IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 29 December 2022, accepted 18 January 2023, date of publication 23 January 2023, date of current version 26 January 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3239266

RESEARCH ARTICLE

A Method-Level Defect Prediction Approach Based on Structural Features of Method-Calling Network

FENGYU YANG, HAOMING XU, PENG XIAO, FA ZHONG, AND GUANGDONG ZENG
School of Software, Nanchang Hangkong University, Nanchang, Jiangxi 330063, China

Corresponding author: Haoming Xu (xu1002601103@163.com)

This work was supported in part by the Key Research and Development Program of Jiangxi Province, China, under Grant 20202BBEL53002; and in part by the Natural Science Foundation of Jiangxi Province under Grant 20212BAB212009.

ABSTRACT Software defect prediction models help testers find program modules that have a high probability of having defects. A method-calling network can express the dependencies between methods in a program. Existing approaches do not sufficiently utilize method-calling network to characterize the structural features between methods. To address this problem, in this study, it is proposed for the first time that the characteristics of methods in a program are obtained by analyzing the method-calling network, and a new approach is proposed for defect prediction at the method-level. Specifically in this study, the method-calling network of the program was first constructed and the network metrics of the method-calling network were obtained. Next, the new network embedding technique (node2vec) was used to automatically encode the method-calling network structure into a low-dimensional vector to obtain the network embedding metrics. Finally, they were combined with code metrics to construct defect prediction models. We evaluated our approaches on 13 open-source software systems. The experimental results show that the proposed method improved the values of area under the receiver operating characteristic curve by 2.5% to 6.7% and Matthews correlation coefficient by 13% to 178.4% compared to the baselines. Therefore the method-calling network contains rich structural features between methods, and the structural features extend the features used for method-level defect prediction and further improve the performance of defect prediction models.

INDEX TERMS Software defect prediction, structural features, method-calling network, network embedding.

I. INTRODUCTION

In the software lifecycle, a software test is an important safeguard for a product before its release. Software testers must find potential defects within the software before they are released. However, the time and number of testers are often limited, which makes it unfeasible to examine all source code files. With defect prediction models, software quality assurance teams can expend more effort on code modules with a high propensity for defects and provide effective guidance to testers.

In software defect prediction studies, the granularity level of defect prediction ranges from coarse to fine, which includes packages [1], files [2], classes [3], methods [4], [5], and commits [6]. It may be laborious for testers to locate defects using coarse-grained defect prediction because reviewers need to examine all the methods in the package or file to find the defects. In contrast, a more fine-grained defect prediction can ease the work of software quality assurance personnel and thereby reduce the time spent in locating defects and modifying them [7]. In addition, several studies have shown that larger files are more prone to defects [8], resulting in an increased workload of reviewing the code. Hata et al. [5] found that fine-grained defect prediction outperforms coarse-grained defect prediction when considering the effort required to find defects. Pascarella et al. [9] replicated previous studies and found that all models showed a significant dip in performance

The associate editor coordinating the review of this manuscript and approving it for publication was Giuseppe Destefanis.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

when evaluated using a more realistic strategy and pointed out that method-level defect prediction remains a significant challenge.

Network metrics are obtained by analyzing software networks, and use dependencies between code modules to capture the structural characteristics of the network. Gong et al. [10] extracted network metrics from the software dependency network using social network analysis (SNA) and then combined them with code metrics to obtain a defect prediction model with better performance. They proposed that the ego network metrics and the global network metrics of the SNA metrics would exhibit different defect prediction performance. Chen et al. [11] found that most network metrics are significantly associated with a propensity for severe defects. Recently, network-embedding techniques have yielded good results in network node classification and link prediction tasks. Qu et al. [3] used the latest network embedding technique to extract structural features in a class dependency network, and they achieved good prediction results by combining structural features with code metrics. However, they used only the features obtained from network embedding to characterize the class-dependency network and ignored the information contained in other network metrics.

Inspired by previous research, this study analyzes the method-calling network of a program using the network metrics and network embedding technique [12]. They can analyze the method-calling network from different perspectives to obtain richer structural features. Then, the structural features are combined with code metrics to construct method-level defect prediction model.

The main contributions of this study are as follows:

- This study characterizes the dependencies of methods in a program by constructing a method-calling network, and obtains network embedding metrics and network metrics by analyzing the method-calling network.
- This study also proposes a new method for defect prediction at the method-level; a defect prediction model is constructed by combining the network embedding metrics, network metrics, and code metrics.
- Experiments on 13 open-source software systems have shown that the proposed method can improve the performance of the defect prediction model.

The remainder of this paper is organized as follows. Section II presents related work of software defect prediction and background on software metrics, method-calling network, and network embedding techniques. Section III presents our approach. Section IV presents the experiments and discusses the experimental results. Section V introduces the threats to the validity of the approach. Finally, Section VI concludes the study and gives future work.

II. RELATED WORK AND BACKGROUND

This section introduces related work of software defect prediction and background on software metrics, method-calling networks, and network embedding techniques.

A. SOFTWARE DEFECT PREDICTION

Software defect prediction [1], [2], [3], [4], [5] builds defect prediction models by uncovering historical data of software projects, aiming to identify modules that are at risk of defects in software projects. Software defect prediction is divided into different levels of prediction according to the software modules that need to predict defects. Giger et al. [4] constructed a method-level defect prediction model using the project's change metrics and code metrics and achieved good performance. In the software defect prediction contexts [10], within-project software defect prediction [3], [13] uses historical data from the same project and version to build training set and test set. Cross-version software defect prediction [2], [14] in the same project, using historical data from previous versions to build the training set and using data from new versions as the test set. Cross-project software defect prediction [15], [16] uses data from different projects to build training set and test set separately. Static features [3], [4] have been used extensively in the past studies. Code metrics are obtained by analyzing the source code to obtain statistical features. Process metrics [4] are obtained by analyzing the changes in software modules in historical software versions. Network metrics [17] obtain dependencies between modules by analyzing software dependency networks and are used for defect prediction. Qu et al. [18] use k-core decomposition on class dependency networks, found that classes with larger k values have higher defect risk, and then they proposed a new equation to make classes with high defect prediction risk rank higher in the classes defect risk list. With the development of deep learning, researchers have started to try to extract features that are not accessible by static features in the program. These features include the semantic features [19], [20], [21] of the program as well as the structural features, since two pieces of code with the same static features may have different contextual information as well as structural features. Wang et al. [22] analyzed the source code through the abstract syntax tree of the program, extracted semantic features by deep belief network, and obtained better defect prediction performance. Lin et al. [23] improve the performance of defect prediction by representing a simplified abstract syntax tree through two sequences and learning semantic features using a bi-directional long and short-term memory neural network. With the development of network embedding techniques [12], researchers have started to use network embedding techniques to characterize software networks [3]. While network metrics can statically analyze software networks, network embedding techniques automatically learn the structural features of software networks. Qu et al. [3] learned the structural features of class dependency networks through network embedding techniques and combined them with traditional software engineering features to construct defect prediction models, and their approach improved the performance of defect prediction.

B. SOFTWARE METRICS

Software defect prediction uses metric data from software projects to train machine learning models, and these defect prediction models can be used to predict program modules with a high defect risk factor [24].

1) CODE METRICS

Code metrics are statistics obtained after static analysis of a program and quantify various properties of the project source code. Commonly used code metrics include the Halstead measure for operators and operands [25], McCabe measure for dependencies [26], object-oriented CK measure [27], polymorphism factor, and MOOD measure for coupling factors [28]. Pham et al. [29] proposed a new set of metrics based on different metrics, and the new combined metric achieved good results in identifying defective classes.

2) NETWORK METRICS

The nodes in a software network can be evaluated from ego network metrics as well as global network metrics [10], [11]. In the method-calling network, the ego network metrics consider the direct calling relationship between methods; while the global network metrics consider the indirect influence of methods in the entire method-calling network.

Ego network metrics (EN): EN measures various properties of a node's domain, which consists of the node and the nodes directly connected to it. Each node has three types of EN: In, Out, and Un. The In type only considers the node and the nodes on which it depends, the Out type only considers the node and the nodes that depend on it, and Un type considers both In and Out types.

Global network metrics (GN): The metrics obtained by GN take into account a larger scope of the network compared to the domain, and they measure the role played by the node in the larger scope of the network.

C. METHOD-CALLING NETWORK

In object-oriented programming, the functions that must be implemented are often written as methods and can be called by each other to satisfy the specified requirements. The execution of a program is the result of many calls between the methods. In an object-oriented program P, the method-calling network (MCN) can be described as:

$$MCN_p = (V, E) \tag{1}$$

Here, V denotes the set of methods in the program P and the set of edges E denotes the calling relationship. $v_i, v_j \in V$ denotes the methods mi and mj; then, $vi \rightarrow vj$ denotes a call relationship between m_i and m_j. The study of complex networks plays an active role in the analysis of influence propagation and interactions. An MCN is a collection of method-calling graphs in which the nodes represent the individual methods in a program and the edges are the calling relationships.

Throughout the project, a series of method-calling graphs will form an MCN. In the MCN, the calling relationship between the methods can be observed (*A.m1→C.m3, A.m1→B.m2, B.m2→D.m4*).

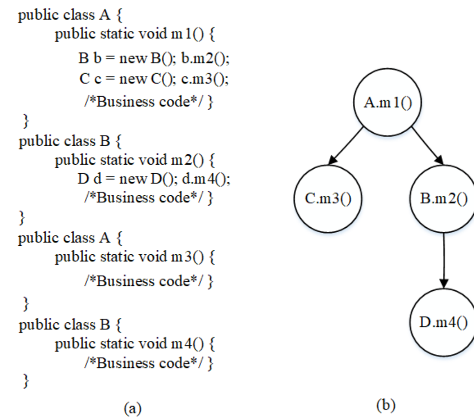


FIGURE 1. Method-calling network in the program.

Figure 1 shows a sample MCN diagram for a Java program. Figure 1(a) shows a Java program fragment, and Figure 1(b) shows its method-calling diagram [30].

D. NETWORK EMBEDDING TECHNIQUE

Networks are commonly used to express connections and interactions, and the analysis of networks is an area of research that continues to be of interest. The rich structural features of networks allow researchers to understand network systems better. Network embedding techniques can map the nodes in a network to low-dimensional vectors:

$$f : v_i \rightarrow y_i \in \mathbb{R}^d, \quad \forall v_i \in V \tag{2}$$

where d is much smaller than |V|, f holds some measures defined on the network N [31]. In recent years, researchers have proposed several network-embedding techniques [32], [33], [34], [35], [36].

The node2vec algorithm [12] is based on the natural language processing (NLP) model skip-gram and generates network domains for nodes using a flexible random wandering method. In a network, a path sampled by random wandering corresponds to a sentence in a text in NLP, and each network node corresponds to a word [3]. Node2vec uses stochastic gradient descent to optimize new network perception and neighborhood preservation goals efficiently. Node2vec uses a second-order random walk, which controls the transfer probability of a random wander through the parameters p and q. The two parameters control the wandering strategy and the speed of leaving the starting node. Specifically, the parameter p controls the possibility of returning immediately to the last sampled node. A large value of p avoids sampling nodes that have already been visited with high probability, while a small value of p causes the next sampling step to return to the previous node and maintains the sampling in the domain. Parameter q controls whether the sampling is biased toward nodes outside or inside the domain. When $q < 1$, the sampling process tends to sample nodes further away from the original node, approximating the depth-first

sampling; when $q > 1$, the sampling process remains within the domain, approximating the breadth-first sampling. The sampling schematic of node2vec is shown in Figure 2, where parameter α is the search deviation term that identifies the next node that will be sampled; its value is determined by parameters p and q.

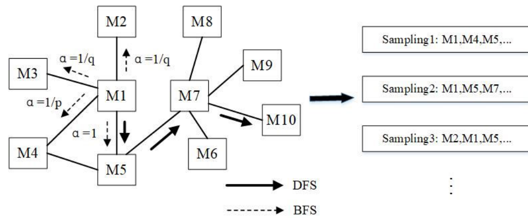


FIGURE 2. Sampling method of node2vec.

III. APPROACH

This section describes the flow of this approach in detail. This study first constructed the MCN and then analyzed the network to obtain the network metrics and network embedding metrics of the MCN. The framework of the general approach is illustrated in Figure 3.

For a software project that requires an analysis of defect propensity, the program's MCN is first constructed based on the call relationships between the methods. The network embedding technique node2vec is used in the MCN to learn the structural features of the program. It maps each method node to a low-dimensional vector to generate network embedding metrics. Then, the network metrics for each node are extracted in the MCN. In addition, the code metrics of the software project are used as part of the features, following which, the network embedding metrics, network metrics, and code metrics are used to form a comprehensive feature dataset. Finally, the features of each method in the project are input with defect labels into a machine-learning classifier to train the defect prediction model.

A. CONSTRUCT METHOD-CALLING NETWORK

This approach uses the MCN of a program. The structure and relationships of classes and methods in a project can be extracted by analyzing the source code, bytecode, and other file forms of the project to derive information about the MCN. This study used parsing bytecode files in Java programs to construct an MCN for software projects.

The algorithm describes the process of constructing an MCN, where the input of the algorithm is a collection of project files that can obtain information about method calls. The output result is a *Map*; the *key* to the *Map* is the method in the extracted program, and the value is the method called by each method in this project. Each step is briefly described as follows.

1) Extract class nodes. First, iterate through each input project file and record information regarding each class node, including the class name and method node information. The method node information includes the name of the method

Algorithm 1 Construct Method-Calling Network

Input: Set of project files $F = \{f_1, f_2, \ldots, f_n\}$.
Output: *Map* with the extracted methods as *key* and the set of methods they call as *Value.*
1 *Map D, E*
2 **for** i in $\{1, 2, \ldots, n\}$ **do**
3 $\quad cn$ = ParseClassByClassReader (f_i)
4 $\quad$ **for** j in $\{1, 2, \ldots, cn.methods.length\}$ **do**
5 $\quad\quad$ {Method:CallInvokeSet} = ParseMethodInsnNode(cn_j)
6 $\quad\quad$ $D = D\cup$ ({Method:CallInvokeSet})
7 $\quad$ **end**
8 **end**
9 **for** i in $\{D\}$ **do**
10 $\quad$ {MethodNode:CallMethodSet} = ParseCallMethod(D_i)
11 $\quad$ $E = E\cup$ {MethodNode:CallMethodSet}
12 **end**

belonging to this project and information about which other methods are called by this method (Line 3).

2) Extract method nodes. The recorded class node information is iterated and the method and method-calling information are extracted. Store each method with its method call information in a *Map*, with the method name as *key* and the set of method calls as *value* (Lines 5-6).

3) Save method-calling relationships. Iterate through the map that stores the method-calling information and resolve the methods that belong to this project. Based on the method-calling information, the extractable method node and its calling method node are saved to the map, with the parent node as the key and the set of called child methods as the value (Lines 10-11).

B. FEATURES EXTRACTION

In this study, the structural features of the method-calling network are obtained by extracting the network embedding metrics and the network metrics, and they analyze the method-calling network from different perspectives. The richer structural features help to characterize the method-calling network. The code metrics characterize each method by analyzing the source code. Therefore, these three metrics of methods are extracted as features for constructing defect prediction models in this study.

1) NETWORK EMBEDDING METRICS

In this study, node2vec is used to automatically encode the network nodes into a low-dimensional vector, and we input the MCN into the node2vec algorithm in the form of node pairs. By traversing the map that stores the MCN, we can represent the MCN as a collection of node pairs, where the first methods in the node pair represent the parent methods that call other methods and the second nodes represent the child methods that are called by the parent methods. The file that holds the collections can then be input into the node2vec algorithm. The MCN is input as an undirected network because it provides richer node sampling results, and thus, richer structural features. The output of node2vec is a low-dimensional vector representation for each method node. The features

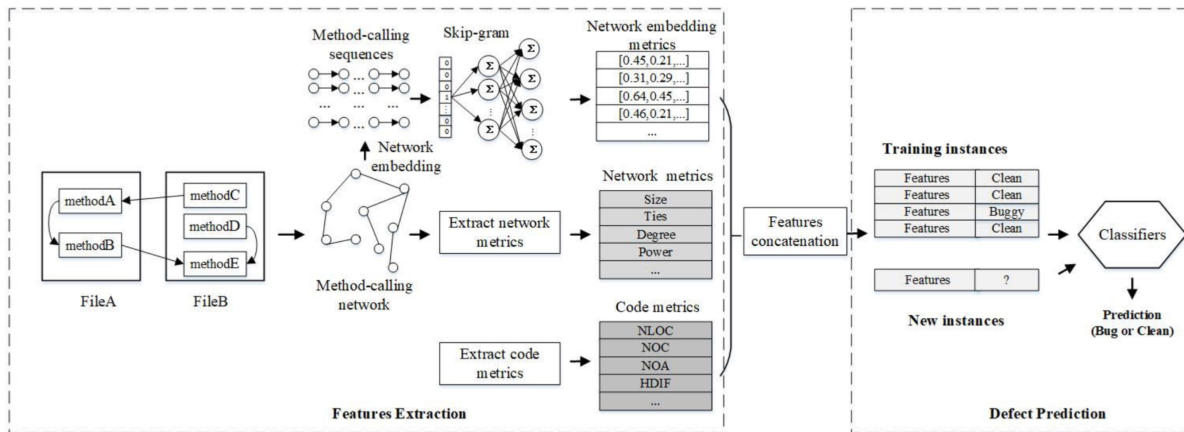


FIGURE 3. General framework of the approach.

obtained by network embedding are referred to as the network embedding metrics.

2) CODE METRICS

This study used 20 code metrics, including size, structural complexity, and Halstead metrics. Size metrics include the number of lines of code (NLOC) and the number of comments (NOC). Structural complexity metrics include the McCabes cyclomatic complexity (CC) for the method, Halstead metrics include: the Halstead length (HLTH) of the metric, the Halstead vocabulary (HVOC), and so on. The code metrics used in this study are presented in Table 1.

TABLE 1. Code metrics used in this study.

Symbol	Metrics Name
CC	McCabes cyclomatic complexity
NOCL	Number of comment lines
NOS	Number of statements
HLTH	Halstead length of the metric
HVOC	Halstead vocabulary
HEFF	Halstead effort
HBUG	Halstead prediction of the number of bugs
NLOC	Number of lines of code
NOC	Number of comments
NOA	Number of arguments
HDIF	Halstead difficulty
VDEC	Number of variables declared
CAST	Number of class casts
TDN	Total depth of nesting
HVOL	Halstead volume
NAND	Total number of operands
VREF	Number of variables referenced
NOPR	Total number of operators
MDN	Maximum depth of nesting
NEXP	Number of expressions

3) NETWORK METRICS

In this study, the network metrics of the MCN, including EN and GN, which have been widely used in previous studies [10], [11], are extracted. For each node in the network, three types of EN (In, Out, and Un) are calculated, as well as 11 GN. In total, 53 network metrics, including 36 EN and 17 GN, were extracted. Table 2 lists the network metrics and their descriptions [11].

C. HANDLING IMBALANCE AND PREDICTING DEFECTS

1) HANDLING IMBALANCE

Data imbalance is common in defect prediction [37], [38]. Defective data is often a small fraction of the entire dataset. In this study, we used a random oversampling technique to address the problem of data imbalance. Specifically, we randomly oversampled the data with defects in the training dataset to keep a ratio of defective data to data without defects as 1:1. We used random oversampling in the training set but not in the test set.

2) PREDICTING DEFECTS

In defect prediction, machine learning classification models are often used to predict the final outcome. This study used a random forest classifier that performs well in defect prediction within projects [39] and followed the above steps to obtain the network embedding metrics, code metrics, and network metrics of the methods. The three metrics were combined to form the features of the methods, and the combined features corresponded to the methods with defective labels one by one to form the dataset. Finally, the dataset was divided into training and test sets, which were used to train and validate the proposed model, respectively.

IV. EXPERIMENTS

A. STUDIED SYSTEM

The experimental dataset was obtained from the dataset published by Shippey et al. [40]. Due to the missing source files or dependency libraries incompatible in some projects, this study compiled 13 of these projects for the experiments. 5 of these 13 projects have been used in previous software defect prediction studies [3]. EclEmma is a free Java code coverage tool for Eclipse, HtmlUnit is a Java GUI-Less browser, Jmol is a molecular viewer for 3D chemical structures, OmegaT is a computer-assisted translation tool, Saros is an Eclipse plug-in for distributed assistance and twinning programming, UNICORE is a software suite for building federated systems, JMRI is a modeling tool, JPPF is an open-source grid computing solution, DrJava is a Java development environment, and GenoViz can be used for data visualization and sharing

TABLE 2. Network metrics used in this study.

Category	Metric	Description
Ego network measures	Size	Number of nodes directly connected to the node
	Ties	Number of edges in the EN
	Pairs	Number of possible directed ties in each EN, that is, Size × (Size − 1)
	Density	Ratio of the actual number of ties to the possible number of ties, that is, ties/pairs
	Broker	Number of indirectly connected pairs
	nBroker	Broker normalized by the number of pairs
	nWeakComp	Number of weak components in the EN
	pWeakComp	Number of weak components normalized by size
	EgoBetween	Percentage of shortest paths from neighbor to neighbor that pass-through ego
	2StepReach	Percentage of nodes within two steps normalized by size
	ReachEffic	2StepReach normalized by the sum of the sizes
Global network measures	Degree	Number of methods associated with a method
	Clus Coef	Extent of connectivity between neighboring methods of a method
	Closeness	Measure the shortest path from one method to other methods
	Reachability	Number of other methods a given method can reach
	Eigenvector	Assign relative scores to methods in the method-calling network
	Betweenness	Number of times a method appears in the shortest path of other methods
	Power	Measure the connections of methods in the method-calling network
	EffSize	Number of methods associated with a method
	Efficiency	Normalizes EffiSize to the total size of the network
	Constraint	Measures constraints of methods in the method-calling network
	Hierarchy	Measures the extent to which a method is centrally constrained in a method-calling network

in genomics. CDK is a software related to 3D rendering, JUMP is a software related to GIS, and jTDS is a software related to database. These systems and their MCN statistics are shown in Table 3. The data can be explained as follows: the *Systems* column shows the corresponding name of the dataset, and the *Version* column shows the corresponding systems and versions. $N_{MCN} \cap N_{MID}$ shows the number of methods that appear in the MCN and dataset. Methods with a calling relationship were recorded during the construction of a MCN, which was extracted by analyzing the bytecode file. Methods in the MCN whose names cannot be found in the dataset will not be recorded, for example, methods in some internal classes. Therefore, the intersection of the methods in the MCN with those in the dataset is less than the number of methods in the dataset. Here, the $\mathrm{E_{MCN}}$ column shows the number of edges extracted from the MCN and the P_{Defect} column shows the number of methods with defects in the MCN as a percentage of the overall MCN.

TABLE 3. Details of the studied systems.

Systems	Version	$\mathrm{N_{MCN} \cap N_{MID}}$	$\mathrm{E_{MCN}}$	$\mathrm{P_{Defect}}$
EclEmma	2.1	446	605	1.56
HtmlUnit	2008	1,440	2,830	13.19
DrJava	2008	1,062	7,730	14.12
GenoViz	5.4	4,551	9,779	12.96
Jmol	10	2,360	3,606	3.6
OmegaT	3.5	3,359	5,994	2.2
JMRI	2.0	5,885	11,689	1.27
Saros	1.0.6	755	1,156	5.03
UNICORE	1.4	832	929	8.65
JPPF	4.0	4,494	8,579	4.09
CDK	1.1	9,310	27,146	2.44
JUMP	1.5	7,171	15,613	1.42
jTDS	23072009	393	2,788	1.52

B. BASELINES

This study used the implementation of the node2defect framework of Qu et al. [3] in an MCN as one of the baselines. In addition, this study also used a single or combined form of network embedding metrics, CM, EN, and GN as baselines.

C. EVALUATION METRICS

1) METRICS FOR NON-EFFORT-AWARE EVALUATION

To evaluate these approaches, the following evaluation metrics frequently used by researchers [3], [7], [19] were also used in this study: Recall, Probability of false alarm (PF), Precision, F1 score (F1), area under the receiver operating characteristic curve (AUC) and Matthews correlation coefficient (MCC).

In the case of AUC, the horizontal coordinate indicates the false positive rate and the vertical coordinate indicates the recall rate. An AUC value between 0.5 and 1 indicates an overall better effect than a random guess; the closer the value is to 1, the better the effect.

The MCC indicates the correlation coefficient between actual and predicted classifications. As a balanced evaluation index, it is applied to class imbalanced data. The value of MCC is in the range of $[-1, 1]$, where -1 means the effect of the prediction result is completely wrong and 1 indicates the prediction result is correct.

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

$$FAR = \frac{FP}{FP + TN} \tag{4}$$

$$Precision = \frac{TP}{TP + FP} \tag{5}$$

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

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)\,(TP+FN)\,(TN+FP)\,(TN+FN)}} \tag{7}$$

Here, TP and TN are the number of methods correctly identified as defective and clean, respectively. FP is the number of actual clean methods predicted to be defective.

TABLE 4. AUC values of our NENCM and the baselines.

Systems	CM	NM	EN	GN	NEM	NENM	NECM	NENCM
EclEmma2.1	0.503	0.674	0.657	0.649	0.683	0.68	0.678	**0.693**
HtmlUnit2008	**0.58**	0.565	0.543	0.561	0.545	0.55	0.558	0.563
DrJava2008	0.583	0.549	0.534	0.556	0.539	0.555	0.583	**0.585**
GenoViz5.4	0.598	0.563	0.558	0.576	0.569	0.583	0.602	**0.604**
Jmol10	0.591	0.635	0.655	0.612	0.641	0.664	0.651	**0.679**
OmegaT3.5	**0.562**	0.527	0.544	0.516	0.527	0.542	0.541	0.555
JMRI2.0	0.564	0.584	0.56	**0.588**	0.531	0.528	0.536	0.539
Saros1.0.6	0.52	0.577	0.577	0.571	0.572	0.575	0.568	**0.581**
UNICORE1.4	0.55	0.51	0.503	0.505	0.567	0.568	0.568	**0.569**
JPPF4.0	0.554	0.55	0.547	0.542	0.552	0.555	0.549	**0.557**
CDK1.1	0.659	0.692	0.677	0.689	0.656	0.663	0.658	**0.695**
JUMP1.5	0.549	0.585	0.577	0.576	0.579	0.585	0.593	**0.631**
jTDS23072009	0.628	0.639	0.643	0.645	0.589	0.623	0.666	**0.694**
Average	0.572	0.588	0.582	0.583	0.58	0.59	0.596	**0.611**

FN is the number of actual defective methods predicted to be clean.

2) METRICS FOR EFFORT-AWARE EVALUATION

In this study, PofB20 [41] and Precision@20% [42] were used as effort-aware evaluation metrics. The effort-aware evaluation metrics evaluate the performance of the defect prediction model with a limited amount of effort. Developers want to get better benefits with limited effort. This effort can be expressed as the number of lines of code to be inspected. In this study, the number of lines of code to be inspected was set to 20% of the total number of lines of code. To calculate these two metrics, the probability of each module being predicted to be defective is first calculated based on the defect prediction model, and then the modules are ranked based on the probability. After inspecting 20% of the total number of lines of code according to the sorting results, the percentage of defective modules identified is PofB20 and the proportion of defective modules inspected to all inspected modules is Precision@20%.

D. VALIDATION SETTING

Cross-validation allows for the evaluation of defect prediction models [3]. The experiments in this study refer to Qu et al. [3] to divide the dataset in the ratio of 2:1, as well as the number of experimental repetitions. Specifically, in each experiment, two-third of the data was used for defect prediction model training and the remaining one-third for model testing. For each system, the experiments were repeated 30 times, thus reducing the bias caused by the random assignment of the experimental examples.

E. STATISTICAL ANALYSIS

Performance gain [7]. To compare our approach with the baselines, the performance difference between them in predicting defects was calculated using the following method:

$$\%ResultDiff = \frac{\sum (Res_{ours} - Res_{baseline})}{\sum Res_{baseline}} \quad (8)$$

Here, Res_{ours} indicates the average experimental results of our approach on different data sets, and $Res_{baseline}$ indicates the average experimental results of the baselines.

Statistical test. This study used the Wilcoxon signed-rank test, which is a nonparametric statistical test that allows a two-by-two comparison between two data distributions, to confirm the statistical differences between the experimental results of the groups. Specifically, the performance of the proposed approach was compared with those of other approaches. This study also performed effect size analysis. This study used Cliff's delta (δ), commonly used in previous studies, to quantify the difference between our approach and the baselines. The amount of difference can be negligible ($|\delta| < 0.147$), small ($0.147 \leq |\delta| < 0.33$), medium ($0.33 \leq |\delta| < 0.474$), large ($|\delta| \geq 0.474$).

F. EXPERIMENTAL RESULTS AND DISCUSSION

This section presents the results of our approach in different case systems, poses relevant research questions, and discusses the results for each question.

RQ1: Can richer structural features of method-calling network improve the performance of method-level defect prediction models?

In previous studies, researchers improved the performance of software defect prediction models by extracting network embedding metrics for class dependency network and combining them with code metrics. They did not use network metrics of class dependency network to build software defect prediction models. And previous studies have shown that network metrics have good performance in software defect prediction. The purpose of RQ1 is to investigate whether richer structural features of the method calling network can improve the performance of method-level defect prediction models. This study formalizes a hypothesis for RQ1: Richer structural features of the method-calling network cannot improve the performance of method-level defect prediction models.

To answer RQ1 and to test the proposed hypothesis, this study compares our approach with the baselines. This study used the implementation of the node2defect framework in the MCN as one of the baselines. The experiments refer

TABLE 5. MCC values of our NENCM and the baselines.

Systems	CM	NM	EN	GN	NEM	NENM	NECM	NENCM
EclEmma2.1	0.036	0.234	0.166	0.14	0.48	0.5	0.491	**0.539**
HtmlUnit2008	0.18	0.122	0.071	0.113	0.153	0.184	0.211	**0.22**
DrJava2008	0.179	0.096	0.057	0.1	0.118	0.191	0.233	**0.257**
GenoViz5.4	0.24	0.151	0.117	0.171	0.216	0.254	0.287	**0.298**
Jmol10	0.223	0.306	0.284	0.229	0.394	0.512	0.439	**0.527**
OmegaT3.5	0.143	0.162	0.04	0.017	0.149	0.222	0.233	**0.286**
JMRI2.0	0.125	0.104	0.046	0.069	0.128	0.165	0.183	**0.189**
Saros1.0.6	0.03	0.128	0.09	0.086	0.212	**0.268**	0.239	0.213
UNICORE1.4	0.124	0.016	0.004	0.013	0.184	0.197	**0.225**	0.204
JPPF4.0	0.144	0.128	0.075	0.079	0.201	0.227	0.23	**0.255**
CDK1.1	0.252	0.231	0.223	0.255	0.428	0.508	0.506	**0.575**
JUMP1.5	0.191	0.145	0.085	0.121	0.323	0.331	0.397	**0.433**
jTDS23072009	0.425	0.307	0.398	0.462	0.382	0.246	0.406	**0.615**
Average	0.176	0.163	0.127	0.142	0.259	0.292	0.313	**0.354**

TABLE 6. Comparison of non-effort-aware evaluation metrics results between our NENCM and baselines (Statistical Significance: ** $p < 0.01$, * $p < 0.05$, o $p \geq 0.05$.).

Ours vs	AUC		MCC		F1		Precision		Recall		PF	
Baseline	%Diff	E.Size	%Diff	E.Size	%Diff	E.Size	%Diff	E.Size	%Diff	E.Size	%Diff	E.Size
CM	6.7	M*	101.1	L**	35.7	M*	135.6	L**	22	So	-73.6	L**
NM	3.8	S*	116.4	L**	45.3	M**	197.3	L**	-1.3	No	-82.3	L**
EN	4.8	M**	178.4	L**	80.4	L**	270.2	L**	-3.8	No	-91.2	L**
GN	4.7	S**	148.5	L**	63.7	L**	241.2	L**	-5	So	-89.1	L**
NEM	5.2	M**	36.9	M**	19.1	S**	39.8	L**	28.6	S**	-31.9	S*
NENM	3.5	S**	21.1	S**	14.4	S*	11.7	M**	19.1	S**	-18.6	S**
NECM	2.5	S**	13	S*	16.1	S**	12.3	M**	12.5	S**	-23.7	S**

to node2defect as a combination of the network embedding metrics and the code metrics (NECM). In addition, a single or combined form of the code metrics (CM), EN, GN, and network embedding metrics (NEM) were used, such as NM (EN+GN), and NENM (NEM+NM), as the baselines. Our approach combines all the metrics, which we call NENCM (NEM+NM+CM). We used the default parameters of node2vec in our experiments and set the dimensionality of the generated vectors to 32.

As shown in the Table 4 and Table 5, NENCM has the highest AUC values on 10 projects and NENCM has the highest MCC values on 11 projects, shown in bold. As shown in Figure 4. The PF values of NENCM are overall lower than those of the baselines, the precision and F1 values of NENCM are overall higher than those of the baselines, and the recall values of NENCM are overall higher than those of most of the baselines. The values of recall for NM, EN and GN are a little higher than those of NENCM, but NM, EN and GN have lower precision and higher probability of false alarm. Our NENCM has better performance overall.

Table 6 demonstrates the comparison of the results of our NENCM with those of the baselines for the non-effort-aware evaluation metrics. Our NENCM improves AUC values by 2.5% to 6.7% and MCC values by 13% to 178.4% compared to the baselines. The results of the Wilcoxon signed-rank test show significant differences in AUC, MCC, F1, precision, and PF values for our NENCM compared to baselines (p-value < 0.05). Table 6 also shows the results of the effect size analysis. The precision values of our NENCM show large or medium amounts of difference compared to the baselines.

TABLE 7. Comparison of effort-aware evaluation metrics results between our NENCM and baselines (Statistical Significance: ** $p < 0.01$, * $p < 0.05$, o $p \geq 0.05$.).

Ours vs	PofB20		Precision@20%	
Baseline	%Diff	E.Size	%Diff	E.Size
CM	51.8	L**	81	L**
NM	46.3	L**	108.4	L**
EN	36.3	L**	140.7	L**
GN	47.6	L**	111	L**
NEM	19.1	M*	50.8	M**
NENM	19.3	M**	46.2	S**
NECM	31.8	L**	33.7	S**

Our NENCM values of AUC, MCC, F1, and PF show large or medium amounts of difference compared to some of the baselines. The amount of difference in recall values of our NENCM compared to the baselines is negligible or small.

As shown in Figure 5, the values of PofB20 and Precision@20% of our NENCM are overall higher than the baselines. As shown in Table 7, the results of the Wilcoxon signed-rank test show significant differences in PofB20, Precision@20% values for our NENCM compared to baselines (p-value < 0.05). the PofB20 values of our NENCM show large or medium amounts of difference compared to all baselines. Our Precision@20% values for NENCM show large or medium amounts of difference compared to most baselines.

MCNs are established by obtaining dependencies between the methods. The network metrics count the values of various attributes of each node in an MCN. Homogeneity and structural equivalence have an important role in the task of

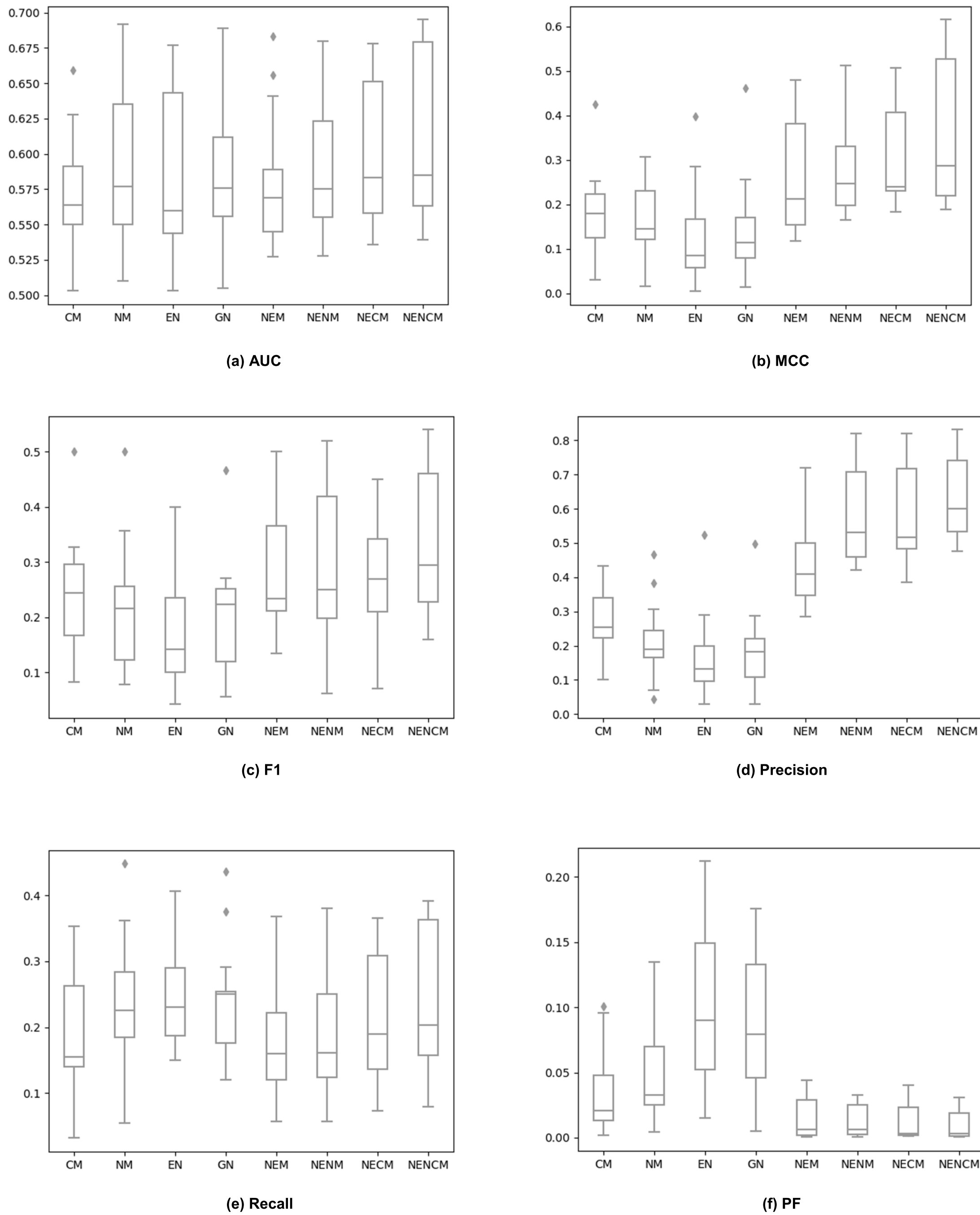
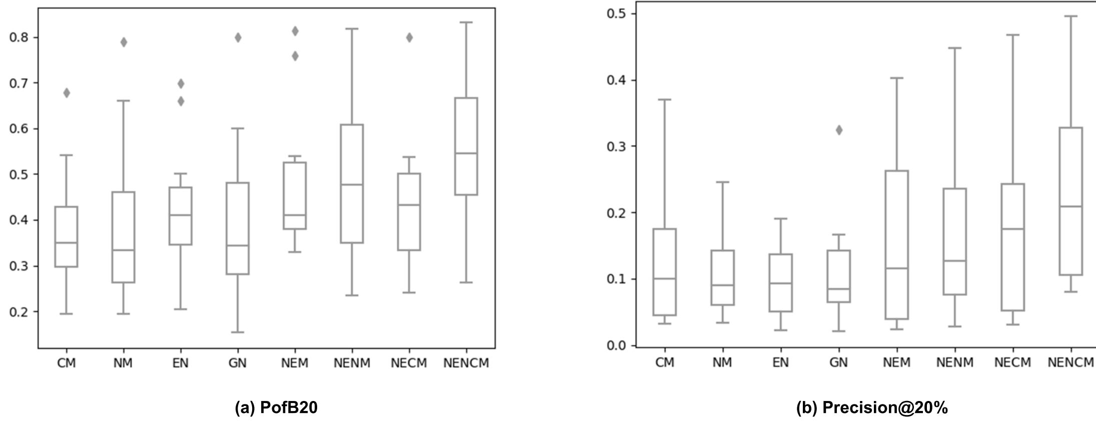


FIGURE 4. Distribution of the values of AUC, MCC, F1, Precision, Recall, and PF for our NENCM with the baselines.

prediction of network nodes [12]. Breadth-first search (BFS) and depth-first search (DFS) achieve structural equivalence and homogeneity, respectively [12]. Therefore, in a method-calling network, similar structural properties of methods can be obtained using the network-embedding technique [3]. Methods with similar structural defect risks in the low-dimensional vector are similar [3]. Code metrics count the values of the various properties of each method in the program source code and the different characteristics of the code. The network metrics extract features of the MCN from the perspective of network analysis. These features are richer than the NEM, for example, power (which measures the connections of methods in the method-calling network). The combination of network metrics and NEM can better characterize the MCN, and the combination of CM can improve the performance of defect prediction.

This study responds to the hypothesis and answers RQ1 based on the experimental results. This study rejects



(a) PofB20

(b) Precision@20%

FIGURE 5. Distribution of the values of PofB20, Precision@20% for our NENCM with the baselines.

TABLE 8. Comparison of the results of the non-effort-aware evaluation metrics between approaches that combined other metrics with NEM under different parameter settings and our NENCM (Statistical Significance: ** $p < 0.01$, * $p < 0.05$, o $p \geq 0.05$.).

Ours vs	AUC		MCC		F1		Precision		Recall		PF	
Baseline	%Diff	E.Size	%Diff	E.Size	%Diff	E.Size	%Diff	E.Size	%Diff	E.Size	%Diff	E.Size
DC	4.4	S*	23.3	S*	26.1	M**	18.9	M**	27.1	S*	-24.8	N**
BC	3.4	S**	20.9	S**	27.5	M**	20.4	M**	19.7	S**	-23.6	N*
ED	4.7	M**	33.5	M*	26.4	M**	20.2	M**	30.3	S**	-20	S**
GD	3.8	M**	29.3	S**	18	S**	16.2	M**	21.9	S**	-29.5	S**
EB	3.5	S**	17.1	S**	21.8	S**	17.4	M**	21.1	S**	-11.6	So
GB	3.6	M**	25	M**	19	S**	14	M**	21.2	S**	-20.7	N*
EBC	4	S**	19.2	S**	16.2	S**	7.5	S**	25.2	S**	-15.5	No
GBC	3.8	S**	21	S*	16.9	S**	15.2	M**	23.6	S**	-15.7	N*
EDC	4	M*	18.2	So	17.8	S**	14	S**	24.6	S*	-16.9	N*
GDC	4	S**	28.1	S*	24.2	M**	15.9	M**	17.9	S*	-22.7	N**

the hypothesis of RQ1. The richer structural features of the method call network help to improve the performance of the method-level defect prediction model, and the improvement is not so significant under the non-effort-aware evaluation metrics and is significant under the effort-aware evaluation metrics.

RQ2: How are the performances of the defect prediction models after combining other metrics with NEM under different parameter settings?

The purpose of RQ2 is to investigate the influences of combining different metrics with NEM under different parameter settings on the performances of defect prediction models. This study formalizes a hypothesis for RQ2: The defect prediction models performances can be improved by combining other metrics with NEM under different parameter settings. The network metrics include EN and GN, which reflect the importance of nodes in the ego and global networks, respectively. Node2vc controls the sampling of nodes in the network through two important parameters, p and q. Here, p determines the probability of returning to the last sampled node in the sampling process, and q determines whether the sampling process is biased toward DFS or BFS, as described in Section II-D. Different NEM were obtained by setting different values of p and q. The value of p was set between 1 and q, as this does not always return the last sampled node during node sampling, making node sampling richer. The value of q was set to either 0.25 or 4 because node2vec will achieve a good performance with parameters p and q in the interval of 0.25 and 4 [7]. When the value of p is 0.5 and the value of q is 0.25, the NEM is called the DFS network embedding metric (DFS-NEM). When the value of p is 2 and the value of q is 4, the NEM is called the BFS network embedding metric (BFS-NEM). Subsequently, different NEM were combined with CM or different network metrics. Their specific combinations are the following: EN+DFS-NEM+CM (EDC), GN+DFS-NEM+CM (GDC), EN+BFS-NEM+CM (EBC), GN+BFS-NEM+CM (GBC), DFS-NEM+CM (DC), BFS-NEM+CM (BC), EN+DFS-NEM (ED), GN+DFS-NEM (GD), EN+BFS-NEM+CM (EB), and GN+DFS-NEM (GB).

NEM-DFS favors depth-first sampling in the node-sampling process, which allows method nodes that are not directly connected in the MCN to be close to each other in the NEM. This can be considered as a structural feature of the method nodes in the global network obtained by means of network embedding. NEM-BFS favors breadth-first sampling in the node-sampling process, which allows method nodes in the MCN and their domain method nodes to be close to each other in the NEM. We can consider this a structural feature of the method nodes in the ego network obtained by means of network embedding. The network metrics include EN and GN, which reflect the importance of the

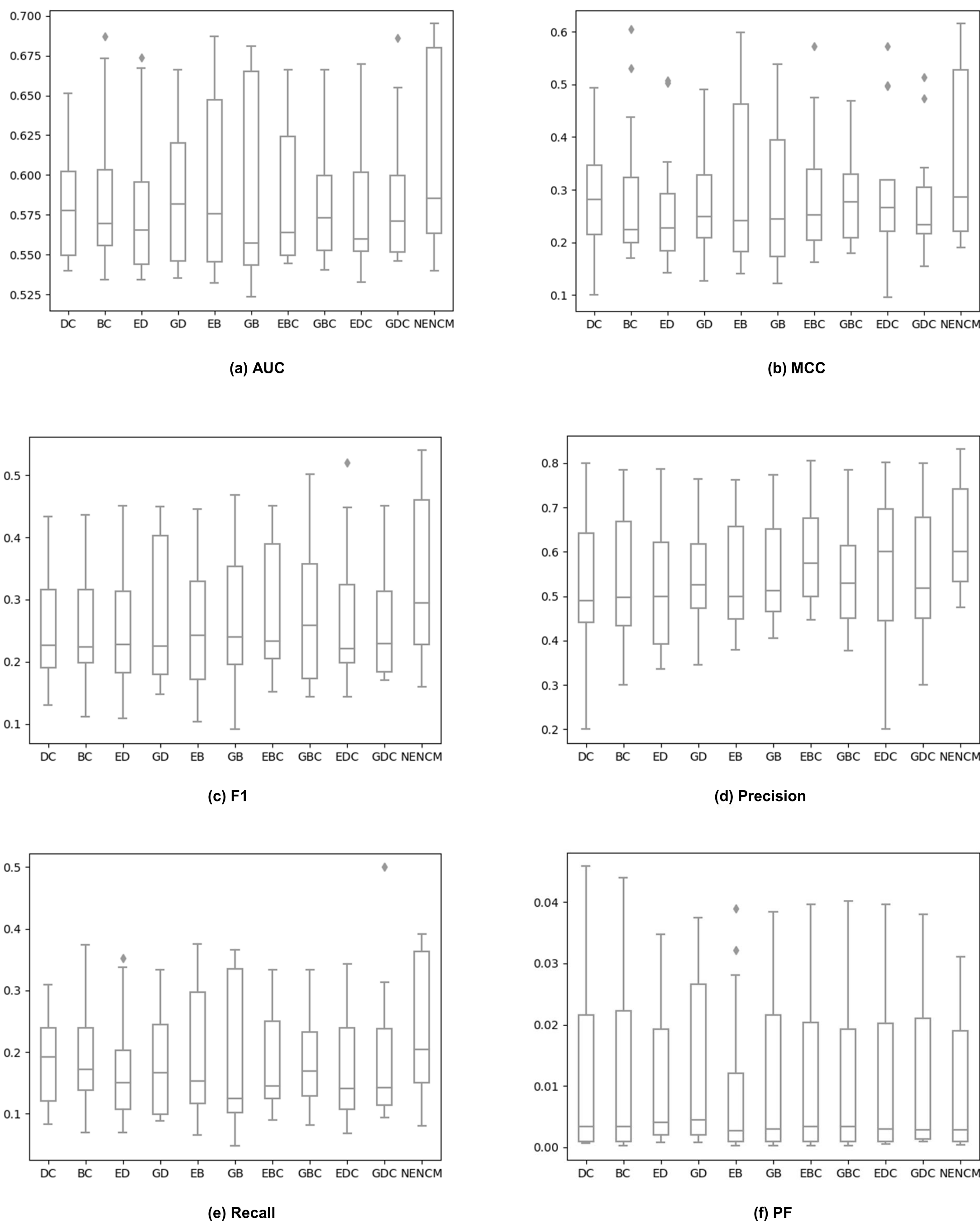


FIGURE 6. Distribution of the values of AUC, MCC, F1, Precision, Recall, and PF for the approaches that combined other metrics with NEM under different parameter settings and our NENCM.

method nodes in the ego and global networks, respectively. ED denotes the combination of EN and global network features obtained from the network embedding, and GD denotes only the combined features of the global network. GB denotes the combination of the GN and ego network features obtained from the network embedding, and EB denotes only the combined features of the ego network. EDC, GDC, EBC, and GBC are combined methods after adding the CM.

As shown in Figure 6, The AUC, MCC, F1, precision, and recall values of the approaches after combination of the other metrics with NEM under different parameter settings are overall lower than those of our NENCM. The PF values of the approaches after combination of the other metrics with NEM under different parameter settings are overall similar to the PF values of our NENCM. As shown in Table 8, in the results of AUC, MCC, F1, and precision, our NENCM shows small or medium amounts of differences compared to the

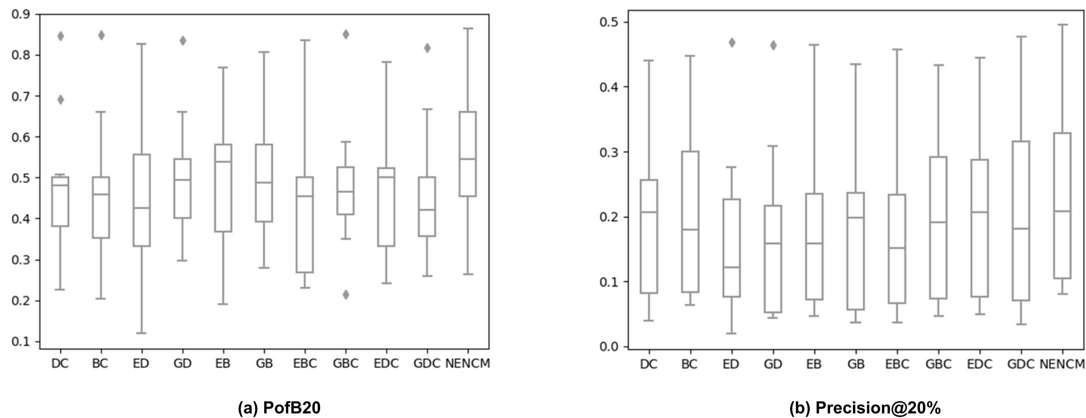


FIGURE 7. Distribution of the values of PofB20, Precision@20% for the approaches that combined other metrics with NEM under different parameter settings and our NENCM.

TABLE 9. Comparison of the results of the effort-aware evaluation metrics between approaches that combined other metrics with NEM under different parameter settings and our NENCM (Statistical Significance: ** $p < 0.01$, * $p < 0.05$, o $p \geq 0.05$.).

Ours vs Baseline	PofB20 %Diff	PofB20 E.Size	Precision@20% %Diff	Precision@20% E.Size
DC	21	M**	21.9	S**
BC	22.6	M**	16.8	S**
ED	30.9	M**	45.8	S**
GD	15.2	S**	45.8	M**
EB	17.1	S**	29.3	S**
GB	13.7	S**	37.4	S**
EBC	31.3	M**	39.4	S**
GBC	19.3	M**	15.1	S**
EDC	19	M**	19.4	S**
GDC	25.1	M**	19.6	S**

approaches that combined other metrics with NEM under different parameter settings, and the results of the Wilcoxon signed-rank test indicate significant differences (p-value < 0.05). In the results of recall and PF, our NENCM shows negligible or small amounts of difference.

As shown in Figure 7, Our NENCM has higher values of PofB20, Precision@20% overall compared to other approaches. As shown in Table 9, in the results of PofB20, Precision@20%, the amounts of differences in our NENCM are small or medium compared to the other approaches, and the results of the Wilcoxon signed-rank test indicate significant differences (p-value < 0.05).

From the experimental results, it can be observed that combining other metrics with the NEM under different parameter settings does not improve the performance of the defect prediction models, with NENCM still showing an overall better performance. The network metrics integrate the EN and the GN and add the NEM obtained by node2vec to balance DFS and BFS. Finally, the integrated features obtained by connecting the CM can better characterize the methods in the program, thus improving the performance of the defect prediction model.

This study responds to the hypothesis and answers RQ2 based on the experimental results. This study rejects the hypothesis of RQ2. The performances of the defect prediction models cannot be improved by combining other metrics with NEM under different parameter settings.

G. TIME CONSUMPTION

To understand the time complexity of our approach, this study measured the time of our approach and the baselines in building defect prediction models and predicting defects. This study did not calculate the time for other processes, including constructing method-calling network, obtaining network metrics, obtaining network embedding metrics, and obtaining code metrics. This study measured the average time consumed by all projects for 30 times of repeating the experiments on different approaches, with the same number of repetitions as in the previous parts. We calculated the average time consumed by the different approaches on all studied projects. The time calculations were performed on the computer with Intel Core i5 2.5GHz and 16GB of RAM.

TABLE 10. Computation time in building models and predicting defects of our NENCM and baselines.

Approach	Time (s)
CM	0.45
NM	0.49
EN	0.41
GN	0.39
NEM	0.9
NENM	0.95
NECM	0.98
NENCM	1.03

As shown in Table 10, CM, NM, EN, and GN spend less time in building models and predicting defects. The number of input features for these approaches are smaller. The average time consumed by our NENCM is 1.03 seconds, without much increase. As shown in Table 11, the approaches that combined other metrics with NEM under different parameter settings consumed similar time as our NENCM in building models and predicting defects.

TABLE 11. Computation time in building models and predicting defects of the approaches that combined other metrics with NEM under different parameter settings and our NENCM.

Approach	Time (s)
DC	0.9
BC	0.91
ED	0.93
GD	0.96
EB	0.91
GB	0.96
EBC	0.97
GBC	0.98
EDC	0.98
GDC	1.0
NENCM	1.03

V. THREATS TO VALIDITY

A. INTERNAL VALIDITY

Several parameters in node2vec must be set manually and the setting of these parameters may affect the effectiveness of the prediction model. This study aims to investigate the extraction of structural features in MCNs through network embedding techniques and network metrics and to improve the performance of defect prediction at the method-level. Therefore, the dimensionality of the method mapping to low-dimensional vectors in this study was referenced from Qu et al. [3] in the class dependency network. In addition, this study used the various parameters in the publicly available source code of Grover et al. [12] in the embedding process of the network nodes using the node2vec technique.

B. EXTERNAL VALIDITY

The proposed approach may exhibit inconsistent performance in other software systems. To mitigate such threats, this study conducted some experiments using 13 software systems disclosed by Shippey et al. [40]. The positive results for these systems suggest that the adaptability of the proposed approach will also be stronger. The performance of our approach is better overall. The results of Wilcoxon signed rank test and effect size analysis also show significant performance improvement of our approach under some evaluation metrics, which to some extent proves the generalizability of our approach.

VI. CONCLUSION AND FUTURE WORK

This study characterized the dependencies between methods by constructing an MCN and analyzed it to obtain the NEM and network metrics of the methods. The CM were then combined to achieve method-level defect prediction. Specifically, network embedding technology mapped each method in the MCN to a low-dimensional vector space and automatically learned the structural features in the MCN. The network metrics could obtain more static features by analyzing the MCN, and a combination of the two metrics could better characterize the MCN. Finally, the two metrics are combined with CM to construct a defect prediction model. The experimental results using 13 open-source systems show that the proposed method has a better overall predictive performance than the baselines. In future research, we will focus on combining structural and semantic features in method-calling networks at the method-level. The application of this study to defect prediction across versions and projects can also be explored.

REFERENCES

[1] Y. Kamei, S. Matsumoto, A. Monden, K.-I. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–10.

[2] S. Wang, J. Wang, J. Nam, and N. Nagappan, "Continuous software bug prediction," in *Proc. 15th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Oct. 2021, pp. 1–12.

[3] Y. Qu, T. Liu, J. Chi, Y. Jin, D. Cui, A. He, and Q. Zheng, "node2defect: Using network embedding to improve software defect prediction," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, Sep. 2018, pp. 844–849.

[4] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proc. ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Sep. 2012, pp. 171–180.

[5] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 200–210.

[6] S. Tabassum, L. L. Minku, D. Feng, G. G. Cabral, and L. Song, "An investigation of cross-project learning in online just-in-time software defect prediction," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, 2020, pp. 554–565.

[7] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, "Predicting defective lines using a model-agnostic technique," *IEEE Trans. Softw. Eng.*, vol. 48, no. 5, pp. 1480–1496, May 2022.

[8] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 340–355, Apr. 2005.

[9] L. Pascarella, F. Palomba, and A. Bacchelli, "Re-evaluating method-level bug prediction," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2018, pp. 592–601.

[10] L. Gong, G. K. Rajbahadur, A. E. Hassan, and S. Jiang, "Revisiting the impact of dependency network metrics on software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 48, no. 12, pp. 5030–5049, Dec. 2022.

[11] L. Chen, W. Ma, Y. Zhou, L. Xu, Z. Wang, Z. Chen, and B. Xu, "Empirical analysis of network measures for predicting high severity software faults," *Sci. China Inf. Sci.*, vol. 59, no. 12, pp. 1–18, Dec. 2016.

[12] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, San Francisco, CA, USA, Jun. 2016, pp. 855–864.

[13] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, Jan. 2017.

[14] Z. Xu, S. Li, Y. Tang, X. Luo, T. Zhang, J. Liu, and J. Xu, "Cross version defect prediction with representative data via sparse subset selection," in *Proc. 26th Conf. Program Comprehension*, May 2018, pp. 132–143.

[15] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "HYDRA: Massively compositional model for cross-project defect prediction," *IEEE Trans. Softw. Eng.*, vol. 42, no. 10, pp. 977–998, Oct. 2016.

[16] C. Ni, X. Xia, D. Lo, X. Chen, and Q. Gu, "Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction," *IEEE Trans. Softw. Eng.*, vol. 48, no. 3, pp. 786–802, Mar. 2022.

[17] W. Ma, L. Chen, Y. Yang, Y. Zhou, and B. Xu, "Empirical analysis of network measures for effort-aware fault-proneness prediction," *Inf. Softw. Technol.*, vol. 69, pp. 50–70, Jan. 2016.

[18] Y. Qu, Q. Zheng, J. Chi, Y. Jin, A. He, D. Cui, H. Zhang, and T. Liu, "Using K-core decomposition on class dependency networks to improve bug prediction model's practical performance," *IEEE Trans. Softw. Eng.*, vol. 47, no. 2, pp. 348–366, Feb. 2021.

[19] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 46, no. 12, pp. 1267–1293, Dec. 2020.

[20] J. Chen, K. Hu, Y. Yu, Z. Chen, Q. Xuan, Y. Liu, and V. Filkov, "Software visualization and deep transfer learning for effective software defect prediction," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, Jun. 2020, pp. 578–589.

[21] J. Xu, F. Wang, and J. Ai, "Defect prediction with semantics and context features of codes based on graph representation learning," *IEEE Trans. Rel.*, vol. 70, no. 2, pp. 613–625, Jun. 2021.

[22] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng. (ICSE)*, May 2016, pp. 297–308.

[23] J. Lin and L. Lu, "Semantic feature learning via dual sequences for defect prediction," *IEEE Access*, vol. 9, pp. 13112–13124, 2021.

[24] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007.

[25] M. Halsted, *Elements of Software Science* (Operating and Programming Systems Series). New York, NY, USA: Elsevier, 1977. [Online]. Available: https://dl.acm.org/doi/book/10.5555/540137

[26] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[27] M. Jureczko and D. D. Spinellis, "Using object-oriented design metrics to predict software defects," in *Proc. Models Methods Syst. Dependability*, 2010, pp. 69–81.

[28] R. Harrison, S. J. Counsell, and R. V. Nithi, "An evaluation of the MOOD set of object-oriented software metrics," *IEEE Trans. Softw. Eng.*, vol. 24, no. 6, pp. 491–496, Jun. 1998.

[29] V. Pham, C. Lokan, and K. Kasmarik, "A better set of object-oriented design metrics for within-project defect prediction," in *Proc. Eval. Assessment Softw. Eng.*, Apr. 2020, pp. 230–239.

[30] Y. Qu, X. Guan, Q. Zheng, T. Liu, J. Zhou, and J. Li, "Calling network: A new method for modeling software runtime behaviors," *ACM SIGSOFT Softw. Eng. Notes*, vol. 40, no. 1, pp. 1–8, 2015.

[31] P. Goyal and E. Ferrara, "Graph embedding techniques, applications, and performance: A survey," *Knowl.-Based Syst.*, vol. 151, pp. 78–94, Jul. 2018.

[32] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online learning of social representations," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2014, pp. 701–710.

[33] B. Perozzi, V. Kulkarni, H. Chen, and S. Skiena, "Don't walk, skip! Online learning of multi-scale network embeddings," in *Proc. IEEE/ACM Int. Conf. Adv. Social Netw. Anal. Mining*, Jul./Aug. 2017, pp. 258–265.

[34] D. Wang, P. Cui, and W. Zhu, "Structural deep network embedding," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, San Francisco, CA, USA, Aug. 2016, pp. 1225–1234.

[35] S. Cao, W. Lu, and Q. Xu, "GraRep: Learning graph representations with global structural information," in *Proc. 24th ACM Int. Conf. Inf. Knowl. Manage.*, Oct. 2015, pp. 891–900.

[36] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *Proc. 24th Int. Conf. World Wide Web*, May 2015, pp. 1067–1077.

[37] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid, "Class imbalance evolution and verification latency in just-in-time software defect prediction," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 666–676.

[38] S. Feng, J. Keung, X. Yu, Y. Xiao, K. E. Bennin, M. A. Kabir, and M. Zhang, "COSTE: Complexity-based oversampling technique to alleviate the class imbalance problem in software defect prediction," *Inf. Softw. Technol.*, vol. 129, Jan. 2021, Art. no. 106432.

[39] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, May 2015, pp. 789–800.

[40] T. Shippey, T. Hall, S. Counsell, and D. Bowes, "So you need more method level datasets for your software defect prediction? Voilà!" in *Proc. 10th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Sep. 2016, pp. 1–6.

[41] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2013, pp. 279–289.

[42] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empirical Softw. Eng.*, vol. 24, no. 5, pp. 2823–2862, Oct. 2019.

FENGYU YANG received the master's degree from the Zhejiang University of Technology, in 2006. He is currently an Associate Professor with the School of Software, Nanchang Hangkong University, China. His main research interests include software testing, data mining, and deep learning.

HAOMING XU is currently pursuing the master's degree with the School of Software, Nanchang Hangkong University. His research interests include software defect prediction and software testing.

PENG XIAO was born in 1988. He received the Ph.D. degree from the School of Reliability and System Engineering, Beihang University, in 2018. His current research interests include software testing, defect prediction, and software reliability.

FA ZHONG is currently pursuing the master's degree with the School of Software, Nanchang Hangkong University. His research interests include software defect prediction and machine learning.

GUANGDONG ZENG is currently pursuing the master's degree with the School of Software, Nanchang Hangkong University. His research interests include software defect prediction and association rules.