2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC) | 979-8-3503-2697-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/COMPSAC57700.2023.00114

EDP-BGCNN: Effective Defect Prediction via BERT-based Graph Convolutional Neural Network

Hao Shen†, Xiaolin Ju†*, Xiang Chen†, Guang Yang‡
†*School of Information Science and Technology*, *Nantong University*, China
‡*School of Information Science and Technology*, *Nanjing University of Aeronautics and Astronautics*, China
§ Email: shen.h@stmail.ntu.edu.cn, ju.xl@ntu.edu.cn, xchencs@ntu.edu.cn, novelyg@outlook.com

***Abstract*—Software defect prediction (SDP) is a critical task that aims to identify potential defects and allocate resources for testing to enhance software reliability. In this study, we present a novel defect prediction framework called EDP-BGCNN, which leverages the power of BERT and graph convolutional neural networks to represent code. Our approach first extracts the code's structural semantic features based on its abstract syntax tree (AST), followed by applying BERT for embedded learning to extract the code's semantic features. We then use latent Dirichlet allocation (LDA) to extract descriptive semantic features and convert them into a numeric vector. The code and descriptive semantic features are then combined and processed by GraphSMOTE to address the class imbalance problem. Finally, we obtain a more comprehensive representation using graph convolutional neural networks. We evaluated our approach on five open-source projects and compared it with three state-of-the-art deep-learning methods. Our experimental results demonstrate that EDP-BGCNN can achieve significant improvements in AUC (4.9% - 23%) and F1-measure (6.6% - 10.7%) on average.**

I. Introduction

As software systems have grown increasingly complex, ensuring their quality and reliability has become a critical concern. To address this issue, many companies, such as Google, have implemented code reviews and team unit testing to identify potential code defects [1]. However, manual code reviews and unit testing can be costly and time-consuming. To address these challenges, software defect prediction (SDP) technology has emerged as a powerful tool that automatically identifies potential defects, helping developers allocate limited resources while ensuring software reliability at a lower cost [2], [3].

SDP prefers to construct models by considering data like code complexity and change history [4], and then predict defects in new code modules. Many studies on SDP have concentrated on prediction by machine learning. By designing metrics based on historical data and utilizing classifiers like the Naive Bayesian (NB), support vector machine (SVM), decision tree (DT), and random forest (RF). They manually design new discriminative features, such as Halstead feature [5], McCabe feature [6], and CK feature [7], to better predict defects in their studies. Furthermore, Many SDP studies have supported the advantage of abstract syntax trees (AST) [8]–[10]. Deep learning techniques are utilized to build powerful neural networks and construct prediction models using AST or program dependency graphs (PDG) [11].

Unfortunately, existing code representation methods may overlook crucial code structural information. Previous studies have concentrated on manual features that emphasize the statistical properties of programs, presuming that defect codes differ from non-defect codes in software metrics [12]. In fact, the manual features of the defect and non-defect codes might be similar, which makes it challenging for the classifier to predict. Therefore, many studies concentrate on manual features and code semantic features. We need features that can distinguish semantic differences to construct more appropriate defect prediction models. Similar to natural language, programming languages also have syntactic rules, which offers the possibility of extracting rich structural semantic feature. AST is a special source code representation containing information about how the code is organized and interacts [13]. Many studies [14], [15] extracted code flattened features. As shown in Fig. 1 (a), this is a code snippet that represents the bubble sorting function. Fig. 1 (b) shows that current deep learning methods construct ASTs to learn flattened token sequences. However, AST also contains structural features as shown in Fig. 1 (c). If we follow the methods studied in the past to deal with code, we will ignore the code's structural features.

Moreover, the code description can be seen as a summary. The description may contain information such as program language, modification time, functional description, etc. Numerous studies [16]–[18] have demonstrated the effectiveness of using models to extract valuable insights from unstructured software engineering (SE) data. As a result, there is growing interest in this area, with an increasing focus on theme modeling to measure concerns in source code and identify the likelihood of defects at different levels in the code. Previous research has explored the possibility of enhancing the semantic features of programs represented by AST by combining them with descriptive features. This approach intuitively leads to a richer feature representation.

In this paper, we introduce a novel framework called Defect Prediction by Graph Convolutional Neural Network (DGNN), which combines program semantic and structural features with descriptive features to improve defect prediction. The framework begins by parsing the source code into Abstract Syntax Trees (ASTs) and generating a sequence of AST nodes and node-relationship pairs. We then combine the features obtained

* Xiaolin Ju is the corresponding author.

979-8-3503-2697-0/23/$31.00 ©2023 IEEE
DOI 10.1109/COMPSAC57700.2023.00114

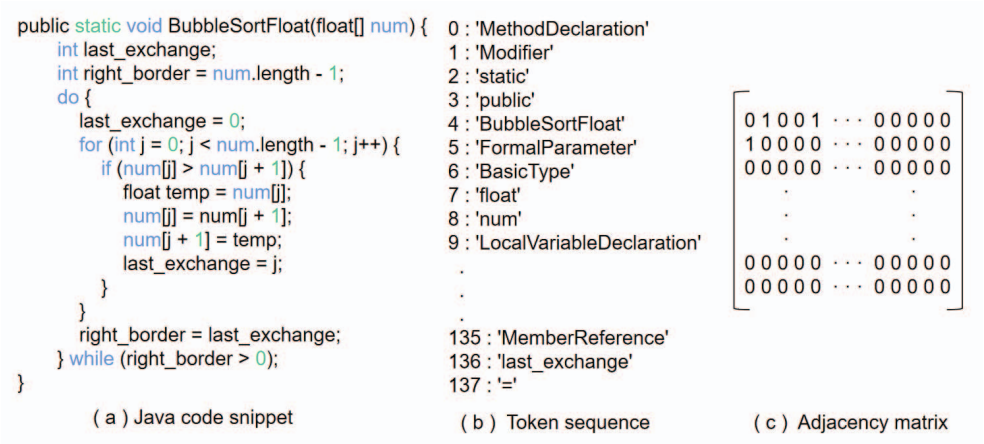

Fig. 1. Structural Feature Sample

from the description and apply them as node attributes in the AST. The node sequences and description are converted to numeric vectors and fed into the graph convolutional neural network to obtain the final representation. The features are then sent to the Multi-Layer Perception (MLP) classifier to determine whether they contain defects. Finally, we evaluate and compare our method with three state-of-the-art baselines on five open-source projects. The experiment results show EDP-BGCNN can improve the performance 4.9%−23%, in terms of AUC and 6.6%−10.7% in terms of F1-measure on average.

The main contributions of this study are as follows:

- We propose a defect detection framework based on graph convolutional neural networks for automatically generating distinctive features from the AST of the program to mine both semantic and structural data.
- We apply the pre-trained model-BERT to encode nodes extracted from ASTs and description, which helps graph convolutional neural networks to more precisely learn the semantics of programs.
- We also compare the performance of EDP-BGCNN with popular deep learning methods on Java projects and outperform them in AUC and F1-measure.

II. Background

In this section, we provide background on the main techniques within our model: deep features extraction by Abstract Syntax Tree, graph convolutional neural network, and word embedding.

A. Deep Features Extraction from Abstract Syntax Tree

Abstract syntax trees(ASTs) have been utilized in numerous research studies for source code search [19], program repair [20], and source code representation learning [13]. An AST is designed to describe the abstract syntactic structure of source code [21], where each node in the tree represents a structure in the source code and is a high level of abstraction of the source code. Identifiers and literals in the code are represented by leaf nodes in ASTs, while non-leaf nodes can express some grammatical structure.

As shown in Fig. 1 (a), it utilizes bubble sort to achieve a column of data from smallest to largest. As shown in Fig. 1 (b) and (c) is the token sequences and relationship matrix of the AST corresponding to the code in (a). Where MethodDeclaration is the declaration node and BubbleSortFloat is the method name. AST includes some nodes that can and cannot be seen in the source code compared with the initial source code. For example, float corresponds to float in the source code, and static corresponds to static in the code. However, FormalParameter and BasicType do not find corresponding statements in the code. Assuming that these nodes do not exist in the AST, the information we obtain about the coed structural features may be affected. Therefore, we cannot discard these types of nodes and need to supplement the structural information of the graph with the properties of their children to avoid the incomplete structure of the graph. All nodes in the AST have their corresponding information in the source code. It can be determined that their corresponding code parts are problematic when some nodes are judged to be in the class of problematic nodes.

B. Semantic Learning by Graph Convolutional Neural Network

Some studies [11] have applied graph representation to software defect prediction with the application of deep learning in SDP, driven by a number of success factors. Since non-Euclidean spaces are used to generate data in many real-world application scenarios, traditional deep learning methods still perform poorly when processing this type of data. Kipf et al. [22] first proposed the graph convolutional neural network (GCN). GCN combines the features of convolutional neural networks and graph neural networks. Additionally, it extends the convolutional non-Euclidean space of neural networks that are only applicable to Euclidean space. GCN applies the Fourier transform principle to perform convolutional operations on graphs. It contains spectral-based and spatial-based methods. The central concept of GCN is the aggregation of node information using edge information to generate new node representations. Several studies have shown which demon-

strates effectiveness in the fields of image classification [23], text classification [24], and unsupervised learning [25].

Hamilton et al. [26] proposed a framework for inductive representation learning on large graphs: GraphSAGE. It learns node representations by aggregating information over neighborhoods. To make inductive learning adaptive, GraphSAGE samples a fixed-size neighborhood for each node. Additionally, three aggregation functions are provided: MEAN Aggregator, Pooling Aggregator, and LSTM. MEAN Aggregator performs MEAN operation on the current node and its neighbor nodes, which is linearly similar to the local spectral convolution of the node and its neighbor vectors. And it quotes the average value of related nodes as the aggregation result. The pooling Aggregator puts all neighbors' vectors into a fully connected network. Then attaches a maximum pooling layer, and takes the maximum value of the relevant nodes as the aggregation result. LSTM performs a random permutation of all neighbor nodes before aggregating. Then sends the relevant nodes to the model and the output is taken as the aggregation result. Considering the time overhead of the model, many studies apply MEAN Aggregator as the aggregation function (as in Equation (1)) [27].

$$H_v^k \leftarrow \sigma\Big(W \cdot MEAN\Big(\big\{H_v^{k-1}\big\} \bigcup \big\{H_v^{k-1}, \forall u \in N(v)\big\}\Big)\Big) \quad (1)$$

where σ denotes the nonlinear transformation, W is the weight matrix for each layer of learning, H_v^k denotes the node embedding of node v after the k-th aggregation, and h_u^{k-1}, $\forall u \in N(v)$ denotes the $(k-1)$-th layer vector of neighboring embeddings. Splicing the central node's neighboring node mean vectors with the central node's vector to produce the k-th layer's nodes embedding.

GraphSAGE and GCN aggregate features on the neighboring nodes. The main difference is that GCN learns the entire graph, which tests the computational power of the GPU for processing large graphs. A shallow GCN network cannot propagate label information over a large area, and a deep GCN network can lead to excessive smoothing problems. GraphSAGE aims at learning an Aggregator rather than learning a representation for each node, which is inductive learning. New nodes will be aggregated to obtain the latest representation to avoid overfitting when added to the graph. GraphSAGE can be trained in batches to improve the convergence speed. GraphSAGE applies a special sampling method to solve the memory problem and improve flexibility and generalization ability.

C. Feature Vectorization by Word Embedding

Word embedding aims to represent each word in a dataset with a fixed-length numeric vector [28] and is a widely used technique in NLP [29], [30]. The vector representation is learned before or during the training of a deep learning model. After training and learning, the vector can be applied to measure the semantic distance between two words by various distance calculation methods. This distributed representation can improve the performance of the model on tasks such as code classification [31], sentiment classification [32], and speech recognition [33].

Each node in AST is similar to a word in NLP. Devlin et al. proposed the pre-training model BERT [34], which is a Transformer-based encoder [34]. A deep bidirectional representation from unlabeled text is pre-trained by computing conditionals that are common in the left and right contexts. This preserves the structural information of the word context in the sentence. In the code representation, the node names in the AST are recorded as words in NLP. We applied BERT to transform each node into a vector, which can be fed into the subsequent model directly.

III. Our Method

The overall framework of our method is shown in Fig. 2. The method contains three stages: data processing, model construction, and model evaluation. In the first stage, we parse the source code into ASTs. Secondly, we extract code descriptive features from the description. Then convert ASTs and code descriptive features into numeric vectors by BERT. Finally, we fuse the code semantics and descriptive features. In the second stage, we feed the fused features into GraphSMOTE to handle the class imbalance problem. Then we construct EDP-BGCNN and apply MLP classifier to predict. In the last stage, we evaluate the performance of our model. In the following, we detail the implementation process of these three phases.

A. Data Processing phase

In this stage, we process the data. Firstly, we extract the code descriptive features. Second, we parse the codes into ASTs to obtain the code semantic features. Then, we utilize BERT to transform the code descriptive features and semantic features into vectors. Finally, we fuse the descriptive and semantic features. We describe the details of each part in the following.

We consider the description for each code module as a comprehensive document, including semantic features such as code syntax, method names, and information about fixes and other meaningful contextual changes. We extract descriptive features from them. We utilize the Latent Dirichlet Assignment (LDA) [35] model to generate a set of themes. LDA model is a Bayes-based learning model, which is an extension of Latent Dirichlet Analysis and Probabilistic Latent Dirichlet Analysis to identify and extract themes. It helps to reduce model overfitting [36] compared to other theme models such as probabilistic LSI. Chen et al [37] utilize LDA to account for software defects and argue that defect-prone themes tend to persist over time. We utilize the scikit-learn machine learning library to construct the LDA model. We set the number of theme words to 10, which is the average length of code comments in many studies [38]–[40] considering the size of the description text. Words with the highest probability will be utilized to summarize the descriptive features.

We need to transform the code into vectors to learn the code's semantic features. Peng et al. [41] showed that parsing source code into node representations in AST achieved better performance in program classification tasks. These studies

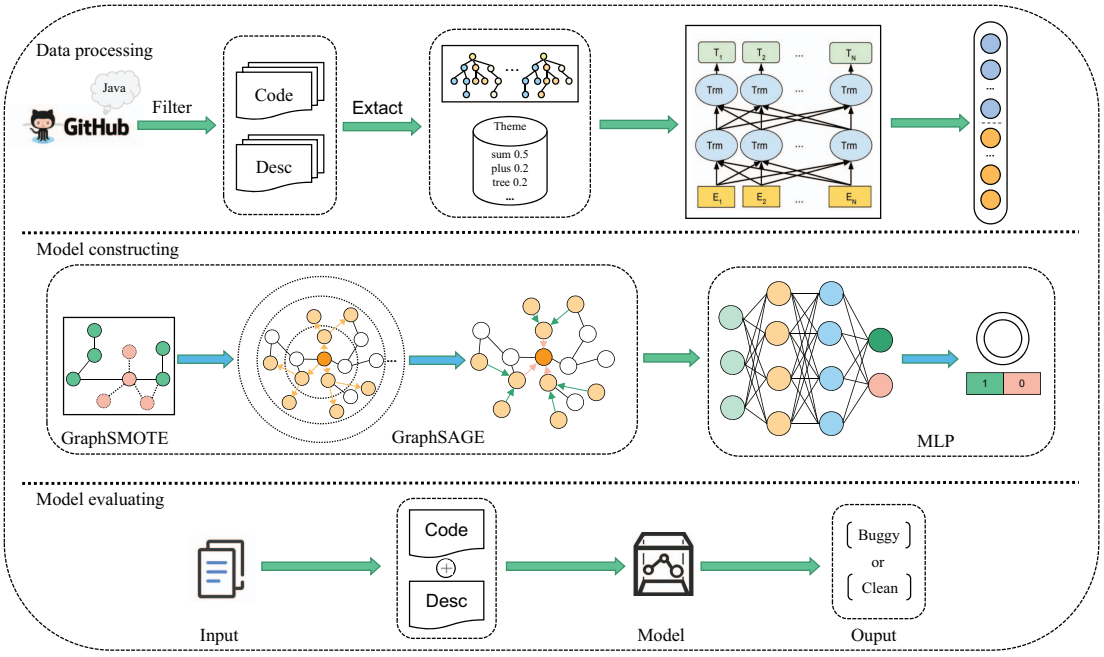

Fig. 2. Overview of EDP-BGCNN

provide the possibility of adopting a strategy that extracts the AST node-level representation from the Java source code in our study. Javalang[1] is a third-party open-source library in the Python language that provides a Java-oriented lexical parser and parser. We utilize Javalang for source code tokenization and AST generation. The node tokens in the AST and the source code may differ, for example, MemberReference, MethodDeclaration, and BasicType. However, we can find the corresponding AST nodes according to the syntax rules of AST. We consider the AST as a special undirected graph after getting the AST nodes. We obtain the adjacency matrix about the structure of the code, which is symmetric on the diagonal by traversing this graph.

We traverse the AST to get all tokens corresponding to the source code. We utilize a pre-trained model-BERT [34] to convert token and descriptive features into numeric vectors. BERT is a multilayer bidirectional transformer encoder. It emphasizes that instead of using a traditional one-way language model or shallow splicing of two one-way language models for pre-training. It utilizes a new Masked Language Model (MLM) to generate a deep bidirectional language representation. One of its distinguishing features is its unified architecture across different tasks. Moreover, each descriptive feature is considered as a sentence representing the whole code project.

We utilize AST to capture the code semantic features. Then we utilize LDA to extract the code descriptive features. To integrate the code semantic and descriptive features, we also need a fusion mechanism. We enhance the dimensional features h_{code} by vector stitching, which can be described as:

$$h_{code} = concat(h_{AST}, h_{LDA}) \tag{2}$$

where h_{AST} represents the code semantic features learned from the AST sequence, and h_{LDA} represents the code descriptive features learned from the description.

B. Model Construction phase

In this stage, we construct the defect prediction model. Firstly, we perform class imbalance processing on the fused features. Then, we input the processed code semantics and descriptive features into GraphSAGE. We obtain the code semantic representation by sampling and aggregation. Finally, we utilize MLP to classify the marked code representation to distinguish between defective and non-defective code. We describe the details of each part in the following.

In most cases, the number of defective code statements is much less than the number of non-defective code statements. Unbalanced data will bias the prediction results in favor of the non-defective statements. Class imbalance refers to the situation where the number of training samples differs significantly across categories [42]. In real life, perfectly balanced datasets in which the training proportions of different classes are the same rarely exist. Suppose the ratio of one category in the data is too high as 10:1, then the accuracy will be high because the model only needs to predict the results for this category. This result is not informative for the model.

There are three solution strategies to solve the class imbalance problem currently. Oversampling, undersampling, and their combination. Oversampling means sampling a small

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

number of classes to increase the number of samples. Undersampling means sampling less and reducing the number of samples for classes with a larger number of samples. The combination of oversampling and undersampling gives comprehensive results. Undersampling eliminates some code features and leads to overfitting because of missing overall samples. Mohammed et al. [43] compared the performance of oversampling and undersampling on machine learning and concluded that oversampling performs better. We utilize oversampling to perform random replication of the data to ensure the integrity of the data.

SMOTE [44] is a classical oversampling algorithm. SMOTE calculates the distance from each sample in the minority class to all samples in the minority class sample set in terms of Euclidean distance to get its k-nearest neighbors. Then generate a number between 0 and 1 randomly to synthesize a new sample. However, SOMTE can only synthesize new samples and lacks the edge relations in graph data. Zhao et al [45] proposed GraphSMOTE which is a migration of SMOTE. GraphSMOTE generates some synthetic nodes mainly by interpolating the expression embedding space obtained from the GNN-based feature extractor. Then predicts the links of synthetic nodes utilizing an edge generator to form an extended graph with balanced classes. Therefore we utilize GraphSMOTE to process the data into a balanced graph.

We extract the code feature by data processing, which is to utilize the AST node features and nod e relationships of the source code as a defect representation. Moreover, we combine the code semantic features with the descriptive features as the input for our graph representation learning. The GNN framework and its variants transform the feature representations of nodes by utilizing aggregated weight matrices, etc., and utilize linear unit activation functions (Relu) of rectification to implement nonlinear transformations. In our method, we utilize GraphSAGE [46] model. We set the number of neurons in each hidden layer to be the same, which can make the model relatively simplified. Algorithm 1, describes the process of embedding generation.

Algorithm 1 GraphSAGE embedding generation

Input: Graph:G(V, E); features:x_v, $\forall v \in V$; depth:K; weight matrices:W_k, $\forall k \in \{1, ..., K\}$; non-linearity:$\sigma$; aggregator functions:$MEAN_k$, $\forall k \in \{1, ..., K\}$;neighborhood function $N : v \rightarrow 2^V$

Output: Vector representations:Z_v

$h_v^0 \leftarrow x_v, \forall v \in V$
for $k = 1...K$ **do**
  for $v \in V$ **do**
    $h_{N(v)}^k \leftarrow MEAN_k(\{h_u^{k-1}, \forall u \in N(v)\})$;
    $h_v^k \leftarrow \sigma(W^k \bullet CONCAT(h_v^{k-1}, h_{N(v)}^k))$
  end for
  $h_v^k \leftarrow h_v^k / \|h_v^k\|_2, \forall v \in V$
end for
return $Z_v \leftarrow h_v^K, \forall v \in V$

Specifically, GraphSAGE utilizes a new information transmission framework where nodes' structural and semantic information is transmitted from point to surface. A node can aggregate the information of its neighbors and update the information of the current node through an update function through an aggregation function, which is an iterative information transfer process. A node can aggregate the information of its higher-order neighbors as the number of iterations increases. We input the set of code nodes after data processing into the model and sample their first-order neighbors. Then we sample their first-order neighbors, which are second-order neighbors, and repeat until the sampling of K-order neighbors is complete. We fix a number of neighbors to sample at each level. The sampling is repeated when the number of neighbors of a node is less than this number. Then we utilize MEAN aggregation for the current node and its neighbors. Each dimension in the embedding of the neighboring nodes is averaged, then stitched with the embedding of its node. After that, we utilize a nonlinear transformation to generate the K-layer representation vector of the target vertex.

After obtaining the learned features, we send them to the MLP. The software defect prediction problem can be viewed as a binary classification problem, so we utilize the softmax function to output the probability of each node (as in Equation (3)):

$$\text{softmax} = \frac{e^{y_i'}}{\sum_{j=1}^{n} e^{y_j'}} \tag{3}$$

where y_i' is the output of the node i, and n is the number of all nodes. The softmax function normalizes the output to obtain a probability distribution for the basis of defective node classification.

To avoid overfitting, in the forward propagation embedding generation, we utilize dropout [9] regularization to randomly remove a portion of neural units to enhance the model's generalization ability.

In the third stage, we evaluate the performance of the model. We take the code and description as input and learn features from the model to identify the defective parts. From the level of the metrics, we can determine whether the model is reliable or not.

IV. Empirical Setup

In our study, we evaluate the effectiveness of our model (EDP-BGCNN) in defect prediction tasks by answering the following four questions(RQs):

RQ1: What is the performance of the proposed method?

RQ2: Is the project information of the project helpful for defect prediction?

RQ3: Does sampling have an effect on the results?

RQ4: Does the size of the dataset have an impact on the performance of the model?

In the rest of this section, we first introduce the details of our empirical research subject. Then, we describe the experimental evaluation indicators. Moreover, we present the three state-of-the-art baselines. Finally, we explain the details of our experiment setting.

A. Experimental subject

We evaluate the effectiveness of the EDP-BGCNN framework using five processed open-source projects: pulsar, processing, cas, keycloak, and vavr. The processed dataset is available online[2]. They are from the GHPR dataset, which is automatically generated by the tool BugMiner and contains their corresponding modified code. Since they were collected from many projects in Github, the sample is diverse. The largest defective instance contains 2704 nodes and the smallest defective instance contains only 5 nodes. We manually removed and corrected some noisy data without defects and incorrect project information. The version number before and after modification corresponded, and structurally weak code was removed. TABLE I provides detailed information about the datasets, including their code line numbers and the average number of nodes and edges in each file.

TABLE I
DETAILS OF DATASET

Project	Loc	#Node	#Edge
pulsar	13910	870	234
processing	12527	829	219
cas	9384	407	109
keycloak	2298	495	140
vavr	8923	759	176

B. Evaluation Indicators

In our study, we choose the metrics widely used in the studies [47], [48]: AUC and F1-measure to evaluate the predictive performance. According to the confusion matrix, we can get a definition of them.

Precision represents the percentage of real defective code among the code predicted to be defective.

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

Recall represents the percentage of true defective code that is predicted to be defective.

$$Recall = \frac{TP}{TP + FN} \tag{5}$$

F1-measure is a derived validity measure that measures the summed average of precision and recall.

$$F1\text{-}measure = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{6}$$

F1-measure [49] is applied to measure the stability of EDP-BGCNN, and AUC is applied to evaluate its discriminative power. F1-measure is a combined Precision and Recall metric. The higher the F1-measure, the better the prediction performance AUC is the area under the ROC (i.e. receiver operating characteristic) curve to evaluate the predictive capability of the model. Additionally, AUC performs well on datasets with class imbalance.

[2]https://github.com/Abel-pipihao/gnndf

C. Baselines

To evaluate the performance of EDP-BGCNN, we apply three state-of-the-art methods as baselines. The first one is the Seml method [15], which built long short-term memory (LSTM) networks that automatically learn semantic features about the program and perform defect prediction. The second is a CNN-based method [14] which contained three convolutional layers and four dense layers with dropouts to generate deep features to help better generalization. The third method [50] constructed a class-dependent network to learn semantic features and apply GCN to obtain a deeper class representation.

D. Implement details

We extract the code's descriptive features and preprocess the text by word embedding. During the training process, the structural code semantic features, the code descriptive features, and their related labels (i.e., defective and non-defective) are sent to the graph neural network. The model is iteratively trained to optimize the training loss. For the testing process, the new code structural representation vector, descriptive information, and related labels are sent into the model. The model gives prediction results for the relationships between them. Through these results, various performance metrics related to the model are calculated to evaluate the performance of our model.

For each experiment, we divided the dataset into training and testing sets in the ratio of 7:3, and each experiment was repeated 25 times. Moreover, we will utilize the average result as the final result to reduce the bias caused by random data. We set the dropout to 0.1, the learning rate to 0.001, and the epoch to 100. We utilize the Adam optimization algorithm to train the model. We run our proposed method on machines running Windows 10, 64-bit, Intel Core i5-9300H CPU @ 2.40GHz, and NVIDIA GeForce GTX 3090.

V. RESULT ANALYSIS

A. RQ1: What is the performance of the proposed method?

RQ1 aims to evaluate the performance of EDP-BGCNN in an automated way. We consider the performance of two deep neural network-based methods and one graph neural network-based method for comparison (including Pan's CNN [14], Seml [15] and GCN2defect [50]).

We put four methods on our five public open-source projects for experiments. We added descriptive semantic features to each method to ensure the fairness of the experiments. Fig. 3 shows the experimental results, i.e., the AUC and F1-measure of four competitive methods. From the perspective of AUC, EDP-BGCNN improves on average 5.2% over Pan's CNN, 23% over Seml, and 4.9% over GCN2defect. This indicates that EDP-BGCNN improves the differentiation ability of software defect prediction. EDP-BGCNN improves 9.7%, 6.6%, and 10.7% in F1-measure compared with Pan's CNN, Seml and GCN2defect. The result shows that EDP-BGCNN is more stable in terms of software defect prediction. This indicates that the special sampling and aggregation methods

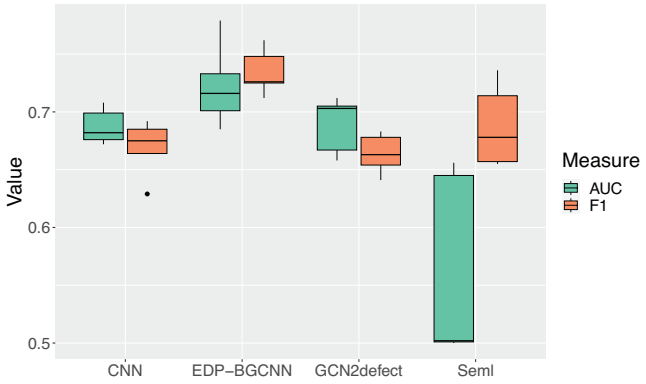

Fig. 3. Comparison with other methods

have a positive impact on a further generation of key features, which leads to better defect prediction performance. Semantic learning that considers structured features also works better than methods that consider only flattened features.

Summary for RQ1: EDP-BGCNN is capable of learning both code semantic features and descriptive semantic features. Notably, EDP-BGCNN outperforms other deep learning methods in terms of AUC and F1-measure.

B. RQ2: Is the descriptive semantic features helpful for defect prediction?

RQ2 aims to investigate whether utilizing code descriptive features helps to improve performance. Therefore, we design a method without adding code description features for comparison. In this method, the experiments are divided into two parts: pure code and code with code descriptive features. The comparison experiments are conducted on five datasets separately.

TABLE II
CONTRIBUTIONS OF DESCRIPTIVE FEATURES

Subject	Model	AUC	F1-measure
pulsar	with desc	0.779	0.762
	without desc	0.650	0.667
processing	with desc	0.733	0.748
	without desc	0.678	0.687
cas	with desc	0.716	0.726
	without desc	0.668	0.669
keycloak	with desc	0.701	0.725
	without desc	0.626	0.669
vavr	with desc	0.685	0.712
	without desc	0.625	0.666

With the experimental results obtained from this widely used ablation experiment [51], [52], we can determine whether adding descriptive semantic features can help us obtain better software defect prediction results. TABLE II shows the experimental results, i.e., the AUC and F1-measure on the five datasets. From the view of AUC, EDP-BGCNN improves 12.9% on average than without descriptive semantic features. As for the comparison without descriptive semantic features on F1-measure, our proposed EDP-BGCNN improves 11.3% on average. This indicates that code descriptive semantic features can improve the comprehension of code.

Finally, to verify whether the code descriptive semantic features statistically outperform other models. We apply paired Wilcoxon tests to assess. We obtained a p-value of 0.009 by the Wilcoxon rank test for both AUC and F1-measure between SAGE with and without descriptive semantic features. The results show that the method with descriptive semantic features is significantly improved compared to the method without descriptive semantic features.

Summary for RQ2: After incorporating descriptive semantic features, EDP-BGCNN achieved superior results in both AUC and F1-measure. This highlights the valuable contribution of descriptive semantic information toward enhancing code comprehension.

C. RQ3: Does sampling has an effect on the results?

RQ3 aims to investigate whether utilizing GraphSMOTE has an impact on the performance of the model. Therefore we designed a method without adding GraphSMOTE for comparison. In this method, the experiments are divided into two parts: sampled and unsampled. The comparison experiments are conducted on five open-source datasets.

TABLE III
CONTRIBUTIONS OF SAMPLING

Subject	Model	AUC	F1-measure
pulsar	with sampling	0.779	0.762
	without sampling	0.887	0.885
processing	with sampling	0.733	0.748
	without sampling	0.884	0.876
cas	with sampling	0.716	0.726
	without sampling	0.853	0.863
keycloak	with sampling	0.701	0.725
	without sampling	0.876	0.862
vavr	with sampling	0.685	0.712
	without sampling	0.757	0.749

As shown in TABLE III, i.e., the AUC and F1-measure on the five datasets. From the view of AUC, EDP-BGCNN reduces on average 15.1% compared to without GraphSMOTE. As for the F1-measure view, our proposed EDP-BGCNNhas an average reduction of 13.3% compared to without GraphSMOTE. This may be due to the fact that there are many more codes without defects than those with defects, producing a large imbalance in the number. This could potentially cause distress to the learning process and cause a decrease in the value of the prediction results. Therefore, it is necessary to deal with this class imbalance problem by GraphSMOTE.

Summary for RQ3: By utilizing GraphSMOTE to address the class imbalance, EDP-BGCNN demonstrates improved prediction performance. Hence, GraphSMOTE is an essential component of EDP-BGCNN.

D. RQ4: Does the size of the dataset have an impact on the performance of the model?

RQ4 aims to discuss the performance of EDP-BGCNN on different size datasets. As shown in TABLE I, we find that pulsar is the largest. These five datasets are in ascending order according to the data size in figure 4. Therefore, we have experimented with these five datasets separately with the same parameters.

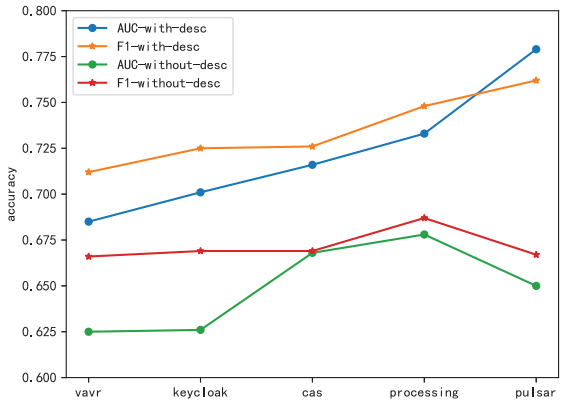

Fig. 4. Effect of data size

As shown in Fig. 4, we find that the AUC and F1-measure of EDP-BGCNN with and without descriptive semantic features on different datasets. The AUC of EDP-BGCNN on both with and without descriptive semantic features becomes better as the size of the dataset increases. This may be due to the fact that GraphSAGE is direct-push learning, and its layer-by-layer sampling and aggregating method is beneficial to large graphs. In terms of F1-measure, EDP-BGCNN increases with the size of the dataset in the project both with and without describing semantic information. We also find that the performance rise gradually becomes smoother as the size of the dataset increases. This suggests that EDP-BGCNN's performance will be optimal at a certain amount of data and will not keep rising.

Summary for RQ4: The performance of EDP-BGCNN is influenced by the size of the dataset, with gradual improvements observed as the dataset grows. However, there is a limit to this improvement, and the best performance is achieved at a certain dataset size.

VI. Threats to Validity

A. Threats to external validity

To evaluate the performance of the model. We compared EDP-BGCNN with Seml, Pan's CNN, and GCN2defect. Both Seml and Pan's CNN are the most advanced defect prediction techniques in deep learning. GCN2defect is the most advanced defect prediction technique in graph neural networks. Since the original implementation was not published, we reimplemented it as described in the text. We set the same parameters to reflect the actual defect prediction performance of the baseline to be as consistent as possible with the original method. Since it is possible to achieve the same results as theirs on the same dataset, we believe in the re-implemented experiments.

The subjects of our experiments are GHPR datasets collected from GitHub. It belongs to many different domains and is representative and general. In the future, we will consider other programming languages (such as C++ and Python) to evaluate our method.

B. Threats to internal validity

To be fair, the parameter combinations for our experiments are set to default values or to be consistent with other baseline methods. However, we want to employ evolutionary algorithms to further optimize our parameter values in the future to obtain the best performance of our proposed method.

Moreover, we only utilize PyTorch to build our model. It has higher programmability compared to TensorFlow. We may compare the feasibility of this method under different frameworks in the future.

C. Threats to construct validity

The independent variables in our study can be divided into code semantic features and code descriptive features. We utilize the pre-training model BERT for word embedding of tokens and descriptive information of the code. Other word embedding techniques may also have an impact on the results, and we will consider utilizing different word embedding techniques in the future.

We only focused on graph features composed of the AST. However, graph features such as CFG and PDG have also been shown to be effective. We only constructed GraphSAGE, and different graph neural networks (e.g. Graph Attention Network) may also have an impact on the results. In the future, we will consider different graph features and graph neural network models.

VII. Related Work

A. AST-based defect prediction

SDP assumes that software defects have a relationship with software features. Many studies [53]–[56] have demonstrated that their relationship can build a prediction model. AST has been widely applied in SDP, which reported high accuracy and recall, indicating that this method is effective [57]. Wang et al. [58] applied AST to extract semantic features. They apply a deep belief network(DBN) to automatically learn semantic features from the tag. The research showed that they can improve the prediction of software defects within and across projects. Li et al. [59] applied AST to extract marker vectors and then coded them into digital vectors through mapping

and word embedding. They applied a convolutional neural network to learn code semantic features and structural features automatically. The research showed that combining the learned features with the standard manual features can improve the performance of defect prediction. Therefore, applying AST to express the code's structural semantic features can improve software defect prediction.

B. Graph-based defect prediction

Most data in the real world is stored as graphs, for example, traffic networks, urban networks, and social networks. Many studies tend to build graph neural networks to process graph-related data which can learn structural and semantic features. AST is employed to process codes graphically since it can be thought of as a particular type of graph. Many studies have shown that semantic learning is improved with the addition of structural features [60]. Shi et al. [61] constructed a complex network model based on the dependencies between code files. Then applied the network embedding method to the generated model as structural features. The improvement of the F1-measure shows that structural features are effective. Tang et al. [62] constructed the graph by connecting the parent and child nodes. Then combining the features learned by the GCN with the manual features. It was shown that the prediction performance of the model was improved. Zhou et al. [63] constructed source code-related class dependency networks, captured the code semantic features by CNN, and learn the structural features of the network by GCN. After incorporating manual features, we combined the learned semantic and structural features using various weights. Our results demonstrate that integrating these features leads to improved code representation, allowing graph neural networks to effectively learn the code's structural features.

In contrast to the mentioned methods, our approach, EDP-BGCNN, extracts both code semantic and descriptive features. This comprehensive approach preserves all information from software modules and enables the development of more accurate defect prediction models.

VIII. Conclusion

In this paper, we propose a new defect prediction framework EDP-BGCNN to learn the code semantic and structural features, which considers the description information as natural language descriptions. Specifically, we extract the descriptive features from the description and construct ASTs to obtain the code semantic and structural features. Secondly, we obtain the vectorized representation of the sequence by BERT. Then, we utilize GraphSMOTE to handle the sample imbalance problem and build a graph convolutional neural network model, and finally perform defect prediction. The evaluation results on the open-source dataset show that EDP-BGCNN improves 9.7%, 6.6%, and 10.7% in F1-measure compared with Pan's CNN, Seml and GCN2defect, and 5.2%, 23% and 4.9% over Pan's CNN, Seml and GCN2defect in AUC.

Moving forward, our plan is to expand our dataset by collecting more open-source projects in different programming languages and building new datasets for deep feature-based defect prediction. Additionally, we aim to incorporate other code semantic features, such as Control Flow Graphs (CFG), to enhance the feature set of EDP-BGCNN. Furthermore, we intend to optimize our parameter values using the evolutionary algorithm. These efforts will help us to continue improving the accuracy and effectiveness of our framework.

Acknowledgements

This work is supported in part by the Jiangsu Province Modern Educational Technology Research Project under Grant No. 2022-R-98984 and the Nantong Application Research Plan under Grant No. JCZ21087.

References

[1] A. G. Liu, E. Musial, and M.-H. Chen, "Progressive reliability forecasting of service-oriented software," in *2011 IEEE international conference on web services*. IEEE, 2011, pp. 532–539.

[2] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010.

[3] A. Chug and S. Dhall, "Software defect prediction using supervised learning algorithm and unsupervised learning algorithm," in *Confluence 2013: The Next Generation Information Technology Summit (4th International Conference)*, 2013.

[4] S. G. Moser R, Pedrycz W, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 181–190.

[5] H. M. H, "Elements of software science," 1977.

[6] M. T. J, "A complexity measure," in *IEEE Transactions on software Engineering*. IEEE, 1977, pp. 308–320.

[7] K. C. F. Chidamber S R, "A metrics suite for object oriented design," in *IEEE Transactions on software engineering*. IEEE, 1994, pp. 476–493.

[8] T. L. Wang S, Liu T, "Automatically learning semantic features for defect prediction," in *2016 IEEE/ACM 38th International Conference on Software Engineering*. IEEE, 2016, pp. 297–308.

[9] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "A deep tree-based model for software defect prediction," *arXiv preprint arXiv:1802.00921*, 2018.

[10] Z. He, F. Peters, T. Menzies, and Y. Yang, "Learning from open-source projects: An empirical study on defect prediction," in *2013 ACM/IEEE international symposium on empirical software engineering and measurement*. IEEE, 2013, pp. 45–54.

[11] R. P. Thota M K, Shajin F H, "Survey on software defect prediction techniques," in *International Journal of Applied Science and Engineering*, 2020, pp. 331–344.

[12] A. Okutan and O. T. Yıldız, "Software defect prediction using bayesian networks," vol. 19, no. 1. Springer, 2014, pp. 154–181.

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

[14] C. Pan, M. Lu, B. Xu, and H. Gao, "An improved cnn model for within-project software defect prediction," vol. 9, no. 10. MDPI, 2019, p. 2138.

[15] H. Liang, Y. Yu, L. Jiang, and Z. Xie, "Seml: A semantic lstm model for software defect prediction," vol. 7. IEEE, 2019, pp. 83 812–83 824.

[16] X. Sun, X. Liu, B. Li, Y. Duan, H. Yang, and J. Hu, "Exploring topic models in software engineering data analysis: A survey," in *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. IEEE, 2016, pp. 357–362.

[17] T. T. Nguyen, T. N. Nguyen, and T. M. Phuong, "Topic-based defect prediction (nier track)," in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 932–935.

[18] A. Agrawal, W. Fu, and T. Menzies, "What is wrong with topic modeling? and how to fix it using search-based software engineering," *Information and Software Technology*, vol. 98, pp. 74–88, 2018.

[19] S. Paul and A. Prakash, "A framework for source code search using program patterns," in *IEEE Transactions on Software Engineering*, vol. 20. IEEE, 1994, pp. 463–475.

[20] C. L. G. W. Weimer, T. Nguyen and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31 st International Conference on Software Engineering*, 2009, pp. 364–374.

[21] L. M. M. S. I. D. Baxter, A. Yahin and L. Bier, "Clone detection using abstract syntax trees," in *Software Maintenance 1998. Proceedings. International Conference*, 1998, pp. 368–377.

[22] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[23] D. Hong, L. Gao, J. Yao, B. Zhang, A. Plaza, and J. Chanussot, "Graph convolutional networks for hyperspectral image classification," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 59, no. 7, pp. 5966–5978, 2020.

[24] L. Yao, C. Mao, and Y. Luo, "Graph convolutional networks for text classification," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 7370–7377.

[25] X. Wang, J. Li, L. Yang, and H. Mi, "Unsupervised learning for community detection in attributed networks based on graph convolutional network," *Neurocomputing*, vol. 456, pp. 147–155, 2021.

[26] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," vol. 30, 2017.

[27] Z. Wang, L. Zheng, Y. Li, and S. Wang, "Linkage based face clustering via graph convolution network," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 1117–1125.

[28] G. E. Hinton *et al.*, "Learning distributed representations of concepts," in *Proceedings of the eighth annual conference of the cognitive science society*, vol. 1. Amherst, MA, 1986, p. 12.

[29] Y. Chen, "Convolutional neural network for sentence classification," 2015.

[30] S. Ruder, P. Ghaffari, and J. G. Breslin, "A hierarchical model of reviews for aspect-based sentiment analysis," 2016.

[31] Y. Xing, X. Qian, Y. Guan, B. Yang, and Y. Zhang, "Cross-project defect prediction based on nlp methods," *Pattern Recognition Letters*, 2022.

[32] D. Tang, F. Wei, N. Yang, M. Zhou, T. Liu, B. Qin *et al.*, "Learning sentiment-specific word embedding for twitter sentiment classification." in *ACL (1)*, 2014, pp. 1555–1565.

[33] S. Bengio and G. Heigold, "Word embeddings for speech recognition," 2014.

[34] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[35] J. M. I. Blei D M, Ng A Y, "Latent dirichlet allocation," in *Journal of machine Learning research*, 2003, pp. 993–1022.

[36] H. T, "Probabilistic latent semantic indexing," in *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*. IEEE, 1999, pp. 50–57.

[37] T.-H. Chen, S. W. Thomas, M. Nagappan, and A. E. Hassan, "Explaining software defects using topic models," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2012, pp. 189–198.

[38] Y. Lu, Z. Zhao, G. Li, and Z. Jin, "Learning to generate comments for api-based code snippets," in *Software Engineering and Methodology for Emerging Domains*. Springer, 2017, pp. 3–14.

[39] C. Yu, G. Yang, X. Chen, K. Liu, and Y. Zhou, "Bashexplainer: Retrieval-augmented bash code comment generation based on fine-tuned codebert," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 82–93.

[40] G. Yang, K. Liu, X. Chen, Y. Zhou, C. Yu, and H. Lin, "Ccgir: Information retrieval-based code comment generation method for smart contracts," vol. 237. Elsevier, 2022, p. 107858.

[41] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *International conference on knowledge science, engineering and management*. Springer, 2015, pp. 547–553.

[42] X. Guo, Y. Yin, C. Dong, G. Yang, and G. Zhou, "On the class imbalance problem," in *2008 Fourth international conference on natural computation*, vol. 4. IEEE, 2008, pp. 192–201.

[43] R. Mohammed, J. Rawashdeh, and M. Abdullah, "Machine learning with oversampling and undersampling techniques: overview study and experimental results," in *2020 11th international conference on information and communication systems (ICICS)*. IEEE, 2020, pp. 243–248.

[44] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[45] T. Zhao, X. Zhang, and S. Wang, "Graphsmote: Imbalanced node classification on graphs with graph neural networks," in *Proceedings of the 14th ACM international conference on web search and data mining*, 2021, pp. 833–841.

[46] L. J. Hamilton W, Ying Z, "Inductive representation learning on large graphs," in *Advances in neural information processing systems*, 2017.

[47] N. J, "Survey on software defect prediction," in *Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology*, 2014.

[48] Z. X. Li Z, Jing X Y, "Progress on approaches to software defect prediction," in *Iet Software*. IEEE, 2018, pp. 161–175.

[49] R. P. Christopher, D. Manning and S. Hinrich, "Introduction to information retrieval," 2008.

[50] C. Zeng, C. Y. Zhou, S. K. Lv, P. He, and J. Huang, "Gcn2defect: Graph convolutional networks for smotetomek-based software defect prediction," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 69–79.

[51] R. Yedida and T. Menzies, "On the value of oversampling for deep learning in software defect prediction," in *IEEE Transactions on Software Engineering*, vol. 48. IEEE, 2022, pp. 3103–3116.

[52] X. Zhang, Y. Lu, and K. Shi, "Cb-path2vec: A cross block path based representation for software defect prediction," in *2020 IEEE 6th International Conference on Computer and Communications (ICCC)*. IEEE, 2020, pp. 1961–1966.

[53] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," in *IEEE Transactions on Software Engineering*. IEEE, 2018, pp. 5–24.

[54] X. Zhang, K. Ben, and J. Zeng, "Cross-entropy: A new metric for software defect prediction," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018, pp. 111–122.

[55] H. T. Shippey T, Bowes D, "Automatically identifying code features for software defect prediction: Using ast n-grams," in *Information and Software Technology*, 2019, pp. 142–160.

[56] A. Viet Phan, M. Le Nguyen, and L. Thu Bui, "Convolutional neural networks over control flow graphs for software defect prediction," in *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2017, pp. 45–52.

[57] C. S. Shen Z, "A survey of automatic software vulnerability detection, program repair, and defect prediction techniques," in *Security and Communication Networks*, 2020.

[58] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 297–308.

[59] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2017, pp. 318–328.

[60] A. V. Phan, M. Le Nguyen, and L. T. Bui, "Convolutional neural networks over control flow graphs for software defect prediction," in *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2017, pp. 45–52.

[61] S. Meilong, P. He, H. Xiao, H. Li, and C. Zeng, "An approach to semantic and structural features learning for software defect prediction," vol. 2020. Hindawi, 2020.

[62] L. Tang, C. Tao, H. Guo, and J. Zhang, "Software defect prediction via gcn based on structural and context information," in *2022 9th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 2022, pp. 310–319.

[63] C. Zhou, P. He, C. Zeng, and J. Ma, "Software defect prediction with semantic and structural information of codes based on graph neural networks," *Information and Software Technology*, vol. 152, p. 107057, 2022.