

II. RELATED WORK

In recent years, deep learning has been utilized to mine nonlinear features in software source code, where capturing semantic information from Abstract Syntax Tree (AST) has attracted widespread attention. For example, Wang et al. [11] extracted the source code of AST, and then leveraged Deep Belief Network (DBN) to automatically learn the hidden semantic and syntactic features in the program for defect prediction. Li et al. [12] employed a Convolutional Neural Network (CNN) to extract the semantic information of ASTs and combined the learned features with traditional hand-crafted features to enhance the prediction performance. However, AST only encapsulates the abstract syntax structure of the source code and cannot represent the execution process of the program. Hence, Phan et al. [13] converted the source code into a program Control Flow Graphs (CFG), and tried to learn from CFG through the convolutional neural network. It is worth mentioning that AST and CFG only focus on the semantic and structural information inside each code file, thereby ignoring the macro-structural information between code files, such as the dependencies between classes. Thus, Qu et al. [2] adopted a network embedding technique (i.e., node2vec) to automatically learn the external structural information of the CDN. In this way, they could achieve good results.

With the development of GNNs, the graph embedding model can integrate node and edge attributes while learning network structure. Indeed in [3], the author enhanced the performance of software defect prediction by successfully managing to learn the network structural features of source codes by using a transductive graph convolution neural network (GCN) model. Nevertheless, software system is constantly evolving in the real world. The addition of new classes, the deletion of irrelevant classes, the update of functional classes, and the repair of error classes will give rise to the iterative evolution of software systems. The transductive graph neural network (e.g., GCN) needs to re-learn the entire graph when generating embeddings for new nodes in the graph, which will result in problems such as excessive computing costs and large space costs. Besides, because network dependence is usually directed and weighted, CDG should be a directed weighted graph. However, the authors in [3,4] regarded CDG as an undirected unweighted graph. Some network information is inevitably lost in this approach.

It is not difficult to find that most existing studies have the following limitations: (1) The work focuses on embedding nodes from a single fixed graph, while many real-world applications require the fast generation of embeddings for unseen nodes or entirely new (sub) graphs. (2) The weight and direction of the edges are not considered during software network modeling. In practice, the dependency between classes is not a bidirectional equivalent dependency, and the degree of dependency between classes also varies.

Given these, we introduced an inductive graph neural network, namely GraphSAGE, to learn the directed weighted CDN. Specifically, we constructed a GSAGE2defect model for learning the structural features of nodes in class dependency networks. Then we performed end-to-end learning based on the semantic and structural information of nodes, and ultimately applied the obtained features to improve defect predictive performance.

III. METHOD

The framework diagram of this research is shown in Figure 1, which mainly includes three parts: (1) Constructing a class dependency network, and then learning the node embedding by the node2vec method; (2) Initializing node attributes by combining learned node embedding and traditional hand-crafted metrics, then introducing GraphSAGE to learn network structural features; (3) Training a classifier for defect prediction.

A. Network Modeling and Embedding Learning

1) Class dependency network modeling

A Class Dependency Network (CDN) is a directed weighted network constructed according to the dependencies between class files. For object-oriented software, its class dependency network $CDN_P = (V, E)$,where V is the set of nodes, each node $v \in V$ represents a class or interface, E is the set of edges, representing the dependencies between classes or interfaces. Three main dependencies are considered in this paper: Inheritance dependency, Interface implementation dependency, Method calls dependency (aggregation).

For weight extraction, the calculation method of the edge weight was as follows:

$$W_{ij} = \frac{d_{ij}}{\sum_{k \in N(j)} d_{ik}} \quad (1)$$

where W_{ij} represents the weight between node v_i and node v_j, d_{ij} stands for the number of dependencies between the two nodes, and $N(j)$ denotes the set of neighbors of the node v_j.

It is worth mentioning that in the weighted network, the weight of the edge is not related to the direction, but to the current target node. Some explanations are mentioned in reference [10].

2) Node attribute generation

Before training GraphSAGE, we must provide the attributes of the nodes in CDN. Node attribute metrics can include many types, such as traditional static code metrics, complex network metrics, and network-embedded metrics. Traditional static Code Metrics (TCM) consist of twenty manually designed metrics, such as CBO (number of classes coupled to a given class), WMC (number of methods in a given class), and LOC (lines of source code). Complex Network Metrics (CNM) are extensively employed in social networks, including seventeen metrics such as density, size, and extent. Network structure information is extracted from CDN through network embedding learning as a Network Embedding Metric (NEM). We used the node2vec [14] method to map each class node to a low-dimensional vector.

Different metrics can be combined in many ways. According to our previous research [4], the combination of TCM and NEM as node attributes is the optimal choice. Therefore, we selected this combination of metrics as the initial attributes of nodes in class dependency networks to feed into GraphSAGE for training.

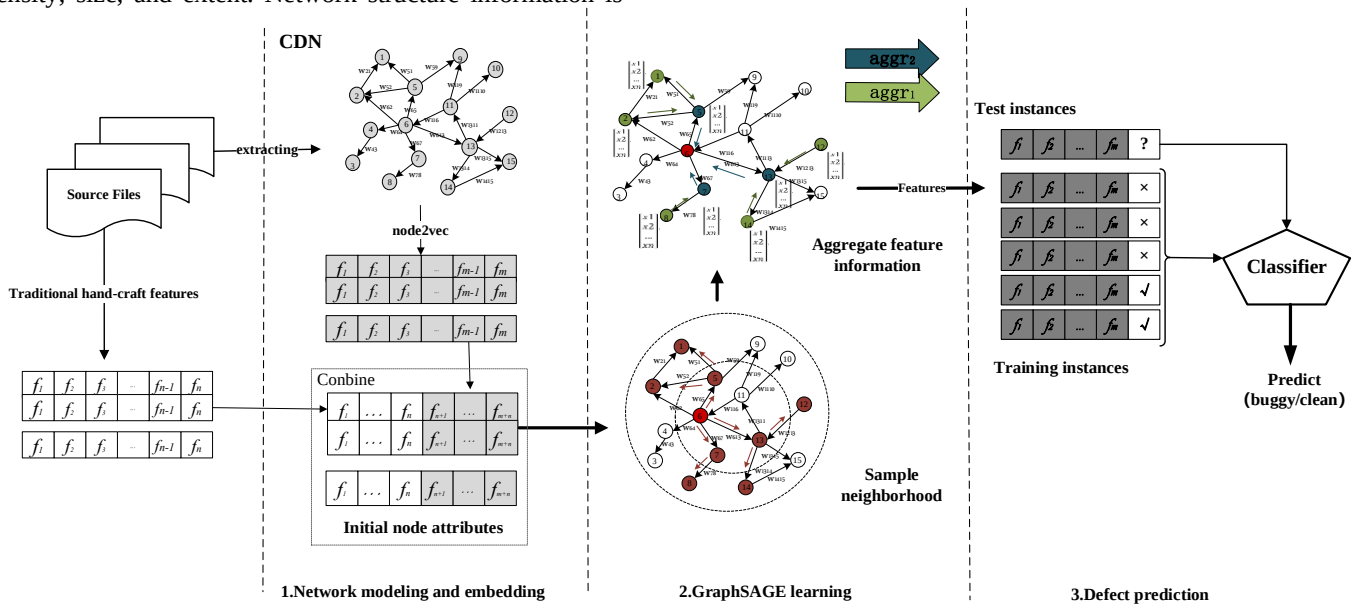

Figure 1. The framework of our approach

B. *GraphSAGE learning*

Algorithm 1 describes the entire learning process. Specifically, a CDN and its node attributes are provided as input. Each step in the outer loop of Algorithm 1 proceeds as follows, where k signifies the current search depth in the outer loop, and $\boldsymbol{h}_v^k$ indicates the node representation at this search depth. First, each node $v \in \mathcal{V}$ aggregates the representations of the nodes in its immediate neighborhood, $\boldsymbol{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\}$, and then convert to a single vector $\boldsymbol{h}_{N(v)}^{k-1}$. After aggregating adjacent feature vectors, GraphSAGE concatenates the current representation of the node $\boldsymbol{h}_v^{k-1}$ with the aggregated domain vector $\boldsymbol{h}_{N(v)}^{k-1}$ to obtain the features of the current layer node representation, the final output is the feature matrix $\boldsymbol{Z}$ of the node.

Algorithm 1: GraphSAGE algorithm for generating node features

Input: $G(V, E)$; Node initial feature vector X; Depth K; weight matrix $W^k, \forall k \in \{1, \cdots, K\}$; aggregation functions $AGGREGATE_k, \forall_k \in \{1, \ldots., k\}$; neighborhood function $N: v \rightarrow 2^v$

Output: Node Feature Matrix Z.

1 Initialize the node representation vector $h_v^0 = X_v, \forall v \in V$;
2 if $k = 1 \cdots K$:
3 do the following for each node $v \in V$:
4 $h_{N(v)}^k = \mathrm{AGGREGATE}_k(\{h_u^{k-1}, \forall u \in N(v)\})$;
5 $h_v^k = \sigma(W^k \cdot \mathrm{CONCAT}(h_{N(v)}^k, h_v^{k-1}))$;
6 $h_v^k = h_v^k / \|h_v^k\|_2$; // Normalize the h_v^k obtained by each layer
7 ends
8 $z_v = h_v^k, , \forall v \in V$;
9 Output node feature matrix Z.

GraphSAGE provides three aggregator functions: average aggregation, Long Short-Term Memory (LSTM) aggregation, and max pooling aggregation. we selected the default aggregator function (i.e., max pooling aggregation).

IV. EXPERIMENT

A. *Datasets*

In our experiments, we utilized eight classical defect projects published by the PROMISE[1] library. Table I shows the details of the eight software projects, in which #Nodes represents the number of class files in CDN, #Edges indicates the number of dependencies between class files, #Defective signifies the number of buggy files in the project, and %Defective denotes the corresponding buggy rate.

B. Experimental setup

Because the defect dataset was imbalanced, it needs to be sampled before training. Here we consider 4 commonly used sampling strategies: SMOTE [15]、BorderlineSMOTE [16]、SMOTEENN [17] and SMOTETomek [18].

Five-fold cross-validation was used (the training set accounted for 80% and the test set 20%), and we repeated the experiment 25 times. The final results were averaged to reduce the bias introduced by randomly dividing the data. The experimental parameters of this study are shown in Table II.

TABLE I. DATASET

project	version	#Nodes	#Edges	#Defective	%Defective
Ant	1.7.0	703	3012	128	22.76%
Camel	1.6.0	906	3644	145	20.09%
Ivy	2.0	343	1710	31	11.37%
jEdit	4.1	292	1044	58	25.00%
Velocity	1.6.1	210	1035	60	35.71%
Poi	3.0	421	1304	273	64.85%
Lucene	2.4.0	324	1353	194	59.88%
Xalan	2.6.0	801	3965	362	45.19%

TABLE II. EXPERIMENTAL ENVIRONMENT

Parameter	Parameter value
Feature dimension	32
epochs	2000
optimizer	Adam
initial learning rate	0.001
output dimension	32
dropout	0.1
Imbalance Handling Threshold σ	0.4

To verify the effectiveness of GSAGE2defect, we selected the following eight benchmark methods for comparison: **dw2defect** [18], **node2defect** [2], **DP-CNN** [12], **Seml** [19], **SDNE2defect** [20], **Struc2defect** [21], **GCN2defect** [3], and **GAT2defect** [22].

C. Evaluation metrics

The evaluation metric of our experiment adopts F-measure. The formula for calculating F-measure is as follows:

$$precision = \frac{TP}{TP + FP} \quad (2)$$

$$recall = \frac{TP}{TP + FN} \quad (3)$$

$$F1 = \frac{2 * precision * recall}{precision + recall} \quad (4)$$

Among them, TP (True Positive) and TN (True Negative) are the numbers of positive and negative samples predicted correctly, FP (False Positive) is the number of negative samples predicted as positive samples, and FN (False Negative) is the number of positive samples predicted as negative samples.

D. Results and Analysis

RQ1*: Does the proposed GSAGE2defect approach work well?*

For a fair comparison, the sampling method and classifier in the benchmark model were chosen consistent with our proposed method. All baseline models use SMOTETomek and Random Forest (RF) in predictions.

To further explore RQ1, we contrasted our approach with four baseline models and four classical model approaches. Figure 2 (left) exhibits that our proposed method is superior to the four baseline methods in terms of overall performance, which is evident by the higher median value of the F-measure (0.919). Among the four baseline methods, the performance of the GAT prediction model of the graph attention neural network (labeled as GAT2defect) was better than the other three baseline methods, and its median value of the F-measure was 0.843. Moreover, Figure 2 (right) displays that the proposed method is also better than the four classical methods. This is reflected by its median value of the F-measure which is the greatest (0.919).

The overall F-measure value of dw2defect was 0.793, struc2defect was 0.702, sdne2defect was 0.804, GAT2defect was 0.843, and GSAGE2defect was 0.917. The performance of GSAGE2defect was indeed better than the four baseline methods. The F-measure values of the other four classical methods of DP-CNN [12], Seml [19], node2defect [2], and GCN2defect [3] were 0.807, 0.714, 0.714, and 0.894, respectively. The performance of GSAGE2defect was clearly better than the four classical methods.

Figure 3 (left) depicts the improvement in the prediction of the GSAGE2defect model compared to that of the eight benchmark models. It can be observed from the figure that the GSAGE2defect model has provided a certain improvement in the prediction compared with other benchmark models, and the improvement range is between 2.09% and 26.69%.

In general, according to the results of Figure 3, it can be found that the GSAGE2defect model has a better improvement than the benchmark model, with a maximum improvement of 26.69% and an average improvement of 13.22%.

RQ2: *Do the dependency direction and weight of CDN have a significant impact on GSAGE2defect?*

To answer RQ2, we designed four sets of experiments to discuss the prediction performance of the GSAGE2defect model in four network scenarios and to explore the influence of the dependency direction and weight on the model during network modeling. The results are displayed in Figure 3(right) and Table III. Table III shows the F-score of the GSAGE2defect model under eight projects in four network scenarios.

On the whole, the GSAGE2defect model had the best performance under the directed and unweighted network with the highest average F-score (0.921), and five of the eight projects achieved the highest F1-score in the directed and unweighted scenario, followed by the scenario of undirected weighted and directed weighted, and the worst was undirected unweighted.

Besides, we introduced Wilcoxon signed-rank test (p-value) and Cliff's [23] influence factors, and compared and analyzed the differences in the experimental results. According to the results of the p-value and d-value in Table 4, the class dependency direction and weight do not have a very substantial impact on the prediction performance of the GSAGE2defect model.

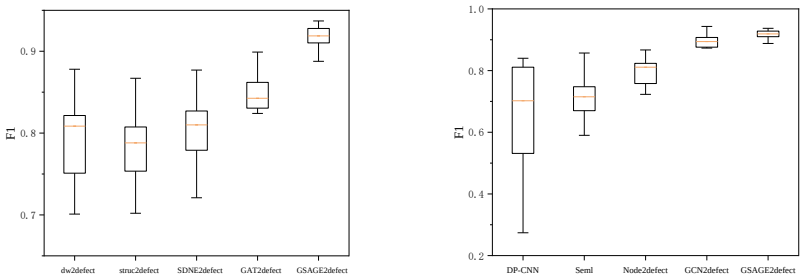

Figure 2. Comparison of F1 values with the baseline model (left) and the classic model (right)

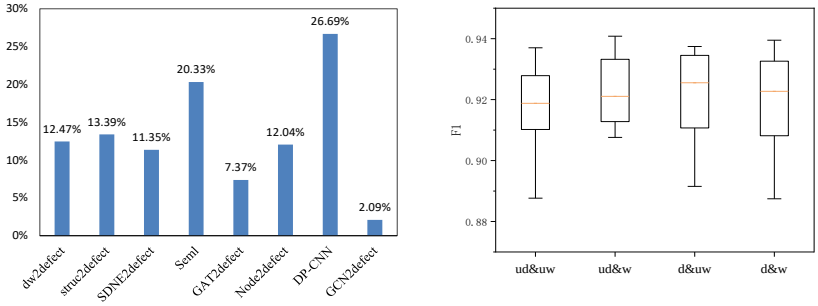

Figure 3. The improvement in the prediction effect compared to the baseline model(left) and F1 comparison of GraphSAGE under four network scenarios(right)

TABLE III. COMPARISON OF MICRO-F1 RESULTS IN FOUR NETWORK SCENARIOS

Project	F1(F-measure)			
	ud & uw	ud & w	d & uw	d & w
Ant	0.907	**0.908**	0.903	0.905
Camel	0.936	**0.941**	0.937	0.937
Ivy	0.937	0.938	0.937	**0.939**
jEdit	0.919	0.923	**0.932**	0.925
Velocity	0.913	**0.918**	**0.918**	0.911
Poi	0.920	0.928	**0.930**	0.928
Lucene	0.919	0.919	**0.921**	0.921
Xalan	0.888	0.881	**0.892**	0.887
Average	0.917	0.920	**0.921**	0.920
comparation			Sig.p < 0.05, d	
ud&uw vs ud&w			0.1913 **(-0.1875)**	
d&uw vs d&w			0.1627 (0.0625)	
ud&uw vs d&uw			0.0780 (**-0.2180**)	
ud&w vs d&w			0.8321 (0)	

V. DISCUSSION

For addressing RQ1, we reproduced the GCN2defect method on the dataset processed in this paper. The defect rate of the dataset in our study was slightly different from that in [3]. This may originate from the fact that the direction information and weight information between nodes were not extracted in [3] when processing the dataset, however, we extracted these two types of information.

For addressing RQ2, the comparison of directed unweighted vs. directed weighted and undirected weighted vs. directed weighted groups demonstrated that considering the effect of both direction and weight is not optimal compared to considering a single element. In the comparison between undirected unweighted and undirected weighted cases, the F-score of seven out of eight projects in the undirected weighted was greater than the former and the d-value was negative. This indicated that in the undirected case, the weighting effect was better. Compared with the directed and unweighted case, the F-score of six out of the eight projects of the directed and unweighted case was greater than the undirected unweighted case and the d-value was negative, suggesting that the directed effect was better in the case of the unweighted. Therefore, it is still necessary to consider the dependency direction and weight in the software network modeling process.

Moreover, we chose random forest as the classifier in RQs. To further explore the influence of the classifier on the model predictions, we tried four different classifiers in the GSAGE2defect model, including Random Forest (RF), Multilayer Perceptron (MLP), Decision Tree (DT), and Logical Regression (LR). The GAGE2defect model worked best in directed and unweighted scenarios. Thus, in this scenario, we conducted a comparative experiment of classifiers in the GSAGE2defect model. Table IV exhibits the effects of the four classifiers on GSAGE2defect, and the best values are shown in bold. In addition, Table V shows the significant differences when using different classifiers. From the results of the p-value and d-value, it can be observed that the difference between RF and the other three classifiers is significant.

TABLE IV. THE EFFECT OF THE CLASSIFIER ON GSAGE2DEFECT

Project	LR	DT	MLP	RF
Ant	0.732	0.653	0.830	**0.903**
Camel	0.805	0.757	0.872	**0.937**
Ivy	0.800	0.689	0.812	**0.937**
jEdit	0.821	0.781	0.833	**0.932**
Velocity	0.827	0.818	0.320	**0.918**
Poi	0.922	0.907	0.460	**0.930**

Lucene	0.892	0.854	0.450	**0.921**
Xalan	0.893	0.823	0.872	**0.892**
average	0.836	0.785	0.681	**0.921**

TABLE V. STATISTICAL TEST RESULTS OF GSAGE2DEFECT

GSAGE2defect	p-value	Cliff's delta
LR vs RF	**0.016**	**-0.828**
DT vs RF	**0.006**	**-0.938**
MLP vs RF	**0.019**	**-0.938**

VI. CONCLUSION

This paper proposed a new defect prediction method GSAGE2defect, by introducing an inductive graph neural network model GraphSAGE to automatically learns the dependencies between nodes in a class dependency network and using SMOTETomek sampling to solve the problem of sample imbalance. We verified the effectiveness of our method on data from eight open-source projects, and the results indicated that GSAGE2defect can outperform the baseline model by 26.69% in terms of F-measure, and also revealed that considering weight and direction in the class dependency network is helpful for software defect prediction. If only one of these two factors is considered, a certain improvement will occur in the predictions of the model. In general, the model had the best prediction in the directed and unweighted scenario. Besides, our method has an absolute advantage compared to the four baseline models, and it also has obvious advantages in comparison to the five classic methods.

This study is only a small part of the results of our comprehensive research, and there is still much research that should be completed. In the future, we will extend this study to extract richer features and to build more complete software networks. Furthermore, we will extend our research to focus on defect prediction across various projects.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (Nos. 61832014, 61902114, 61977021), and the Key R&D Programs of Hubei Province (No. 2021BAA184, 2021BAA188).

REFERENCES

[1] Narayanan, A, Chandramohan, et.al. graph2vec: Learning Distributed Representations of Graphs[J]. arXiv preprint arXiv:1707.05005, 2017.

[2] Qu Y, Liu T, Chi J and Zheng Q. node2defect: using network embedding to improve software defect prediction[C]//2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2018: 844-849.

[3] C. Zeng, C. Y. Zhou, S. K. Lv, P. He and J. Huang, "GCN2defect: Graph Convolutional Networks for SMOTETomek-based Software Defect Prediction," 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), 2021, pp. 69-79.

[4] Zhou C, He P, Zeng, C and Ma J. "Software defect prediction with semantic and structural information of codes based on Graph Neural Networks." Information and Software Technology 152 (2022): 107057.

[5] Marcheggiani D, Bastings J, Titov I. Exploiting semantics in neural machine translation with graph convolutional networks[J]. arXiv preprint arXiv:1804.08313, 2018.

[6] Ying R, He R, Chen K, et al. Graph convolutional neural networks for web-scale recommender systems[C]//Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining. 2018: 974-983.

[7] Monti F, Bronstein M, Bresson X. Geometric matrix completion with recurrent multi-graph neural networks[J]. Advances in neural information processing systems, 2017, 30.

[8] Hamilton W, Ying Z, Leskovec J. Inductive representation learning on large graphs[J]. Advances in neural information processing systems, 2017, 30.

[9] Hajibabaee P, Malekzadeh M, Heidari M, et al. An empirical study of the graphsage and word2vec algorithms for graph multiclass classification[C]//2021 IEEE 12th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON). IEEE, 2021: 0515-0522.

[10] ZHOU Chun-Ying, ZENG Cheng, HE Peng and ZHANG Yan. GKCI: An Improved GNN-based Key Class Identification Method[J].Journal of Software,2023,34(6):0-0.

[11] Wang, S., Liu, T., Nam, J., & Tan, L. (2018). Deep Semantic Feature Learning for Software Defect Prediction. IEEE Transactions on Software Engineering, 46, 1267-1293.

[12] Li J, He P, Zhu J and Lyn. M. R. Software defect prediction via convolutional neural network[C]//2017 IEEE international conference on software quality, reliability and security (QRS). IEEE, 2017: 318-328.

[13] P Phan A V, Le Nguyen M, Bui L T. Convolutional neural networks over control flow graphs for software defect prediction[C]//2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI). IEEE, 2017: 45-52.

[14] Grover A, Leskovec J. node2vec: Scalable feature learning for networks[C]//Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining. 2016: 855-864.

[15] Chawla N V, Bowyer K W, Hall L O and Kegelmeyer W P. SMOTE: synthetic minority over-sampling technique[J]. Journal of artificial intelligence research, 2002, 16: 321-357.

[16] Han H, Wang W Y, Mao B H. Borderline-SMOTE: a new over-sampling method in imbalanced data sets learning[C]//Advances in Intelligent Computing: International Conference on Intelligent Computing, ICIC 2005, Hefei, China, August 23-26, 2005, Proceedings, Part I 1. Springer Berlin Heidelberg, 2005: 878-887.

[17] Batista G E, Prati R C, Monard M C. A study of the behavior of several methods for balancing machine learning training data[J]. ACM SIGKDD explorations newsletter, 2004, 6(1): 20-29.

[18] Perozzi B, Al-Rfou R, Skiena S. Deepwalk: Online learning of social representations[C]//Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. 2014: 701-710.

[19] Liang H, Yu Y, Jiang L and Xie Z. Seml: A semantic LSTM model for software defect prediction[J]. IEEE Access, 2019, 7: 83812-83824.

[20] Wang D, Cui P, Zhu W. Structural deep network embedding[C]//Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining. 2016: 1225-1234.

[21] Ribeiro L F R, Saverese P H P, Figueiredo D R. struc2vec: Learning node representations from structural identity[C]//Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining. 2017: 385-394.

[22] Veličković P, Cucurull G, Casanova A, et al. Graph attention networks[J]. arXiv preprint arXiv:1710.10903, 2017.

[23] Cliff N. Ordinal methods for behavioral data analysis[M]. Psychology Press, 2014.