Information and Software Technology 152 (2022) 107057

Contents lists available at ScienceDirect

Information and Software Technology

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

Software defect prediction with semantic and structural information of codes based on Graph Neural Networks

Chunying Zhou, Peng He [*], Cheng Zeng, Ju Ma

School of Computer Science and Information Engineering, Hubei University, Wuhan, China

A R T I C L E I N F O

Keywords:
Software defect prediction
Class Dependency Network
Convolutional Neural Network
Graph Convolutional Network

A B S T R A C T

Context: Most defect prediction methods consider a series of traditional manually designed static code metrics. However, only using these hand-crafted features is impractical. Some researchers use the Convolutional Neural Network (CNN) to capture the potential semantic information based on the program's Syntax Trees (ASTs). In recent years, leveraging the dependency relationships between software modules to construct a software network and using network embedding models to capture the structural information have been helpful in defect prediction. This paper simultaneously takes the semantic and structural information into account and proposes a method called CGCN.
Objective: This study aims to validate the feasibility and performance of the proposed method in software defect prediction.
Method: Abstract Syntax Trees and a Class Dependency Network (CDN) are first generated based on the source code. For ASTs, symbolic tokens are extracted and encoded into vectors. The numerical vectors are then used as input to the CNN to capture the semantic information. For CDN, a Graph Convolutional Network (GCN) is used to learn the structural information of the network automatically. Afterward, the learned semantic and structural information are combined with different weights. Finally, we concatenate the learned features with traditional hand-crafted features to train a classifier for more accurate defect prediction.
Results: The proposed method outperforms the state-of-the-art defect prediction models for both within-project prediction (including within-version and cross-version) and cross-project prediction on 21 open-source projects. In general, within-version prediction achieves better performance in the three prediction tasks.
Conclusion: The proposed method of combining semantic and structural information can improve the performance of software defect prediction.

1. Introduction

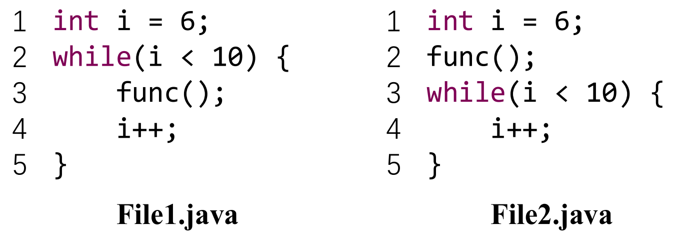
Software Defect Prediction (SDP) techniques aim to predict the defect-prone modules in the software system, which can efficiently help software developers to allocate the testing resources better [1–3]. Accurately predicting defective components in the software life cycle and fixing bugs in time can significantly improve the software quality [4]. The existing studies usually use manually designed static metrics to build predictors. These traditional hand-crafted features mainly focus on the statistical characteristics of programs, such as Halstead features based on the number of operators and operands, McCabe features based on dependencies, and CK features for object-oriented programs [5,6]. However, the hand-crafted features that perform well for a certain project may perform poorly in other projects [7].

Recently, some researchers pointed out that in addition to the features represented by a series of code metrics, the source code is rich in syntax and semantic information as a unique text. Thus, ignoring the context information may significantly impact understanding the program. For example, both java files in Fig. 1 contain a *while* statement and a *func()* function call. The two files have some same static features, such as lines of code and the number of function calls. However, the code execution in both files is different due to the different positions of the *func()*, thus resulting in different semantic information. Therefore, it is self-evident that predicting defects based on the features expressed by traditional metrics has serious shortcomings.

Abstract Syntax Tree (AST) is an abstract representation of the syntax structure of a source code capable of describing a program structure in the form of a tree, in which each node represents a construct or symbol in the source code. The differences in the two java files shown in Fig. 1 are distinguishable by their corresponding ASTs. Indeed, the authors in [5–9] improved the performance of software

* Corresponding author.
E-mail addresses: zcy9838@stu.hubu.edu.cn (C. Zhou), penghe@hubu.edu.cn (P. He), zc@hubu.edu.cn (C. Zeng), dcsxan@nus.edu.sg (J. Ma).

https://doi.org/10.1016/j.infsof.2022.107057
Received 8 March 2022; Received in revised form 18 August 2022; Accepted 23 August 2022
Available online 31 August 2022
0950-5849/© 2022 Elsevier B.V. All rights reserved.



```
int i = 6;
while(i < 10) {
    func();
    i++;
}
```

File1.java

```
int i = 6;
func();
while(i < 10) {
    i++;
}
```

File2.java

Fig. 1. A typical example.

defect prediction by successfully learning the semantic information from the ASTs of source code.

As an artifact, the software can also be abstracted into a coarse-grained network structure based on the dependencies between classes, namely a Class Dependency Network (CDN) scheme. In CDN, each class is regarded as a node, and the dependencies between classes are the directed edges. Some researchers [10–12] have verified that network metrics are better than traditional hand-crafted metrics in defect prediction. In recent years, as an application of deep learning in network analysis, network representation learning of a graph structure has attracted great attention [13]. The main idea of network embedding is to encode the nodes by learning the graph structure or the adjacency relationship between the nodes and map all nodes into the same dimensional vectors. In other words, network embedding aims to find a mapping function to transform each node into a low-dimensional representation. Recently, Qu et al. [14] attempted to learn the representation of nodes automatically from the CDN using network embedding and achieved encouraging results. With the development of neural network models, many high-performing Graph Neural Networks (GNNs) have emerged [15], which combine node adjacency information and node attributes to capture structural information well.

Inspired by this research trend, this paper proposes a method, CGCN, which combines the advantages of CNN and GCN to extract semantic and structural information for more accurate SDP. Specifically, the source codes are first parsed into ASTs and a software network. CNN is then used to capture the semantic information in the ASTs, while GCN is used to capture the structural information in the software network. Then, we combine the two types of learned features using different weights to obtain the joint features. Finally, the joint features are concatenated with traditional hand-crafted features to train a predictor. For within-version prediction, we adopt Synthetic Minority Oversampling Technique (SMOTE) to alleviate the data imbalance problem and Random Forest (RF) as the classifier. For cross-version and cross-project predictions, we adopt Random Under-Sampling (RUS) as the re-sampling technique and Multi-Layer Perceptron (MLP) as the classifier. The proposed method is evaluated on seven open-source Java projects. The experimental results demonstrate that the proposed CGCN outperforms the state-of-the-art methods in most cases. This paper extends our preliminary study published in ISSRE 2021 [16]. In particular, the extensions include the following aspects.

(1) The novel defect prediction method CGCN, combined with CNN and GCN, is an extended version of the supervised method (i.e., GCN2defect) proposed in our preliminary work [16]. The major difference between CGCN and GCN2defect is that in GCN2defect, only the structural information of the software network is considered, while in CGCN, the semantic information from the ASTs of the source codes is further considered. The experiment results show that CGCN performs much better than GCN2defect. We open-source our dataset and method for interested readers to replicate our research (https://github.com/Emily-zcy/CGCN).

(2) The influence differences between semantic and structural information in CGCN are explored by giving different weights to make them contribute differently to different tasks and improve the model's generalization ability.

(3) The experimental part is strengthened by evaluating the efficiency of CGCN on seven Java open-source projects (each project selects three versions, with a total of 21 datasets) and three tasks (within-version defect prediction, cross-version defect prediction, and cross-project defect prediction). The results demonstrate that the CGCN variant using weight parameters outperforms the state-of-the-art methods. In contrast, the CGCN variant without weight parameters outperforms the baseline models only in the within-version defect prediction.

(4) The performance of CGCN with different combinations of semantic and structural information, different re-sampling methods, and different underlying classifiers, is further discussed.

The remainder of this paper is organized as follows. The related work is introduced in Section 2. Some preliminary concepts directly related to the proposed method are reviewed in Section 3. The proposed method is detailed in Section 4. The experimental setup and results analysis are presented in Section 5 and Section 6. We discuss our work and shortcomings in Section 7. Finally, the conclusion is drawn in Section 8.

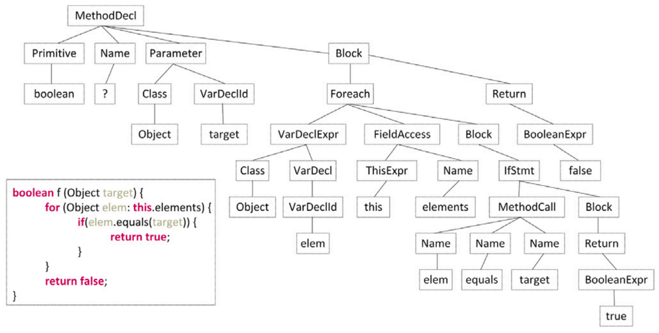
2. Related work

SDP is one of the highly active branches of software engineering [7,14,17–22]. Based on previous studies, the commonly used features in SDP can be roughly divided into three categories, i.e., traditional static hand-crafted features, internal semantic information of source code, and external structural information of software network.

Most conventional SDP studies focus on hand-crafted code metrics, which are fed into machine learning classifiers to predict defects [2,9,23–29]. Recently, tree-based neural network algorithms have attracted much attention in capturing semantic information from ASTs [5–9]. For example, Wang et al. [6] leveraged Deep Belief Network (DBN) to automatically learn semantic information of ASTs from source code. Li et al. [5] used a CNN to extract the semantic information of ASTs and combined the learned features with traditional hand-crafted features to improve the prediction performance. Dam et al. [7] and Zhang et al. [9] used Long Short Term Memory (LSTM) and bidirectional Gated Recurrent Unit (GRU) to fully learn the semantic and syntactic information of long-term dependencies of source code, respectively. Their satisfactory results indicated that tree-based algorithms produced better predictions than conventional machine learning methods. However, ASTs cannot describe some control information, such as control flow and data flow dependencies. Therefore, Phan et al. [30] converted the program into Control Flow Graph (CFG) and extracted the control information from CFG.

The features captured from ASTs and CFGs are more internal information of code files since each file is learned independently. However, the dependencies between code files as external information are also necessary, reflecting the overall structure of software. It is not difficult to imagine that if a class is defective, the classes that depend on it are likely to become defective. Even distant classes or entire modules may eventually be affected by cascading effects. Based on this, Qu et al. [14] introduced the software networks in SDP and adopted network embedding technique (i.e., node2vec) to learn the external structural information of the network automatically. Xu et al. [31] extracted the call dependency graph of the components for each module and then applied the UCINET tool to calculate the structural information of the network.

In fact, real-world networks are often more complex, with not only topological information but also attribute information of nodes and edges. However, conventional network embedding models, such as deepwalk [32], line [33], and node2vec [34], are unsupervised models and cannot utilize node and edge attribute information. With the development of GNNs such as GraphSage [35], GCN [36] and Graph Attention Network (GAT) [37], graph embedding models become able to incorporate node and edge attributes while learning network structure. Therefore, the representations generated by GNNs are more robust than traditional network embedding techniques.



```
boolean f (Object target) {
    for (Object elem: this.elements) {
        if(elem.equals(target)) {
            return true;
        }
    }
    return false;
}
```

Fig. 2. An example of an AST.
Source: Adapted from [7].

Given this, we propose a CGCN method, which can automatically capture the internal semantic information of each file and external structural information between all class files by utilizing CNN and GCN.

3. Preliminaries

In this section, we mainly introduce the construction of AST and CDN, as well as the basic principles of CNN and GCN.

3.1. Abstract syntax tree

Traditional features-based methods in defect prediction often fail to capture code structure and syntactic information. In recent years, many SDP studies have been based on ASTs [5–9]. The nodes on the trees are relatively appropriate granularity to describe a program, which preserves the syntactic and structural information of the source code and ignores unnecessary details such as punctuation and delimiters [38]. We use a public tool javalang[1] to parse the source code into ASTs (cf. Fig. 2). Representative nodes are selected on ASTs to form token sequences and then mapped to the numerical vectors. Finally, each source file is parsed as a representation vector. More details will be introduced in Section 4.

3.2. Convolutional neural network

Convolutional Neural Network (CNN) is a mainstream algorithm in deep learning [39–43], which has a powerful representation capacity for processing grid-like topology. Recently, many studies in SDP used CNN to extract the semantic information from ASTs. Specifically, first of all, all source files are parsed into ASTs. Then, the representative nodes are selected to form a series of token sequences. Finally, each token is encoded as a numerical vector, which is input to CNN. For example, a java file is parsed into a token sequence T of length n:

$$t_{1:n} = t_1 \oplus t_2 \oplus \cdots \oplus t_n, \tag{1}$$

where $t_i \in R^d$ is a d-dimensional real-valued vector of the ith token in the sequence, $t_{1:n}$ refers to the concatenation of token vectors t_1, t_2, $\ldots t_n$, and $\oplus$ is the concatenation operator.

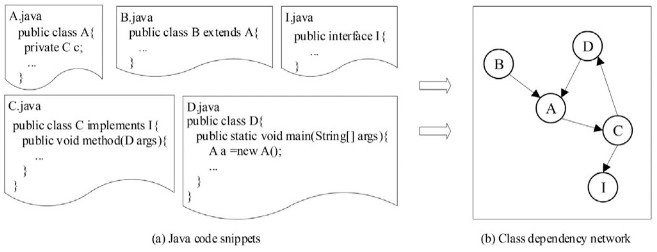
A convolution kernel $w \in R^{h\times d}$ is applied to a window of h tokens to produce a new feature. For instance, the feature c_i is generated from a window of tokens $t_{i:i+h-1}$ by:

$$c_i = \sigma(w \cdot t_{i:i+h-1} + b), \tag{2}$$

where b is a bias term and σ is a non-linear function such as *tanh* or *ReLU*. w is applied to each possible window of tokens in the token sequence $t_{1:h}, t_{2:h+1}, \ldots, t_{n-h+1:n}$ to generate a feature map:

$$\mathbf{c} = [c_1, c_2, \ldots, c_{n-h+1}](\mathbf{c} \in R^{n-h+1}). \tag{3}$$

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



```
A.java
public class A{
private C c;
...
}

B.java
public class B extends A{
...
}

I.java
public interface I{
...
}

C.java
public class C implements I{
public void method(D args){
...
}
}

D.java
public class D{
public static void main(String[] args){
A a =new A();
...
}
}
```

(a) Java code snippets (b) Class dependency network

Fig. 3. Example of the CDN.

We then apply a max-pooling operation over the feature map to take the maximum value $\hat{c} = max\{\mathbf{c}\}$ as the final feature. These learned features focus on the internal semantic information of the code file, i.e., each file is learned independently, so we also call the semantic information extracted by CNN internal features, i.e., $X_{internal}$.

3.3. Software network

The software network, also called Class Dependency Network (CDN), is a directed network based on the dependency relationship between class files within the software. Currently, the *Dependencyfinder API*[2] can be used to parse the compiled source code files and obtain various dependencies between the classes in a project. Fig. 3(a) shows five Java file code fragments, and Fig. 3(b) shows the corresponding CDN. Each node in the network represents a class, and the edge represents the dependency relationship between two class nodes, e.g., class B is a subclass of class A. According to the inheritance relationship between the two classes, an edge exists from B to A ($B \rightarrow A$). Accordingly, $C \rightarrow I$ represents the interface realization relationship, $C \rightarrow D$ denotes the parameter type dependency relationship, and $A \rightarrow C$ and $D \rightarrow A$ represent the aggregation relationship.

3.4. Graph convolutional network

With the application of software networks in SDP, CNN-based models can no longer handle graph data. Hence, we follow Kipf et al.'s study [36] to introduce GCN in SDP to learn structural information from CDN. A given network G is divided into topological information and initial node attribute information, represented by adjacency matrix A and node representation X, respectively. Note that A is globally shared and fixed, while X is updated in each graph convolution layer. GCN uses the Fourier transform principle to perform the convolution operation on a graph, which captures the external interaction information between nodes by the first-order domain:

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}\Theta^{(l)}), \tag{4}$$

where $\tilde{A} = A + I_N$ is the adjacency matrix of the CDN with added self-connections, I_N is the identity matrix, $\tilde{D}$ is the degree matrix of $\tilde{A}$, $\tilde{D} = \sum_j \tilde{A}_{ij}$, $H^{(l)}$ is the output node representations of the lth graph convolution layer, and $\sigma(\cdot)$ is the activation function. The node representations learned by GCN focus on the external interaction information between classes in the CDN, so we call the structural information extracted by GCN external features, i.e., $X_{external}$.

4. Methodology

4.1. Main framework

The workflow of our proposed CGCN (cf. Fig. 4) consists of three parts: (1) parsing source code into ASTs and selecting representative

[2] https://sourceforge.net/projects/depfind/

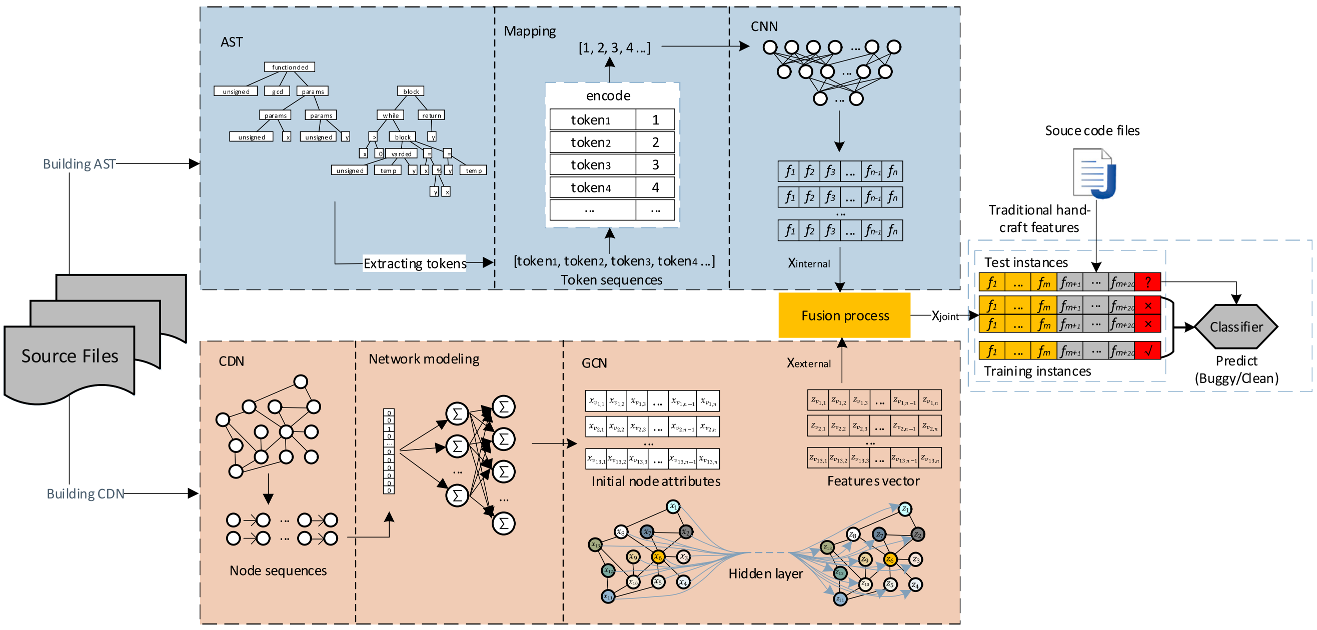


Fig. 4. The overall workflow of our proposed CGCN.

Table 1
The selected AST nodes.

Node type	AST nodes
$Node_1$	MethodInvocation, SuperMethodInvocation, MemberReference, SuperMemberReference
$Node_2$	PackageDeclaration, InterfaceDeclaration, ClassDeclaration, MethodDeclaration, ConstructorDeclaration, VariableDeclarator, CatchClauseParameter, FormalParameter, TryResource, ReferenceType, BasicType
$Node_3$	IfStatement, WhileStatement, DoStatement, ForStatement, AssertStatement, BreakStatement, ContinueStatement, ReturnStatement, ThrowStatement, SynchronizedStatement, TryStatement, SwitchStatement, CatchClause BlockStatement, StatementExpression, ForControl, SwitchStatementCase, EnhancedForControl

tokens and employing CNN to generate internal features; (2) building a CDN based on the dependencies between class files and employing GCN to generate external features; (3) combining the internal and external features for SDP.

4.2. Generating internal features

We perform the following steps to generate internal features: (1) parsing source code into ASTs; (2) selecting representative nodes on ASTs to form token sequences and mapping each token to a numerical vector; (3) leveraging CNN to generate internal features.

Parsing source code. We use javalang to parse source code into ASTs. Consistent with Wang et al.'s [6] and Li et al.'s [5], we only select three types of nodes on ASTs: (1) $Node_1$: nodes of method invocations and class instance creations, (2) $Node_2$: declaration nodes, and (3) $Node_3$: control-flow nodes. Table 1 lists the selected AST nodes. Since method names and variables are typical for specific projects, only using those tokens may not be sufficient for cross-project settings. In this case, we use the AST node types for cross-project prediction. For example, a node named *getXXX* is used in within-project, while its type *MethodInvocation* is used in cross-project.

Encoding Tokens. CNN requires inputs as numerical vectors, and the length of the input vectors must be the same. However, the extracted token sequences of AST may differ in their sizes. To solve this problem, we map each token to a unique integer identifier and append zero to the integer vector to make all the lengths consistent and equal to the length of the longest vector.

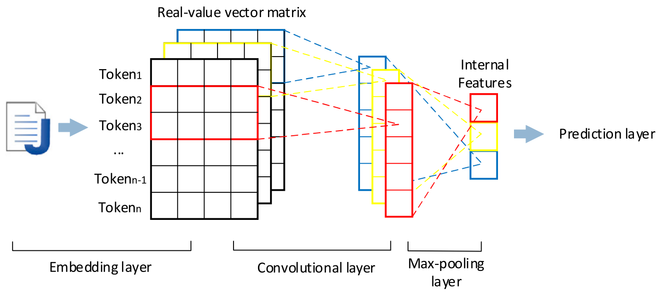


Fig. 5. The generation process of the internal features.

Building CNN. Following Li et al.'s study [5], we use the same standard architecture of CNN, as shown in Fig. 5. It includes an embedding layer, a convolutional layer, a max-pooling layer, and a prediction layer. Given a project P containing n Java source code files, i.e., $P = \{f_1, f_2, \ldots, f_n\}$. After the above processing, the token sequences $\{T_1, T_2, \ldots, T_n\}$ of these Java files are extracted, where T_i is the extracted token sequence from f_i. After the embedding layer, f_i is converted to a real-valued vector matrix $f_i \in R^{l \times d}, i \in [1, n]$, where l is the length of the longest token sequence. Then, after the convolution and max-pooling layers (cf. Eq. (1) ∼ Eq. (3)), $X_{internal} \in R^{n \times D}$ is generated to represent the internal information of the source code.

4.3. Generating external features

We also perform three steps to generate external features: (1) building a Class Dependency Network (CDN); (2) generating initial node attributes of CDN; (3) employing GCN to generate external features.

Building CDN. Following He et al.'s work [44,45], we consider the following three dependency cases when building CDN.

a) Inherit: if class v_{c1} inherits to class v_{c2} or implements the interface v_{c2}, a directed edge $e_{12} = \langle v_{c1}, v_{c2} \rangle$ exists;

Table 2
Description of the software metrics.

Metrics	Description
WMC	Weighted Methods per Class
DIT	Depth of Inheritance Tree
NOC	Number of Children
CBO	Coupling Between Object classes
RFC	Response for a Class
LCOM	Lack of Cohesion in Methods
LCOM3	Lack of Cohesion in Methods, different from LCOM
NPM	Number of Public Methods
DAM	Data Access Metric
MOA	Measure of Aggregation
MFA	Measure of Functional Abstraction
CAM	Cohesion Among Methods of Class
IC	Inheritance Coupling
CBM	Coupling Between Methods
AMC	Average Method Complexity
CA	Afferent Couplings
CE	Efferent Couplings
MAX(CC)	Maximum value of CC methods of the investigated class
AVG(CC)	Arithmetic mean of the CC value in the investigated class
LOC	Lines of Code

b) Aggregation: if class v_{c1} contains the attributes of class v_{c2}, a directed edge $e_{12} = \langle v_{c1}, v_{c2} \rangle$ exists;

c) Parameter: if the method of class v_{c1} calls the method of class v_{c2}, a directed edge $e_{12} = \langle v_{c1}, v_{c2} \rangle$ exists.

In fact, CDN is a directed network. However, the GCN proposed by Kipf et al. [36] is limited to undirected graphs. Therefore, we ignore the direction of the edge in CDN in our study.

Generating initial node attributes. Since the original CDN has no node attributes, we should generate initial node attributes before training GCN. Based on our previous work [16], we use traditional 20 static code metrics (as shown in Table 2) and network embedding metrics generated by the node2vec [14] algorithm as the initial node attributes.

Building GCN. We follow Kipf et al.'s study [36] to adopt a two-layer convolution. GCN takes the simple form as:

$$f(X, A) = softmax(\hat{A}ReLU(\hat{A}XW^{(0)})W^{(1)}), \tag{5}$$

where X is the initial node attributes, $\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ is the normalized adjacency matrix of the Laplace matrix, $W^{(0)} \in R^{C\times H}$ and $W^{(1)} \in R^{H\times D}$ are the weight matrices of the two layers, C is the dimension of the input node representation, H is the dimension of the hidden node representation, and $ReLU(\cdot)$ is a nonlinear activation function.

In addition to just predicting the node labels, it can be helpful to get a more detailed picture of what information the model has learned about the nodes and their neighborhoods. The output of the $softmax$ layer $f(X, A) \in R^{n\times 2}$ is the label probability of a node, which has little useful information about the external features. In this case, we use the node representation of the last graph convolution layer of the GCN model as external features instead of the prediction layer. Therefore, the node representation of the last graph convolution layer is used as $X_{external} \in R^{n\times H}$.

4.4. Generating the final representation and predicting the defects

After the above processes, the internal and external features are generated for all the source files. Considering that the contributions of the internal and external features are not the same in different prediction tasks, we take a parameter α to assign different weights to different features, which can efficiently balance the internal and external information:

$$X_{joint} = combine((1-\alpha) * X_{internal}, \alpha * X_{external}), \tag{6}$$

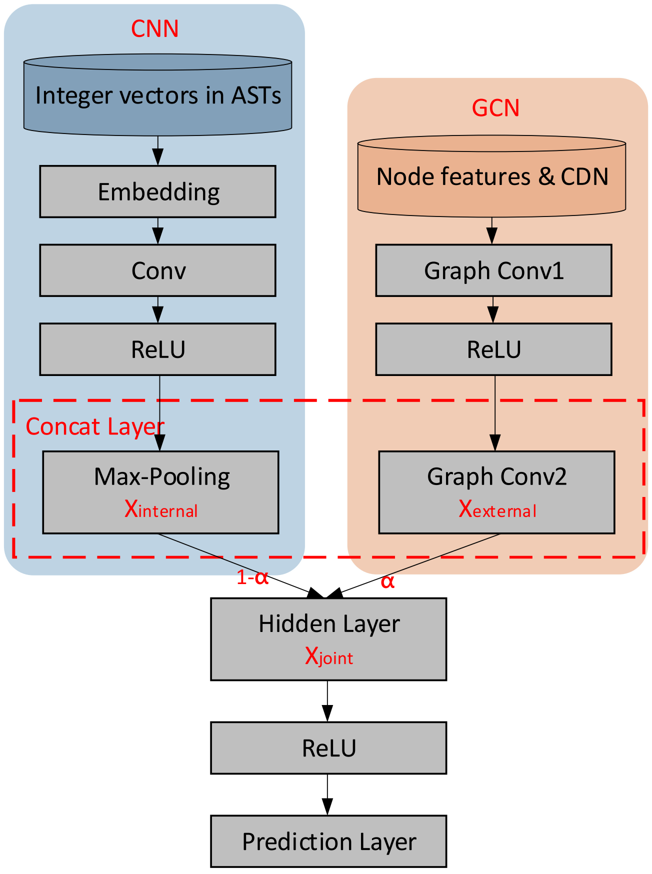


Fig. 6. The fusion process of CNN and GCN.

where X_{joint} is the joint features, $combine(\cdot)$ is a function that joins the two types of features. The parameter α is selected within $[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$.

Fig. 6 illustrates the fusion process of CNN and GCN. The max-pooling layer of CNN and the last layer of the graph convolution of GCN are merged. Note that CNN and GCN are jointly trained. The prediction result of CGCN $Z \in R^{n\times 2}$ is obtained after prediction layer. The loss function is given by:

$$Loss(Z, Y) = -\sum_{i} Y_i log Z_i, \tag{7}$$

where the output Z of the $softmax$ function in the prediction layer represents the probability of different predicted outcomes, Y is the real label, and i donates each node in CDN.

After that, we combine the learned features X_{joint} with traditional hand-crafted features to train a classifier. Since software defect datasets are often imbalanced, we employ data re-sampling techniques to make the training set balanced. We conduct experiments using three re-sampling techniques: (1) SMOTE [46], which calculates the distance between a randomly selected minority class instance and one of its neighbors to generate synthetic instances. (2) RUS [47], which randomly removes some of the majority instances to balance the data. (3) SMOTETomek [48], which is a hybrid technique of over-sampling technique (SMOTE) and under-sampling technique (Tomek Links).

It is essential to mention that the data distribution changes after re-sampling, which affects the original CDN structure. In this case, we do not use the re-sampling technique during CGCN training. In short, the models in this paper are divided into two parts: pre-training and downstream tasks. The pre-training aims to learn node representations through different models. The downstream task aims to sample the pre-trained node representations and train a classifier to predict whether

Table 3
Subject software systems.

Projects	Description	Version	# Nodes	# Edges	% Defective
ant	Java based build tool	1.4	175	2862	22.86%
		1.6	343	1798	26.82%
		1.7	732	3687	22.40%
camel	Enterprise integration framework	1.2	578	2431	36.68%
		1.4	805	3662	18.01%
		1.6	886	4227	21.22%
jEdit	Text editor designed for programmers	3.2	260	1359	34.62%
		4.0	293	1546	25.60%
		4.1	299	1638	26.42%
lucene	Text search engine library	2.0	181	755	50.28%
		2.2	229	948	62.45%
		2.4	324	1481	62.35%
poi	Java library to access Microsoft format files	1.5	228	949	60.53%
		2.5	371	1693	65.77%
		3.0	427	2194	65.34%
velocity	Java based template engine	1.4	192	1016	76.04%
		1.5	212	1127	66.51%
		1.6	227	1219	34.36%
xalan	A library for transforming XML files	2.4	676	4258	16.12%
		2.5	725	4552	50.76%
		2.6	810	4806	46.17%

a new module is defective or not. Therefore, we do not directly use the prediction results of the prediction layer in Fig. 6, since these predictions tend to be biased towards the majority class. Note that the learned features of models in pre-training are not trained jointly with the subsequent downstream prediction tasks.

5. Experiment setup

5.1. Datasets

We choose seven defective projects from the PROMISE[3] repository. Table 3 lists the information about the projects, where #Nodes, #Edges, and %Defective represent the number of class files in the CDN of the software version, the number of dependencies between the class files, and the corresponding buggy rate, respectively.

5.2. Baseline models

To validate the effectiveness of the proposed method, we select the seven methods as our baseline models.

Traditional [29]**:** The traditional methods use 20 traditional hand-crafted code metrics (cf. Table 2) as input to train a classifier, such as Random Forest, Decision Tree, etc.

CNN [5]**:** CNN is an AST-based method that accepts ASTs as input to capture semantic information of source code.

BiLSTM [7]**:** Bidirectional LSTM is also an AST-based method that accepts ASTs as input to capture semantic information of source code.

Deepwalk [32]**:** Deepwalk is an unsupervised algorithm that uses truncated random walks to extract structural information from CDN.

Node2defect [14]**:** Node2defect is an improved method of deepwalk that combines breadth-first traversal (BFS) and depth-first traversal (DFS). It uses node2vec to extract structural information from CDN.

GCN2defect [16]**:** GCN2defect employs GCN to learn node attribute features and network structure information of source code.

GAT[4] [37]: GAT is an improved method of GCN that considers the attention mechanism to learn node attribute features and network structure information of source code.

[3] http://openscience.us/repo/

[4] The GCN and GAT implementations are from https://github.com/stellargraph/stellargraph.

Table 4
Experimental environment.

Parameter	Values
CPU	Intel® Xeon(R) Gold 5218
GPU	NVIDIA GeForce GTX5000-16G
development language	Python-3. 6
Deep learning framework	TensorFlow-2.2.0
development tool	Pycharm-2020.1.3

Deepwalk and node2vec are unsupervised models that do not require labels during pre-training. However, CNN, BiLSTM, GCN, and GAT are supervised models with end-to-end training, and the $softmax$ function is used in the prediction layer. For a fair comparison, the prediction results of the $softmax$ are not directly used, but the node representations of the last hidden layer of the model are saved as the results of pre-training.

5.3. Settings

Within-version prediction: In this setting, training and testing data are extracted from the same version of the same project. Five-fold cross-validation is used in within-version prediction. Specifically, in each run, the dataset is randomly split into five folds, 80% of the instances are used for training a prediction model, and 20% are used to evaluate the model. We repeat this process 25 times and compute the average performance to reduce the bias caused by randomly dividing the data.

Cross-version prediction: In this second experimental setting, the lower version is used to train a prediction model, while the higher version is used to evaluate the model [49]. For example, in the first run, all the source files in ant-1.4 are used to train, and all the source files in ant-1.6 are used to evaluate. In the second run, the model is trained using ant-1.6 and tested using ant-1.7. Then the averaged performance is computed.

Cross-Project prediction: In this third experimental setting, the first version of each project is used to train a prediction model, and the remaining six projects are used for the test [50]. In other words, the model is tested six times. Then the averaged performance is computed.

Environmental environment: The experimental environment is presented in Table 4.

Experimental parameters: The hyper-parameters used in this paper are shown in Table 5. In this study, after pre-training, all models

Table 5
Hyper-parameters settings.

Parameter	Values
embedding dimension after pre-training	32
epoch	200
optimizer	Adam
learning rate	0.001
dropout	0.1
Imbalanced processing threshold δ	0.4

obtain an $n \times 32$ matrix, where n is the number of all the source files in a project, and 32 is the dimension of embeddings. All the supervised models used in this study are trained for 200 epochs with an early stop strategy and employ the Adam optimizer with a 0.001 learning rate. In our implementation, the batch size in CNN and BiLSTM is set as 32. GCN and GAT are full-batch trained. The parameter p and q in node2vec is set as 0.25 and 2, respectively. The other parameters are consistent with their studies. To more accurately predict defects, we concatenate the learned embeddings after pre-training with traditional hand-crafted metrics, and feed the concatenated vector into the downstream task. The threshold value of imbalance in the downstream task is set as 0.4. In other words, if the buggy instances are lower than 40% in the training set, the data should be balanced, and the test set preserves its original defective ratio. Finally, the training data is fed into the final classifier for training.

5.4. Evaluation metrics

We use the widely adopted metrics to evaluate the effectiveness of various methods, i.e., Accuracy and F1-measure:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}, \tag{8}$$

$$precision = \frac{TP}{TP+FP}, \tag{9}$$

$$recall = \frac{TP}{TP+FN}, \tag{10}$$

$$F1 = \frac{2 * precision * recall}{precision + recall}, \tag{11}$$

where TP, TN, FP, and FN denote the number of true positives, true negatives, false positives, and false negatives, respectively.

The Area Under the ROC Curve (AUC) is also widely used for evaluation in SDP, especially when the dataset is imbalanced. The value of AUC is ranged from 0 to 1, and 0.5 represents random prediction. The larger the values of the three evaluation metrics, the better the prediction performance.

5.5. Research questions

This paper aims to answer the following three Research Questions (RQs):

(1) RQ1: Which task is the proposed CGCN method suitable for (within-version, cross-version, or cross-project)?

(2) RQ2: Does the feature combination weight strategy have a significant impact on the proposed CGCN method?

(3) RQ3: Does the proposed CGCN method perform better than the baseline models?

6. Experimental results

6.1. RQ1: Which task is the proposed CGCN method suitable for (within-version, cross-version or cross-project)?

Based on the previous studies [6,7], we also perform the three tasks, i.e., within-version prediction, cross-version prediction, and cross-project prediction. We adopt $add(\cdot)$ as the feature combination method for the three tasks to combine $X_{internal}$ and $X_{external}$, and employ SMOTE as the re-sampling method for within-version prediction and RUS for cross-version and cross-project predictions. In addition, we utilize RF as the classifier for within-version predictions and MLP for cross-version and cross-project predictions. These settings are used for all experiments in this section. A detailed discussion is presented in Section 7. Fig. 7 and Tables 6–8 present the results of CGCN in the three prediction tasks.

Table 6
Performance of CGCN in within-version prediction.

project	version	F1	AUC	Accuracy
ant	1.4	0.862	0.939	0.946
	1.6	0.875	0.955	0.932
	1.7	0.916	0.974	0.962
camel	1.2	0.875	0.938	0.910
	1.4	0.872	0.963	0.953
	1.6	0.830	0.933	0.928
jedit	3.2	0.941	0.983	0.961
	4.0	0.957	0.990	0.979
	4.1	0.922	0.981	0.962
lucene	2.0	0.929	0.969	0.931
	2.2	0.919	0.934	0.900
	2.4	0.944	0.968	0.930
poi	1.5	0.919	0.958	0.903
	2.5	0.964	0.987	0.953
	3.0	0.943	0.961	0.926
velocity	1.4	0.990	0.990	0.984
	1.5	0.969	0.975	0.959
	1.6	0.912	0.991	0.942
xalan	2.4	0.806	0.919	0.934
	2.5	0.941	0.966	0.940
	2.6	0.935	0.975	0.940
average		0.915	0.964	0.942

(1) For within-version prediction, the F1 value is up to 0.8 on all the 21 projects, and the best result is 0.99. Considering AUC and Accuracy, the values are larger than 0.9, and the best results are 0.99 and 0.984, respectively.

(2) For cross-version prediction, the F1 value is up to 0.35 for almost all the projects. The best and worst F1 values are 0.912 and 0.134, respectively. The AUC and Accuracy values are larger than 0.5 for almost all the projects, and the highest values are 0.870 and 0.876, respectively.

(3) For cross-project prediction, the best and worst F1 values are 0.78 and 0.119, respectively. The best and worst AUC values are 0.785 and 0.206, respectively. The best and worst Accuracy values are 0.756 and 0.229, respectively.

Based on the above experimental results, it can be observed that CGCN is more suitable for within-version prediction. The data in different versions or projects have a distribution difference, so the prediction model trained by the source project may perform poorly for the target project. Although the cross-version prediction also belongs to Within-Project Defect Prediction (WPDP), the prediction performance is far inferior to that of within-version prediction. This may be due to software changes, such as the number of source files, dependencies, and code style.

6.2. RQ2: Does the feature combination weight strategy have a significant impact on the proposed CGCN method?

In RQ1, we directly combine the internal and external features, both of which contribute equally, i.e., $X_{joint} = combine(X_{internal}, X_{external})$. Considering that the contribution of internal and external features may not be the same in different tasks, we use the parameter α when combining internal and external features, i.e., $X_{joint} = combine((1-\alpha) *$

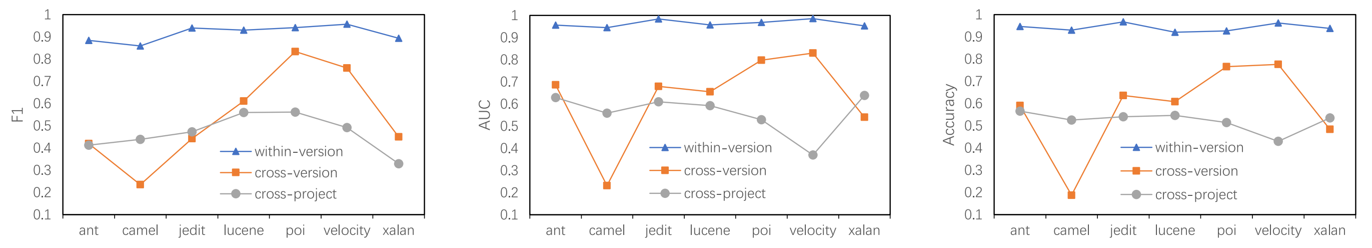


Fig. 7. Performance of CGCN in the three prediction tasks.

Table 7
Performance of CGCN in cross-version prediction.

project	version	F1	AUC	Accuracy
ant	1.4⇒1.6	0.357	0.555	0.633
	1.6⇒1.7	0.481	0.817	0.549
camel	1.2⇒1.4	0.134	0.213	0.166
	1.4⇒1.6	0.336	0.251	0.209
jedit	3.2⇒4.0	0.485	0.741	0.594
	4.0⇒4.1	0.400	0.619	0.679
lucene	2.0⇒2.2	0.790	0.834	0.751
	2.2⇒2.4	0.433	0.477	0.466
poi	1.5⇒2.5	0.912	0.876	0.876
	2.5⇒3.0	0.755	0.720	0.656
velocity	1.4⇒1.5	0.839	0.790	0.755
	1.5⇒1.6	0.681	0.870	0.797
xalan	2.4⇒2.5	0.368	0.549	0.537
	2.5⇒2.6	0.532	0.532	0.431
average		0.536	0.632	0.578

$X_{internal}, \alpha * X_{external}$). The variant in this experiment is referred to as CGCN-α.

Fig. 8 shows the comparison results of the two variants in the three prediction tasks:

(1) For within-version prediction, the F1, AUC, and Accuracy values of CGCN are up to 0.8, 0.91, and 0.9, respectively, while the F1, AUC, and Accuracy values of CGCN-α are up to 0.82, 0.93 and 0.91, respectively. CGCN-α is a bit better than CGCN, and the F1, AUC, and Accuracy values are improved by 1.1%, 0.1%, and 0.8% on average.

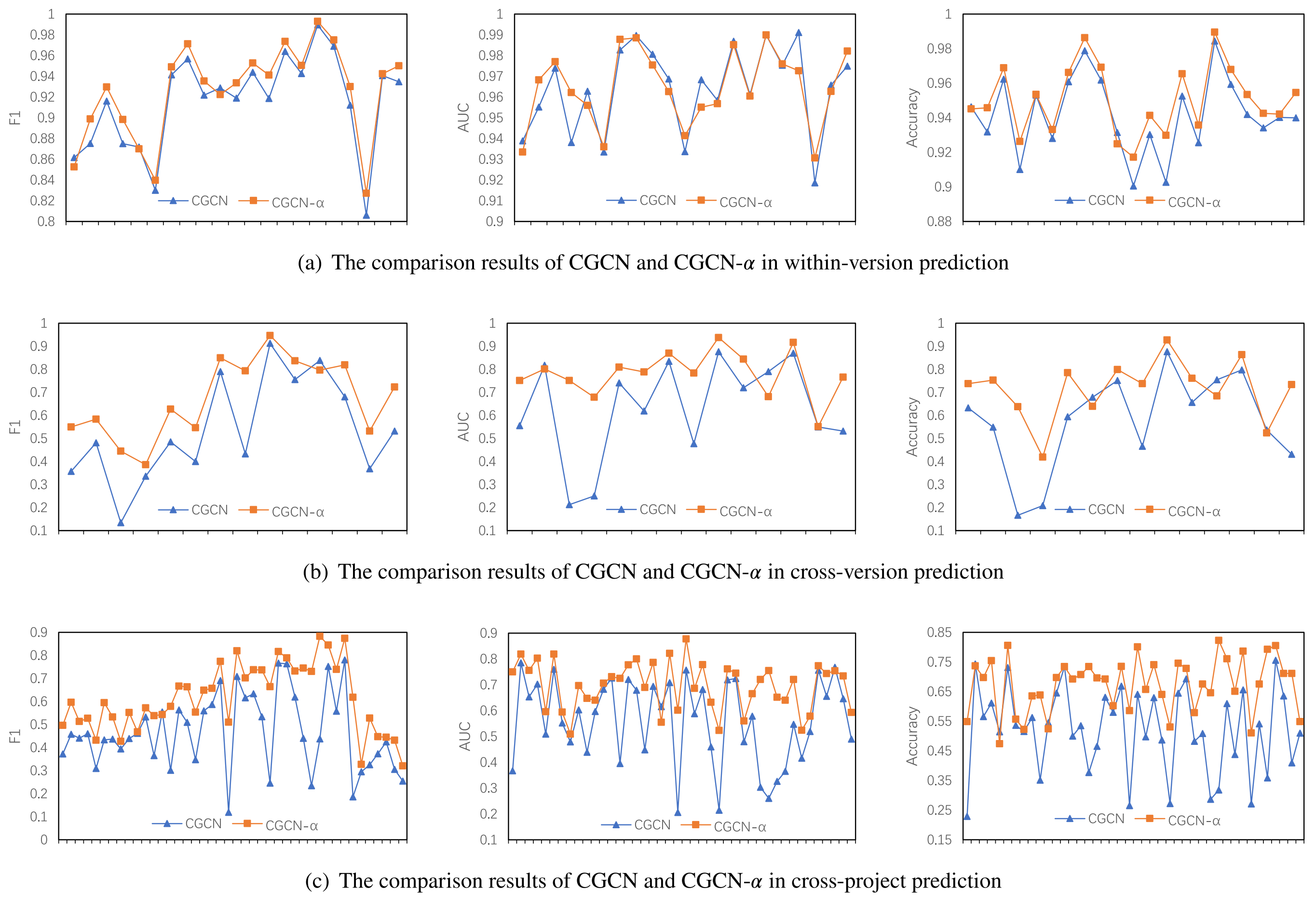
(2) For cross-version prediction, CGCN's lowest, highest, and average F1 values are 0.134, 0.912, and 0.536; lowest, highest, and average AUC values are 0.213, 0.876, and 0.632; lowest, highest, and average Accuracy values are 0.166, 0.876, and 0.578, respectively. CGCN-α's lowest, highest, and average F1 values are 0.386, 0.947, and 0.674; lowest, highest, and average AUC values are 0.551, 0.938, and 0.781; lowest, highest, and average Accuracy values are 0.419, 0.927, and 0.714, respectively. CGCN-α significantly outperforms CGCN, with the values of F1, AUC, and Accuracy improved by 13.8%, 14.9%, and 13.6% on average.

(3) For cross-project prediction, CGCN's lowest, highest, and average F1 values are 0.119, 0.78, and 0.466; lowest, highest, and average AUC values are 0.206, 0.785, and 0.561; lowest, highest, and average Accuracy values are 0.229, 0.756, and 0.523, respectively. CGCN-α's lowest, highest, and average F1 values are 0.321, 0.883, and 0.614; lowest, highest, and average AUC values are 0.509, 0.878, and 0.696; lowest, highest, and average Accuracy values are 0.474, 0.823, and 0.674, respectively. CGCN-α significantly outperforms CGCN, and the F1, AUC, and Accuracy values are improved by 14.8%, 13.5%, and 15.1% on average.

The results demonstrate that CGCN-α outperforms CGCN in all three prediction tasks. In within-version prediction, CGCN-α performs slightly better than CGCN in most cases, indicating no significant difference between the contributions of internal and external features. In cross-version and cross-project predictions, the performance of CGCN-α is significantly better and much more stable than CGCN, indicating that the two features contribute differently to the projects. More specifically, if $\alpha > 0.5$, external features contribute more than internal features. Otherwise, internal features contribute more than external features. However, from Fig. 9, we can observe that the numbers of $\alpha > 0.5$ and $\alpha < 0.5$ are comparable in the three prediction tasks, indicating that the contributions of the two features differ for different projects, but there is no obvious regularity in different prediction tasks.

Table 8
Performance of CGCN in cross-project prediction.

Train project	Test project	F1	AUC	Accuracy
camel	ant	0.372	0.367	0.229
jedit		0.458	0.785	0.743
lucene		0.441	0.653	0.566
poi		0.460	0.703	0.611
velocity		0.309	0.509	0.514
xalan		0.434	0.760	0.731
ant	camel	0.437	0.552	0.536
jedit		0.394	0.479	0.516
lucene		0.439	0.603	0.562
poi		0.462	0.438	0.351
velocity		0.533	0.597	0.545
xalan		0.365	0.683	0.645
ant	jedit	0.555	0.726	0.735
camel		0.301	0.395	0.500
lucene		0.563	0.721	0.535
poi		0.509	0.679	0.377
velocity		0.347	0.448	0.465
xalan		0.560	0.694	0.631
ant	lucene	0.587	0.615	0.580
camel		0.691	0.708	0.669
jedit		0.119	0.206	0.265
poi		0.709	0.757	0.641
velocity		0.616	0.588	0.497
xalan		0.634	0.682	0.630
ant	poi	0.534	0.460	0.487
camel		0.245	0.214	0.272
jedit		0.767	0.719	0.645
lucene		0.764	0.724	0.693
velocity		0.619	0.480	0.482
xalan		0.440	0.578	0.509
ant	velocity	0.235	0.304	0.286
camel		0.438	0.261	0.318
jedit		0.752	0.326	0.609
lucene		0.557	0.365	0.438
poi		0.780	0.547	0.656
xalan		0.186	0.416	0.271
ant	xalan	0.295	0.519	0.541
camel		0.325	0.756	0.359
jedit		0.373	0.655	0.756
lucene		0.424	0.768	0.635
poi		0.305	0.646	0.410
velocity		0.254	0.490	0.510
average		0.466	0.561	0.523



(a) The comparison results of CGCN and CGCN-α in within-version prediction

(b) The comparison results of CGCN and CGCN-α in cross-version prediction

(c) The comparison results of CGCN and CGCN-α in cross-project prediction

Fig. 8. Comparison results of CGCN and CGCN-α in the three prediction tasks.

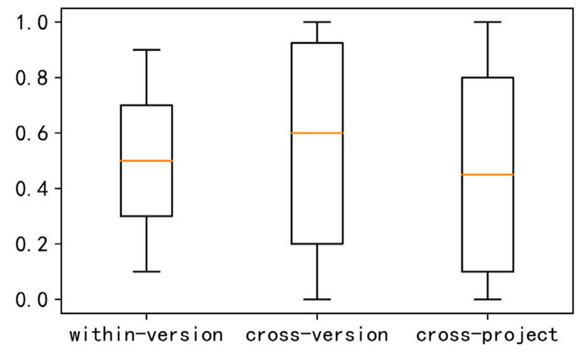


Fig. 9. Distribution of the weight parameter α in the three prediction tasks.

6.3. RQ3: Does the proposed CGCN method perform better than the baseline models?

For a fair comparison, the re-sampling methods and classifiers in the baseline models are kept consistent with our proposed approach. In other words, all baseline models use SMOTE and RF in within-version prediction, while RUS and MLP are used in cross-version and cross-project predictions. Fig. 10 and Tables 9–17 present the experimental results of the two proposed variants with baseline models.

(1) Tables 9–11 and Fig. 10(a) report the results of comparing our method with baseline models in within-version prediction. The two proposed variants outperform the baseline models in all 21 projects. CGCN-α performs best, and the F1, AUC, and Accuracy values are improved by 13.3% ∼ 28.7%, 6.6% ∼ 17.4%, and 9.8% ∼ 18.9% on average.

(2) Tables 12–14 and Fig. 10(b) report the results of comparing our method with baseline models in cross-version prediction. CGCN-α outperforms the baseline models in most cases, and the F1, AUC, and Accuracy values are improved by 7.2% ∼ 15.8%, 8% ∼ 14.9%, and 7.8% ∼ 12.4% on average. But CGCN performs poorly in most cases.

(3) Tables 15–17 and Fig. 10(c) report the results of comparing our method with baseline models in cross-project prediction. Similar to the results in cross-version prediction, CGCN-α outperforms the baseline models in most cases, the values of F1, AUC, and Accuracy improved by 13.2% ∼ 16.9%, 10.7% ∼ 14.2%, and 13.6% ∼ 18.7% on average. But CGCN performs poorly in most cases.

(4) From the experimental results in cross-version and cross-project predictions, the AUC values are smaller than 0.5 in some cases, which indicates the GCCN method is worse than the random guess. The potential reason is that the significant distribution difference between training and testing data in cross-version and cross-project predictions leads to the poor prediction of GCCN.

The results demonstrate that CGCN-α performs best compared to all baseline models. CGCN performs well in within-version prediction but poorly in cross-version and cross-project predictions. The results generally validate our intuition that considering the weighting strategies when combining different types of features can improve the prediction performance and stability of the model.

Table 9
F1 values of different methods in within-version prediction.

project	version	traditional	CNN	BiLSTM	deepwalk	node2vec	GCN	GAT	CGCN	CGCN-α
ant	1.4	0.366	0.697	0.396	0.524	0.529	0.490	0.509	**0.862**	0.853
	1.6	0.620	0.835	0.629	0.581	0.598	0.653	0.617	0.875	**0.899**
	1.7	0.575	0.823	0.590	0.558	0.558	0.628	0.584	0.916	**0.930**
camel	1.2	0.507	0.697	0.518	0.542	0.580	0.738	0.649	0.875	**0.898**
	1.4	0.315	0.585	0.317	0.443	0.459	0.605	0.554	**0.872**	0.870
	1.6	0.348	0.682	0.311	0.473	0.496	0.614	0.540	0.830	**0.840**
jedit	3.2	0.670	0.832	0.704	0.727	0.735	0.771	0.748	0.941	**0.949**
	4.0	0.546	0.836	0.655	0.620	0.600	0.706	0.636	0.957	**0.971**
	4.1	0.613	0.743	0.633	0.620	0.609	0.686	0.637	0.922	**0.935**
lucene	2.0	0.680	0.696	0.642	0.707	0.647	0.763	0.690	**0.929**	0.922
	2.2	0.719	0.769	0.751	0.756	0.749	0.786	0.765	0.919	**0.934**
	2.4	0.779	0.813	0.758	0.798	0.812	0.830	0.822	0.944	**0.953**
poi	1.5	0.792	0.824	0.814	0.819	0.814	0.878	0.877	0.919	**0.941**
	2.5	0.857	0.948	0.886	0.898	0.894	0.924	0.923	0.964	**0.974**
	3.0	0.855	0.933	0.853	0.868	0.865	0.880	0.869	0.943	**0.950**
velocity	1.4	0.911	0.948	0.943	0.920	0.926	0.951	0.938	0.990	**0.993**
	1.5	0.833	0.887	0.807	0.839	0.848	0.869	0.858	0.969	**0.975**
	1.6	0.614	0.835	0.640	0.582	0.578	0.708	0.651	0.912	**0.930**
xalan	2.4	0.388	0.568	0.366	0.381	0.366	0.456	0.375	0.806	**0.827**
	2.5	0.707	0.850	0.745	0.731	0.750	0.780	0.781	0.941	**0.942**
	2.6	0.727	0.847	0.771	0.730	0.732	0.783	0.765	0.935	**0.950**
average		0.639	0.793	0.654	0.672	0.673	0.738	0.704	0.915	**0.926**

Table 10
AUC values of different methods in within-version prediction.

project	version	traditional	CNN	BiLSTM	deepwalk	node2vec	GCN	GAT	CGCN	CGCN-α
ant	1.4	0.694	0.893	0.749	0.838	0.853	0.828	0.836	**0.939**	0.933
	1.6	0.825	0.947	0.854	0.824	0.816	0.862	0.827	0.955	**0.968**
	1.7	0.830	0.954	0.866	0.846	0.846	0.869	0.860	0.974	**0.977**
camel	1.2	0.673	0.852	0.706	0.756	0.777	0.876	0.833	0.938	**0.962**
	1.4	0.713	0.831	0.701	0.808	0.819	0.879	0.868	**0.963**	0.956
	1.6	0.716	0.890	0.730	0.806	0.819	0.854	0.830	0.933	**0.936**
jedit	3.2	0.854	0.950	0.891	0.911	0.913	0.923	0.913	0.983	**0.988**
	4.0	0.803	0.952	0.872	0.868	0.856	0.897	0.875	**0.990**	0.988
	4.1	0.824	0.925	0.872	0.862	0.863	0.893	0.873	**0.981**	0.975
lucene	2.0	0.755	0.761	0.736	0.796	0.740	0.846	0.798	**0.969**	0.963
	2.2	0.637	0.740	0.652	0.721	0.714	0.794	0.764	0.934	**0.941**
	2.4	0.781	0.837	0.749	0.798	0.815	0.861	0.847	**0.968**	0.955
poi	1.5	0.773	0.861	0.824	0.834	0.821	0.890	0.910	**0.958**	0.957
	2.5	0.910	0.976	0.937	0.946	0.941	0.955	0.954	**0.987**	0.985
	3.0	0.890	0.962	0.892	0.914	0.903	0.917	0.914	**0.961**	0.960
velocity	1.4	0.875	0.942	0.934	0.906	0.915	0.977	0.926	0.990	**0.990**
	1.5	0.827	0.938	0.791	0.860	0.855	0.886	0.871	0.975	**0.976**
	1.6	0.814	0.923	0.843	0.774	0.792	0.881	0.835	**0.991**	0.973
xalan	2.4	0.792	0.874	0.786	0.801	0.800	0.807	0.794	0.919	**0.931**
	2.5	0.780	0.924	0.830	0.829	0.833	0.862	0.856	**0.966**	0.963
	2.6	0.848	0.941	0.884	0.867	0.872	0.903	0.891	0.975	**0.982**
average		0.791	0.899	0.814	0.836	0.836	0.879	0.861	0.964	**0.965**

7. Discussion

In Section 5, three different settings are used for within-version prediction, cross-version prediction, and cross-project prediction. Therefore, to evaluate the impact of different experimental settings on our approach, we test our approach using different feature combination methods, re-sampling methods, and classifiers. Note that, when comparing the impact of a setting, the other settings are kept consistent with the experiments in Section 5. In this section, we use CGCN in all the comparative experiments since tuning out the optimal parameter α in CGCN-α has a high computational load.

In this work, we use the Wilcoxon signed-rank test and effect size (i.e., Cliff's δ) to check whether the performance difference between different settings is significant. The Wilcoxon signed-rank test is applied to judge whether there is a statistically significant difference between data pairs. p-value <0.05 indicates the difference between pairs is statistically significant. Otherwise, there is no significant difference between pairs. Cliff's δ is a non-parametric effect size measure that quantifies the difference between data pairs beyond p-values interpretation. The effect size is interpreted as negligible ($|\delta| < 0.147$), small ($0.147 \le |\delta| < 0.33$), medium ($0.33 \le |\delta| < 0.474$), and large ($|\delta| \ge 0.474$) [51].

7.1. The impact of feature combination on the model

We adopt three ways to combine internal and external features, i.e., $concatenate(\cdot)$, $add(\cdot)$ and $multiply(\cdot)$. The dimension of combined features by $concatenate(\cdot)$ is the sum of internal and external features, while the dimension of combined features by $add(\cdot)$ and $multiply(\cdot)$ is the same as that of internal and external features. For convenience, the dimension of both internal and external features is set as 32. Table 21 presents the statistical significance of the three combinations.

Tables 18–20 present the performance results of the three feature combination methods on CGCN. The highest value of each row is

Table 11
Accuracy values of different methods in within-version prediction.

project	version	traditional	CNN	BiLSTM	deepwalk	node2vec	GCN	GAT	CGCN	CGCN-α
ant	1.4	0.739	0.871	0.770	0.805	0.807	0.767	0.807	**0.946**	0.945
	1.6	0.794	0.910	0.796	0.786	0.791	0.804	0.798	0.932	**0.946**
	1.7	0.818	0.920	0.810	0.814	0.810	0.824	0.825	0.962	**0.969**
camel	1.2	0.667	0.775	0.680	0.704	0.720	0.809	0.753	0.910	**0.926**
	1.4	0.794	0.850	0.797	0.833	0.832	0.849	0.851	0.953	**0.954**
	1.6	0.762	0.865	0.776	0.821	0.817	0.829	0.818	0.928	**0.933**
jedit	3.2	0.788	0.884	0.806	0.816	0.818	0.841	0.829	0.961	**0.966**
	4.0	0.775	0.917	0.838	0.815	0.805	0.848	0.818	0.979	**0.986**
	4.1	0.811	0.872	0.823	0.819	0.813	0.837	0.825	0.962	**0.969**
lucene	2.0	0.694	0.704	0.665	0.726	0.674	0.781	0.715	**0.931**	0.925
	2.2	0.620	0.696	0.667	0.694	0.679	0.733	0.705	0.900	**0.917**
	2.4	0.714	0.765	0.689	0.744	0.757	0.788	0.772	0.930	**0.941**
poi	1.5	0.745	0.789	0.772	0.781	0.765	0.854	0.853	0.903	**0.930**
	2.5	0.812	0.932	0.852	0.869	0.863	0.902	0.900	0.953	**0.965**
	3.0	0.811	0.913	0.810	0.830	0.826	0.844	0.831	0.926	**0.936**
velocity	1.4	0.860	0.921	0.910	0.875	0.883	0.924	0.904	0.984	**0.990**
	1.5	0.772	0.852	0.734	0.783	0.792	0.823	0.807	0.959	**0.968**
	1.6	0.741	0.887	0.759	0.741	0.732	0.804	0.776	0.942	**0.953**
xalan	2.4	0.801	0.857	0.819	0.803	0.802	0.815	0.814	0.934	**0.943**
	2.5	0.705	0.849	0.735	0.738	0.750	0.778	0.780	0.940	**0.942**
	2.6	0.765	0.863	0.793	0.770	0.771	0.804	0.794	0.940	**0.955**
average		0.761	0.852	0.776	0.789	0.786	0.822	0.808	0.942	**0.950**

Table 12
F1 values of different methods in cross-version prediction.

project	version (Tr⇒T)	traditional	CNN	BiLSTM	deepwalk	node2vec	GCN	GAT	CGCN	CGCN-α
ant	1.4⇒1.6	0.415	0.459	0.423	0.395	0.420	0.404	0.487	0.357	**0.550**
	1.6⇒1.7	0.545	0.548	0.539	0.539	0.543	0.546	0.347	0.481	**0.584**
camel	1.2⇒1.4	0.321	0.219	0.345	0.327	0.347	0.355	0.378	0.134	**0.445**
	1.4⇒1.6	0.363	0.402	0.372	0.307	0.288	**0.434**	0.281	0.336	0.386
jedit	3.2⇒4.0	0.538	0.542	0.520	0.514	0.530	0.519	0.482	0.485	**0.627**
	4.0⇒4.1	0.589	0.489	0.493	0.494	0.548	0.544	**0.606**	0.400	0.546
lucene	2.0⇒2.2	0.582	0.545	0.669	0.495	0.769	0.590	0.505	0.790	**0.850**
	2.2⇒2.4	0.765	0.768	0.767	0.766	0.768	0.768	0.547	0.433	**0.793**
poi	1.5⇒2.5	0.817	0.840	0.840	0.836	0.848	0.840	0.771	0.912	**0.947**
	2.5⇒3.0	0.802	0.266	0.756	0.673	0.801	0.699	0.781	0.755	**0.837**
velocity	1.4⇒1.5	0.758	0.728	0.775	0.780	0.766	0.671	0.701	**0.839**	0.796
	1.5⇒1.6	0.571	0.508	0.594	0.574	0.511	0.547	0.517	0.681	**0.819**
xalan	2.4⇒2.5	**0.675**	0.531	0.478	0.085	0.653	0.439	0.670	0.368	0.532
	2.5⇒2.6	0.649	**0.725**	0.459	0.445	0.638	0.216	0.253	0.532	0.723
average		0.599	0.541	0.574	0.516	0.602	0.541	0.523	0.536	**0.674**

Table 13
AUC values of different methods in cross-version prediction.

project	version (Tr⇒T)	traditional	CNN	BiLSTM	deepwalk	node2vec	GCN	GAT	CGCN	CGCN-α
ant	1.4⇒1.6	0.559	0.649	0.581	0.569	0.587	0.565	0.648	0.555	**0.752**
	1.6⇒1.7	0.801	0.771	0.754	0.790	0.784	0.793	0.769	**0.817**	0.801
camel	1.2⇒1.4	0.622	0.432	0.637	0.659	0.654	0.642	0.730	0.213	**0.751**
	1.4⇒1.6	0.619	0.651	0.601	0.584	0.636	**0.712**	0.685	0.251	0.679
jedit	3.2⇒4.0	0.757	0.765	0.773	0.762	0.792	0.748	0.767	0.741	**0.809**
	4.0⇒4.1	0.806	0.687	0.737	0.766	0.805	0.791	**0.807**	0.619	0.788
lucene	2.0⇒2.2	0.627	0.611	0.654	0.582	0.656	0.616	0.651	0.834	**0.870**
	2.2⇒2.4	0.713	0.706	0.706	0.702	0.704	0.708	0.578	0.477	**0.784**
poi	1.5⇒2.5	0.745	0.760	0.773	0.726	0.772	0.761	0.715	0.876	**0.938**
	2.5⇒3.0	0.801	0.431	0.670	0.593	0.811	0.618	0.738	0.720	**0.844**
velocity	1.4⇒1.5	0.480	0.484	0.423	0.427	0.479	0.416	0.402	**0.790**	0.681
	1.5⇒1.6	0.695	0.640	0.704	0.741	0.721	0.707	0.667	0.870	**0.917**
xalan	2.4⇒2.5	0.498	**0.656**	0.563	0.503	0.652	0.594	0.628	0.549	0.551
	2.5⇒2.6	0.642	**0.826**	0.484	0.445	0.754	0.329	0.678	0.532	0.766
average		0.669	0.648	0.647	0.632	0.701	0.643	0.676	0.632	**0.781**

Table 14
Accuracy values of different methods in cross-version prediction.

project	version (Tr⇒T)	traditional	CNN	BiLSTM	deepwalk	node2vec	GCN	GAT	CGCN	CGCN-α
ant	1.4⇒1.6	0.580	0.691	0.507	0.563	0.653	0.493	0.606	0.633	**0.738**
	1.6⇒1.7	0.690	0.732	0.736	0.713	0.724	0.730	**0.799**	0.549	0.753
camel	1.2⇒1.4	0.522	0.334	0.666	0.434	0.682	0.540	**0.820**	0.166	0.637
	1.4⇒1.6	0.521	0.551	0.486	0.684	**0.733**	0.579	0.781	0.209	0.419
jedit	3.2⇒4.0	0.672	0.666	0.590	0.638	0.655	0.659	0.502	0.594	**0.785**
	4.0⇒4.1	0.739	0.686	0.649	0.712	0.635	0.652	**0.753**	0.679	0.639
lucene	2.0⇒2.2	0.568	0.541	0.620	0.537	0.624	0.550	0.555	0.751	**0.799**
	2.2⇒2.4	0.627	0.623	0.623	0.620	0.623	0.623	0.525	0.466	**0.738**
poi	1.5⇒2.5	0.706	0.757	0.757	0.752	0.776	0.760	0.704	0.876	**0.927**
	2.5⇒3.0	0.684	0.393	0.672	0.585	0.679	0.614	0.710	0.656	**0.761**
velocity	1.4⇒1.5	0.618	0.580	0.642	0.646	0.632	0.519	0.561	**0.755**	0.684
	1.5⇒1.6	0.537	0.454	0.590	0.542	0.344	0.555	0.392	0.797	**0.863**
xalan	2.4⇒2.5	0.519	0.597	0.549	0.497	**0.622**	0.545	0.520	0.537	0.524
	2.5⇒2.6	0.615	0.707	0.447	0.477	0.519	0.446	0.599	0.431	**0.733**
average		0.614	0.594	0.610	0.600	0.636	0.590	0.630	0.578	**0.714**

Table 15
F1 values of different methods in cross-project prediction.

Train project	Test project	traditional	CNN	BiLSTM	deepwalk	node2vec	GCN	GAT	CGCN	CGCN-α
camel	ant	0.321	0.340	0.338	0.404	0.356	0.359	0.274	0.372	**0.497**
jedit		0.352	0.351	0.385	0.392	0.400	0.403	0.489	0.458	**0.596**
lucene		0.336	0.346	0.362	0.353	0.372	0.400	0.374	0.441	**0.514**
poi		0.382	0.388	0.368	0.383	0.327	0.367	0.411	0.460	**0.527**
velocity		0.359	0.385	0.370	0.302	0.329	0.263	0.363	0.309	**0.432**
xalan		0.367	0.374	0.330	0.327	0.388	0.359	0.350	0.434	**0.595**
ant	camel	0.494	0.506	0.497	0.518	0.435	0.523	0.380	0.437	**0.533**
jedit		0.394	0.405	0.430	0.471	0.376	0.325	**0.475**	0.394	0.427
lucene		0.348	0.421	0.479	0.398	0.539	0.423	0.362	0.439	**0.552**
poi		**0.512**	0.475	0.502	0.497	0.489	0.493	0.511	0.462	0.468
velocity		0.512	0.528	0.509	0.506	0.469	0.530	0.516	0.533	**0.572**
xalan		0.423	0.313	0.272	0.299	0.310	0.286	0.413	0.365	**0.538**
ant	jedit	0.456	0.388	0.482	0.351	0.369	**0.613**	0.314	0.555	0.543
camel		0.169	0.564	0.541	0.579	0.551	**0.625**	0.514	0.301	0.579
lucene		0.604	0.644	0.568	0.637	0.514	**0.675**	0.535	0.563	0.667
poi		0.544	0.550	0.630	0.592	0.630	0.624	0.623	0.509	**0.663**
velocity		0.409	0.505	0.411	0.460	0.428	0.382	0.419	0.347	**0.554**
xalan		0.309	0.437	0.610	0.589	0.630	0.234	0.061	0.560	**0.649**
ant	lucene	**0.687**	0.460	0.459	0.316	0.336	0.441	0.408	0.587	0.657
camel		0.595	0.555	0.461	0.508	0.643	0.367	0.099	0.691	**0.774**
jedit		0.454	0.596	0.428	0.591	0.650	0.214	**0.690**	0.119	0.510
poi		0.669	0.647	0.649	0.678	0.680	0.673	0.653	0.709	**0.820**
velocity		0.626	0.669	0.664	0.612	0.565	0.605	0.493	0.616	**0.702**
xalan		0.660	0	0.669	0	0.022	0.674	0.670	0.634	**0.737**
ant	poi	0.698	0.638	0.528	0.603	0.579	0.712	0.659	0.534	**0.737**
camel		0.633	0.194	0.496	0.608	0.590	0.341	0.199	0.245	**0.665**
jedit		0.695	0.404	0.393	0.718	0.404	0.592	0.777	0.767	**0.816**
lucene		0.656	0.733	0.772	0.748	0.754	0.768	0.558	0.764	**0.789**
velocity		0.722	**0.752**	0.737	0.720	0.720	0.715	0.742	0.619	0.732
xalan		0.434	0.472	0.492	0.595	0.014	0.295	0.423	0.440	**0.745**
ant	velocity	0.626	0.800	0.805	0.546	0.804	**0.829**	0.406	0.235	0.730
camel		0.544	0.614	0.834	0.805	0.569	0.801	0.306	0.438	**0.883**
jedit		0.378	0.289	0.161	0.498	0.683	0.217	0.606	0.752	**0.845**
lucene		0.388	0.402	0.725	0.521	0.864	0.565	0.302	0.557	**0.739**
poi		0.801	0.687	0.785	0.709	0.667	0.716	0.447	0.780	**0.874**
xalan		0.263	0.258	0	0.408	0.519	0.589	**0.864**	0.186	0.618
ant	xalan	0.241	0.208	0.198	0.240	0.188	0.240	0.281	0.295	**0.327**
camel		0.303	0.308	0.315	0.333	0.344	0.359	0.409	0.325	**0.528**
jedit		0.408	0.393	0.288	0.437	0.377	0.416	0.372	0.373	**0.447**
lucene		0.438	0.399	0.335	0.391	0.288	0.407	0.340	0.424	**0.445**
poi		0.300	0.315	0.359	0.347	0.364	0.354	0.351	0.305	**0.432**
velocity		0.262	0.282	0.297	0.261	0.272	0.255	0.253	0.254	**0.321**
average		0.471	0.452	0.475	0.482	0.472	0.477	0.445	0.466	**0.614**

Table 16
AUC values of different methods in cross-project prediction.

Train project	Test project	traditional	CNN	BiLSTM	deepwalk	node2vec	GCN	GAT	CGCN	CGCN-α
camel	ant	0.485	0.516	0.541	0.532	0.497	0.563	0.545	0.367	**0.750**
jedit		0.550	0.528	0.566	0.579	0.591	0.560	0.723	0.785	**0.819**
lucene		0.549	0.551	0.559	0.536	0.576	0.572	0.585	0.653	**0.755**
poi		0.562	0.611	0.595	0.552	0.566	0.567	0.670	0.703	**0.803**
velocity		0.454	0.565	0.480	0.501	0.496	0.459	0.513	0.509	**0.596**
xalan		0.563	0.565	0.570	0.572	0.581	0.574	0.576	0.760	**0.819**
ant	camel	0.500	0.491	0.478	0.539	0.498	0.526	0.513	0.552	**0.595**
jedit		0.524	0.510	0.532	**0.565**	0.509	0.494	0.522	0.479	0.509
lucene		0.486	0.562	0.537	0.547	0.562	0.563	0.486	0.603	**0.697**
poi		0.559	0.581	0.557	0.552	0.546	0.556	0.580	0.438	**0.647**
velocity		0.567	0.547	0.564	0.502	0.476	0.572	0.537	0.597	**0.640**
xalan		0.531	0.534	0.534	0.528	0.619	0.530	0.518	0.683	**0.706**
ant	jedit	0.556	0.511	0.589	0.481	0.577	0.711	0.707	0.726	**0.731**
camel		0.339	0.690	0.688	0.772	0.742	0.771	**0.805**	0.395	0.725
lucene		0.702	0.756	0.776	0.757	0.780	**0.801**	0.715	0.721	0.777
poi		**0.815**	0.658	0.778	0.782	0.764	0.794	0.772	0.679	0.800
velocity		0.450	0.654	0.491	0.552	0.515	0.410	0.466	0.448	**0.690**
xalan		0.506	0.704	0.743	0.719	0.764	0.717	0.464	0.694	**0.786**
ant	lucene	0.489	0.428	0.471	0.462	0.489	0.432	0.507	**0.615**	0.555
camel		0.550	0.518	0.414	0.451	0.569	0.408	0.305	0.708	**0.822**
jedit		0.624	0.636	0.597	0.621	0.655	0.586	**0.684**	0.206	0.601
poi		0.751	0.644	0.674	0.705	0.663	0.678	0.656	0.757	**0.878**
velocity		0.544	0.667	0.550	0.566	0.545	0.529	0.468	0.588	**0.686**
xalan		0.696	0.559	0.749	0.527	0.547	0.751	0.665	0.682	**0.778**
ant	poi	0.432	0.509	0.396	0.613	0.563	0.622	0.609	0.460	**0.633**
camel		0.584	**0.631**	0.429	0.456	0.549	0.486	0.531	0.214	0.523
jedit		0.628	0.633	0.565	0.614	0.640	0.697	0.748	0.719	**0.762**
lucene		0.664	0.704	0.732	0.693	0.729	0.718	0.677	0.724	**0.745**
velocity		0.421	**0.653**	0.612	0.552	0.405	0.514	0.558	0.480	0.560
xalan		0.626	0.643	0.648	**0.698**	0.583	0.517	0.642	0.578	0.665
ant	velocity	0.504	0.463	0.629	0.546	0.460	0.624	0.461	0.304	**0.720**
camel		0.437	0.433	0.520	0.539	0.515	0.679	0.678	0.261	**0.755**
jedit		0.474	0.632	0.553	0.572	0.585	0.420	0.558	0.326	**0.652**
lucene		0.416	0.513	0.403	0.559	0.388	0.496	0.323	0.365	**0.640**
poi		0.401	0.515	0.612	0.444	0.519	0.475	0.384	0.547	**0.721**
xalan		**0.554**	0.419	0.416	0.424	0.441	0.461	0.386	0.416	0.524
ant	xalan	0.465	0.393	0.355	0.459	0.360	0.460	0.533	0.519	**0.579**
camel		0.604	0.629	0.682	0.653	0.645	0.727	**0.775**	0.756	0.774
jedit		0.692	0.699	**0.762**	0.738	0.679	0.705	0.720	0.655	0.743
lucene		0.741	0.697	**0.790**	0.698	0.765	0.747	0.734	0.768	0.754
poi		0.744	0.652	0.710	**0.780**	0.690	0.746	0.689	0.646	0.734
velocity		0.527	**0.651**	0.591	0.523	0.520	0.517	0.501	0.490	0.593
average		0.554	0.582	0.582	0.582	0.575	0.589	0.583	0.561	**0.696**

marked in bold. We average the results for each project. For example, the result of ant is average results of ant-1.4, ant-1.6, and ant-1.7 in within-version prediction, average results of ant-1.4⇒ant-1.6 and ant-1.6⇒ant-1.7 in cross-version prediction, and average results of camel-1.2⇒ant-1.4, jedit-3.2⇒ant-1.4, lucene-2.0⇒ant-1.4, poi-1.5⇒ant-1.4, velocity-1.4⇒ant-1.4, and xalan-2.4⇒ant-1.4 in cross-project prediction.

(1) For within-version prediction, $add(\cdot)$ is a bit better than $concatenate(\cdot)$ and $multiply(\cdot)$ on average. The results of the Wilcoxon signed-rank test (p-value) indicate that the performance of $add(\cdot)$ and $concatenate(\cdot)$ are significantly better than $multiply(\cdot)$, but there is no significant difference between $add(\cdot)$ and $concatenate(\cdot)$. The Cliff's δ value indicates that $add(\cdot)$ slightly outperforms $concatenate(\cdot)$ in terms of F1 and AUC.

(2) For cross-version prediction, $concatenate(\cdot)$ is a bit better than $add(\cdot)$ and $multiply(\cdot)$ on average. However, there is no statistically significant difference between the three combinations according to the p-values. The Cliff's δ value indicates that $add(\cdot)$ and $concatenate(\cdot)$ slightly outperform $multiply(\cdot)$, and $concatenate(\cdot)$ slightly outperforms $add(\cdot)$.

(3) For cross-project prediction, $add(\cdot)$ is a bit better than $concatenate(\cdot)$ and $multiply(\cdot)$ in terms of F1 and Accuracy, and $concatenate(\cdot)$ is a bit better than $add(\cdot)$ and $multiply(\cdot)$ in terms of AUC. The p-value indicates no statistically significant difference between the three combinations, the only exception is $concatenate(\cdot)$ and $multiply(\cdot)$ in terms of AUC. The Cliff's δ value indicates that $add(\cdot)$ slightly outperforms $concatenate(\cdot)$ in terms of F1 and AUC.

The results demonstrate that $multiply(\cdot)$ performs poorly in the three combinations, while $add(\cdot)$ and $concatenate(\cdot)$ do not differ much. Fig. 11 shows that $multiply(\cdot)$ can highlight some features, but may zero out some important features in some cases. $add(\cdot)$ and $concatenate(\cdot)$ can keep the important features in most cases. Therefore, for convenience and comparability, the experiments in Section 6 use $add(\cdot)$ for all three prediction tasks.

7.2. *The impact of the re-sampling method on the model*

To further explore the impact of different re-sampling methods on our method, we adopt the three re-sampling methods, i.e., SMOTE, RUS, and SMOTETomek. Tables 22–24 present the performance results of the three re-sampling methods on CGCN. Table 25 presents the statistical significance of the three re-sampling methods.

(1) For within-version prediction, SMOTE is a bit better than SMOTETomek and RUS in terms of F1 and Accuracy. The p-value indicates that the performance of SMOTE and SMOTETomek are significantly better than RUS in terms of F1 and Accuracy, but there is no significant difference between SMOTE and SMOTETomek. The Cliff's

Table 17
Accuracy values of different methods in cross-project prediction.

Train project	Test project	traditional	CNN	BiLSTM	deepwalk	node2vec	GCN	GAT	CGCN	CGCN-α
camel	ant	0.349	0.446	0.509	0.360	0.360	0.326	**0.697**	0.229	0.549
jedit		0.537	0.366	0.451	0.469	0.486	0.560	0.617	**0.743**	0.737
lucene		0.571	0.503	0.314	0.497	0.229	0.520	0.617	0.566	**0.697**
poi		0.280	0.531	0.411	0.354	0.411	0.389	0.491	0.611	**0.754**
velocity		0.326	0.269	0.280	0.446	0.394	0.583	0.377	**0.514**	0.474
xalan		0.469	0.560	0.606	0.600	0.423	0.571	0.554	0.731	**0.806**
ant	camel	0.408	0.362	0.377	0.472	0.465	0.403	0.514	0.536	**0.557**
jedit		0.564	0.512	0.569	0.557	0.522	**0.576**	0.491	0.516	0.522
lucene		0.573	0.567	0.446	0.561	0.374	0.538	0.519	0.562	**0.635**
poi		0.400	0.533	0.491	0.503	0.505	0.498	0.503	0.351	**0.638**
velocity		0.420	0.427	0.455	0.375	0.415	0.441	0.413	**0.545**	0.524
xalan		0.509	0.606	0.619	0.635	0.661	0.619	0.533	0.645	**0.697**
ant	jedit	0.623	0.538	0.562	0.573	0.658	0.669	0.681	0.735	**0.735**
camel		0.546	0.662	0.485	0.508	0.523	0.700	**0.723**	0.500	0.692
lucene		0.692	**0.719**	0.485	0.715	0.346	0.704	0.692	0.535	0.708
poi		0.427	0.635	0.635	0.538	0.665	0.615	0.712	0.377	**0.735**
velocity		0.388	0.373	0.362	0.531	0.435	0.365	0.446	0.465	**0.696**
xalan		0.638	0.692	0.681	**0.700**	0.665	0.673	0.646	0.631	0.692
ant	lucene	0.547	0.481	0.492	0.497	0.519	0.453	0.519	0.580	**0.602**
camel		0.519	0.486	0.431	0.464	0.492	0.448	0.497	0.669	**0.735**
jedit		0.575	0.580	0.541	0.580	0.619	0.514	**0.652**	0.265	0.586
poi		0.530	0.602	0.630	0.591	0.652	0.613	0.547	0.641	**0.801**
velocity		0.492	0.514	0.558	0.552	0.497	0.503	0.431	0.497	**0.657**
xalan		0.641	0.497	0.503	0.497	0.503	0.514	0.613	0.630	**0.740**
ant	poi	0.548	0.557	0.404	0.583	0.566	0.610	0.596	0.487	**0.640**
camel		**0.553**	0.452	0.430	0.496	0.518	0.474	0.434	0.272	0.531
jedit		0.623	0.522	0.513	0.596	0.522	0.601	0.697	0.645	**0.746**
lucene		0.610	0.671	0.658	0.684	0.605	0.684	0.583	0.693	**0.728**
velocity		0.570	0.610	**0.618**	0.575	0.566	0.566	0.596	0.482	0.579
xalan		0.509	0.539	0.557	0.605	0.399	0.456	0.522	0.509	**0.675**
ant	velocity	0.521	0.677	0.693	0.464	0.682	**0.729**	0.391	0.286	0.646
camel		0.458	0.495	0.734	0.693	0.479	0.714	0.339	0.318	**0.823**
jedit		0.349	0.359	0.297	0.464	0.583	0.286	0.526	0.609	**0.760**
lucene		0.359	0.380	0.578	0.453	**0.760**	0.464	0.276	0.438	0.651
poi		0.677	0.563	0.693	0.573	0.557	0.583	0.344	0.656	**0.786**
xalan		0.328	0.313	0.240	0.365	0.422	0.484	**0.760**	0.271	0.510
ant	xalan	0.493	0.422	0.263	0.521	0.226	0.325	0.462	0.541	**0.675**
camel		0.451	0.459	0.402	0.481	0.499	0.464	0.563	0.359	**0.793**
jedit		0.731	0.618	0.168	0.702	0.694	0.714	0.536	0.756	**0.805**
lucene		0.694	0.665	0.351	0.620	0.168	0.609	0.423	0.635	**0.711**
poi		0.235	0.448	0.515	0.388	0.567	0.419	0.519	0.410	**0.711**
velocity		0.400	0.198	0.457	0.334	0.478	0.396	0.526	0.510	**0.549**
average		0.503	0.510	0.487	0.528	0.503	0.533	0.538	0.523	**0.674**

Table 18
Comparison results of the three types of feature combination, in within-version prediction.

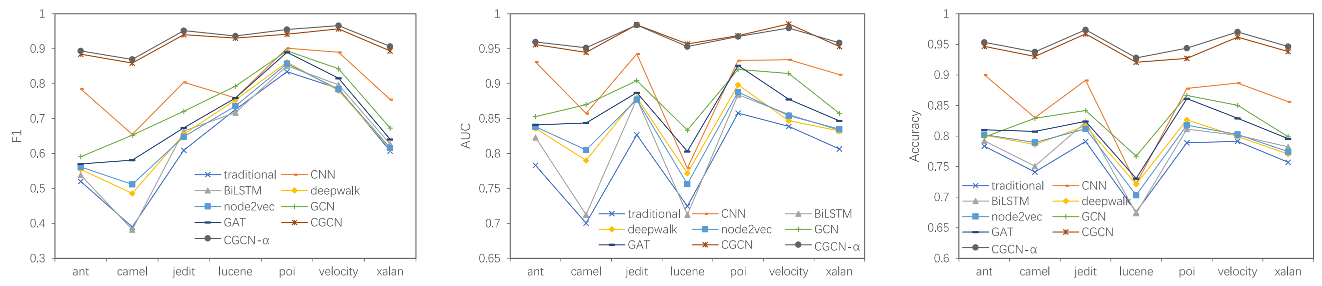
Project	Add			Concatenate			Multiply		
	F1	AUC	Accuracy	F1	AUC	Accuracy	F1	AUC	Accuracy
ant	**0.884**	0.956	**0.947**	0.882	**0.959**	0.947	0.844	0.948	0.927
camel	**0.859**	**0.945**	**0.930**	0.853	0.944	0.929	0.821	0.922	0.912
jedit	**0.940**	**0.984**	**0.967**	0.929	0.979	0.962	0.903	0.97	0.946
lucene	**0.931**	**0.957**	**0.921**	0.924	0.948	0.913	0.917	0.951	0.905
poi	0.942	0.969	0.927	**0.950**	**0.976**	**0.938**	0.939	0.965	0.925
velocity	0.957	**0.985**	0.962	**0.964**	0.984	**0.968**	0.950	0.981	0.953
xalan	**0.894**	0.953	**0.938**	0.893	**0.956**	0.938	0.887	0.954	0.929
average	**0.915**	**0.964**	**0.942**	0.914	0.964	0.942	0.895	0.956	0.928

δ value indicates that SMOTE slightly outperforms SMOTETomek in terms of F1 and Accuracy.

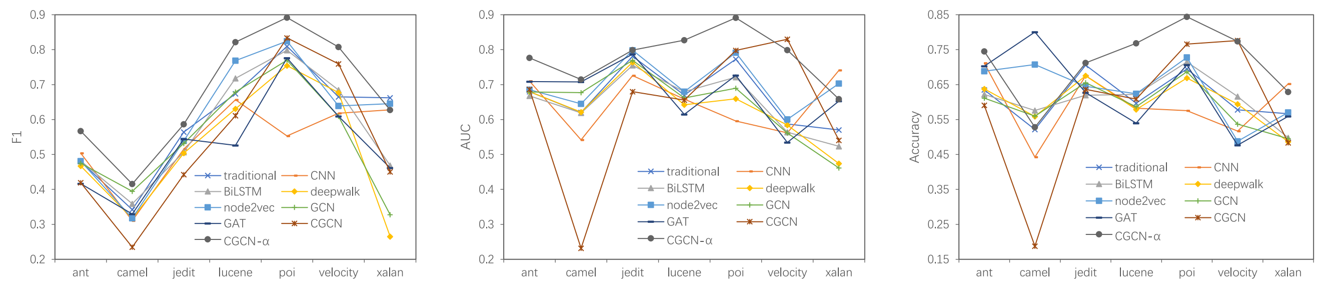
(2) For cross-version prediction, RUS is a bit better than SMOTE and SMOTETomek in terms of F1 and AUC, and SMOTE performs the best in terms of Accuracy. The p-value indicates that the performance of RUS is significantly better than SMOTE and SMOTETomek in most of the cases, but there is no significant difference between SMOTE and SMOTETomek. The Cliff's δ value indicates that SMOTE slightly outperforms SMOTETomek.

(3) For cross-project prediction, similar to the results in cross-version prediction, RUS is a bit better than SMOTE and SMOTETomek. The p-value indicates that the performance of RUS is significantly better than SMOTE and SMOTETomek in terms of F1 and AUC, but there is no significant difference between SMOTE and SMOTETomek. The Cliff's δ value indicates that SMOTE slightly outperforms SMOTETomek.

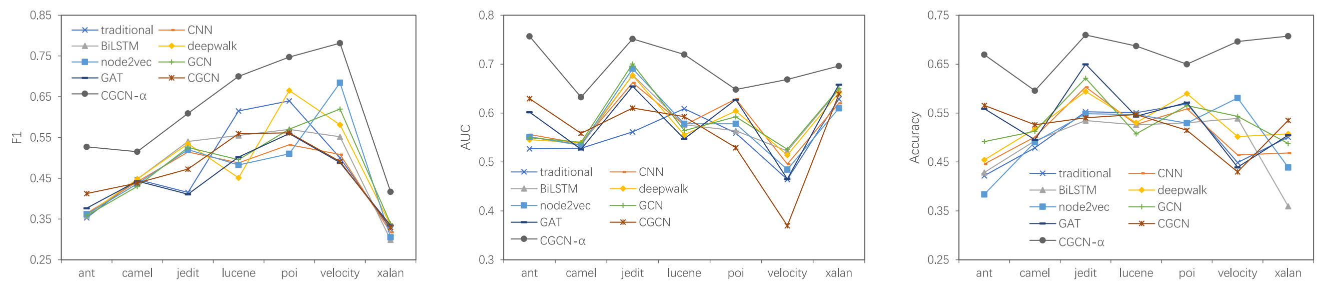
The results demonstrate that SMOTE and SMOTETomek are more suitable for within-version prediction, while RUS is more suitable for cross-version and cross-project predictions. On the one hand, the instances in the training and testing set are entirely random, and we could repeat the experiment 25 times by randomly changing the distribution of the data instances to mitigate the bias in within-version prediction. However, since the training set is a complete project in cross-version



(a) Performance comparison among different models in within-version prediction



(b) The performance comparison among different models in cross-version prediction



(c) The performance comparison among different models in cross-project prediction

Fig. 10. The performance comparison among different models.

Table 19
Comparison results of the three types of feature combination, in cross-version prediction.

Project	Add			Concatenate			Multiply		
	F1	AUC	Accuracy	F1	AUC	Accuracy	F1	AUC	Accuracy
ant	0.419	0.686	0.591	**0.514**	**0.733**	0.649	0.468	0.663	**0.699**
camel	**0.235**	0.232	0.188	0.111	0.232	0.364	0.191	**0.541**	**0.596**
jedit	0.442	**0.680**	0.636	**0.502**	0.675	**0.640**	0.360	0.582	0.600
lucene	0.611	0.655	0.609	0.689	**0.709**	0.615	**0.765**	0.679	**0.640**
poi	**0.834**	**0.798**	**0.766**	0.810	0.764	0.716	0.714	0.604	0.635
velocity	**0.760**	**0.830**	**0.776**	0.743	0.796	0.763	0.689	0.753	0.603
xalan	0.450	0.541	0.484	**0.531**	**0.643**	**0.601**	0.329	0.459	0.461
average	0.536	0.632	0.578	**0.557**	**0.650**	**0.621**	0.502	0.612	0.605

Table 20
Comparison results of the three types of feature combination, in cross-project prediction.

Project	Add			Concatenate			Multiply		
	F1	AUC	Accuracy	F1	AUC	Accuracy	F1	AUC	Accuracy
ant	0.412	0.630	0.566	0.400	**0.652**	0.590	**0.422**	0.608	**0.592**
camel	0.438	0.559	0.526	**0.438**	**0.598**	**0.541**	0.388	0.576	0.514
jedit	**0.473**	0.610	0.540	0.469	**0.637**	**0.637**	0.405	0.563	0.529
lucene	**0.559**	**0.593**	**0.547**	0.412	0.442	0.460	0.441	0.389	0.404
poi	0.561	0.529	0.515	**0.564**	**0.565**	**0.531**	0.481	0.411	0.483
velocity	0.491	0.370	0.430	0.244	0.426	0.306	**0.561**	**0.459**	**0.510**
xalan	0.329	0.639	0.535	**0.343**	**0.639**	**0.580**	0.326	0.633	0.510
average	**0.466**	0.561	**0.523**	0.410	**0.566**	0.521	0.432	0.520	0.506

Table 21
Statistical test results of the three types of feature combination.

Task	Metrics	p-value			Cliff's delta		
		add vs concatenate	add vs multiply	concatenate vs multiply	add vs concatenate	add vs multiply	concatenate vs multiply
within-version	F1	>0.05	**<0.05**	**<0.05**	0.0159	0.2744	0.2517
	AUC	>0.05	**<0.05**	**<0.05**	0.025	0.2018	0.1837
	Accuracy	>0.05	**<0.05**	**<0.05**	−0.014	0.306	0.3197
cross-version	F1	>0.05	>0.05	>0.05	−0.1327	0.0918	0.1531
	AUC	>0.05	>0.05	>0.05	−0.0306	0.1020	0.20408
	Accuracy	>0.05	>0.05	>0.05	−0.1122	−0.0306	0.1020
cross-project	F1	>0.05	>0.05	>0.05	0.188	0.1355	−0.0368
	AUC	>0.05	>0.05	**<0.05**	0.0147	0.1848	0.1905
	Accuracy	>0.05	>0.05	>0.05	−0.0119	0.0816	0.0697

Table 22
Comparison results of the re-sampling methods, in within-version prediction.

Project	SMOTE			SMOTETomek			RUS		
	F1	AUC	Accuracy	F1	AUC	Accuracy	F1	AUC	Accuracy
ant	**0.884**	**0.956**	**0.947**	0.878	0.956	0.943	0.865	0.956	0.935
camel	**0.859**	0.945	**0.930**	0.857	**0.946**	0.930	0.836	0.946	0.915
jedit	**0.940**	**0.984**	**0.967**	0.940	0.984	0.966	0.935	0.984	0.963
lucene	0.931	0.957	0.921	0.932	0.956	0.922	**0.933**	**0.958**	**0.923**
poi	**0.942**	0.969	**0.927**	0.941	**0.970**	0.927	0.939	0.969	0.924
velocity	0.957	0.985	0.962	0.957	**0.986**	0.961	**0.959**	0.985	**0.963**
xalan	**0.894**	0.953	**0.938**	0.893	0.952	0.937	0.886	**0.954**	0.934
average	**0.915**	0.964	**0.942**	0.914	0.964	0.941	0.908	**0.965**	0.937

Table 23
Comparison results of the re-sampling methods, in cross-version prediction.

Project	SMOTE			SMOTETomek			RUS		
	F1	AUC	Accuracy	F1	AUC	Accuracy	F1	AUC	Accuracy
ant	0.351	0.637	0.688	0.382	0.635	**0.693**	**0.419**	**0.686**	0.591
camel	0.193	0.223	**0.253**	0.219	0.229	0.245	**0.235**	**0.232**	0.188
jedit	0.335	0.548	**0.758**	0.317	0.531	0.655	**0.442**	**0.680**	0.636
lucene	0.611	0.655	0.609	0.611	0.655	0.609	0.611	0.655	0.609
poi	0.834	0.798	0.766	0.834	0.798	0.766	0.834	0.798	0.766
velocity	0.760	0.830	0.776	0.760	0.830	0.776	0.760	0.830	0.776
xalan	0.450	0.533	0.481	0.347	0.509	0.480	**0.450**	**0.541**	**0.484**
average	0.505	0.603	**0.619**	0.496	0.598	0.603	**0.536**	**0.632**	0.578

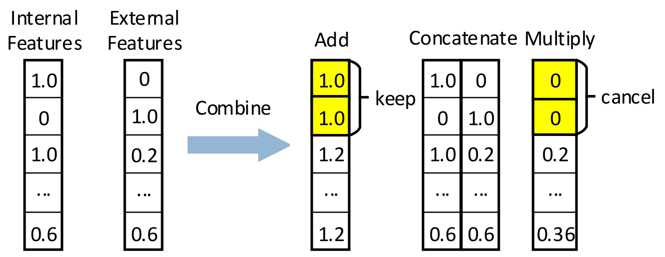


Fig. 11. Three types of feature combination.

and cross-project predictions and the data distribution could not be changed, the probability of prediction bias is higher than within-version prediction. On the other hand, if there is a considerable distribution difference in cross-version and cross-project predictions, the synthetic instances of SMOTE may not be good, so RUS can avoid the noisy data generated by over-sampling.

7.3. The impact of the classifier on the model

In this section, we select the four classifiers, i.e., Multi-Layer Perceptron (MLP), Decision Tree (DT), Logistic Regression (LR), and Random Forest (RF). Tables 26–28 present the performance results of the four classifiers on CGCN. The highest value of each row is marked in bold. Table 29 presents the statistical significance of the four classifiers.

(1) For within-version prediction, on average, RF outperforms the other classifiers, with significant differences in both p-value and Cliff's δ.

(2) For cross-version prediction, on average, MLP is a bit better than the other classifiers. The p-value indicates no statistically significant difference between these classifiers, the only exception is MLP and RF in terms of F1. The Cliff's δ value indicates that MLP slightly outperforms the other three classifiers.

(3) For cross-project prediction, similar to the results in cross-version prediction, MLP is a bit better than the other classifiers in terms of F1 and AUC, and RF performs the best in terms of Accuracy. The p-value indicates no statistically significant difference between these classifiers, the only exception is MLP and RF in terms of F1 and AUC. The Cliff's δ value indicates that MLP slightly outperforms the other three classifiers in most cases.

The results demonstrate that RF performs the best in within-version prediction but poorly in cross-version and cross-project predictions. The data distribution difference between different versions or projects may weaken the transferability of a classifier. Therefore, based on the above experimental results, we can find that RF had poor transferability. Although MLP performed slightly better than the other classifiers in cross-version and cross-project predictions, the performance was still unsatisfactory. Thus, reducing the distribution difference between training and testing sets may be more urgent than choosing an optimal classifier.

Table 24
Comparison results of the re-sampling methods, in cross-project prediction.

Project	SMOTE			SMOTETomek			RUS		
	F1	AUC	Accuracy	F1	AUC	Accuracy	F1	AUC	Accuracy
ant	0.428	**0.634**	0.569	**0.440**	0.612	**0.574**	0.412	0.630	0.566
camel	0.364	0.541	0.548	0.360	0.538	**0.552**	**0.438**	**0.559**	0.526
jedit	0.402	0.550	0.500	0.421	0.572	0.517	**0.473**	**0.610**	**0.540**
lucene	0.446	0.553	0.509	0.461	0.534	0.490	**0.559**	**0.593**	**0.547**
poi	0.429	0.488	0.471	0.400	0.465	0.453	**0.561**	**0.529**	**0.515**
velocity	0.430	0.339	0.409	0.374	0.306	0.368	**0.491**	**0.370**	**0.430**
xalan	0.326	0.610	**0.597**	0.329	0.618	0.576	**0.329**	**0.639**	0.535
average	0.403	0.531	0.515	0.398	0.521	0.504	**0.466**	**0.561**	**0.523**

Table 25
Statistical test results of the three re-sampling methods.

Task	Metrics	p-value			Cliff's delta		
		SMOTE vs SMOTETomek	SMOTE vs RUS	SMOTETomek vs RUS	SMOTE vs SMOTETomek	SMOTE vs RUS	SMOTETomek vs RUS
within-version	F1	>0.05	**<0.05**	**<0.05**	0.0249	0.0476	0.0385
	AUC	>0.05	>0.05	>0.05	−0.0068	−0.0023	0.0023
	Accuracy	>0.05	**<0.05**	**<0.05**	0.0499	0.1066	0.0839
cross-version	F1	>0.05	**<0.05**	**<0.05**	0.0357	−0.1276	−0.0969
	AUC	>0.05	>0.05	**<0.05**	0.0663	−0.0867	−0.1173
	Accuracy	>0.05	**<0.05**	**<0.05**	0.0969	0.1378	0.0969
cross-project	F1	>0.05	**<0.05**	**<0.05**	0.0023	−0.1848	−0.1803
	AUC	>0.05	**<0.05**	**<0.05**	0.023	−0.099	−0.1179
	Accuracy	>0.05	>0.05	>0.05	0.0300	−0.0176	−0.059

Table 26
Comparison results of the classifiers, in within-version prediction.

Project	MLP			DT			LR			RF		
	F1	AUC	Accuracy	F1	AUC	Accuracy	F1	AUC	Accuracy	F1	AUC	Accuracy
ant	0.753	0.910	0.874	0.855	0.922	0.930	0.767	0.921	0.882	**0.884**	**0.956**	**0.947**
camel	0.785	0.924	0.890	0.803	0.883	0.895	0.774	0.942	0.875	**0.859**	**0.945**	**0.930**
jedit	0.865	0.954	0.923	0.912	0.944	0.950	0.890	0.972	0.937	**0.940**	**0.984**	**0.967**
lucene	0.867	0.905	0.845	**0.933**	0.920	0.924	0.916	0.957	0.903	0.931	**0.957**	**0.921**
poi	0.924	0.940	0.903	0.931	0.910	0.912	0.932	0.961	0.912	**0.942**	**0.969**	**0.927**
velocity	0.866	0.897	0.890	0.958	0.947	0.957	0.944	0.974	0.947	**0.957**	**0.985**	**0.962**
xalan	0.808	0.905	0.888	0.847	0.890	0.898	0.820	0.939	0.891	**0.894**	**0.953**	**0.938**
average	0.838	0.919	0.888	0.891	0.917	0.924	0.863	0.952	0.907	**0.915**	**0.964**	**0.942**

Table 27
Comparison results of the classifiers, in cross-version prediction.

Project	MLP			DT			LR			RF		
	F1	AUC	Accuracy	F1	AUC	Accuracy	F1	AUC	Accuracy	F1	AUC	Accuracy
ant	0.419	0.686	**0.591**	0.162	0.296	0.387	**0.491**	**0.790**	0.544	0.138	0.364	0.520
camel	0.235	0.232	0.188	0.224	**0.380**	**0.502**	**0.246**	0.160	0.168	0.072	0.107	0.333
jedit	**0.442**	0.680	0.636	0.204	0.5	0.496	0.333	0.550	0.652	0.413	**0.743**	**0.809**
lucene	0.611	0.655	0.609	0	0.5	0.376	**0.793**	**0.764**	**0.705**	0.059	0.722	0.394
poi	0.834	0.798	0.766	**0.904**	**0.807**	**0.861**	0.093	0.038	0.169	0.455	0.618	0.608
velocity	**0.760**	0.830	**0.776**	0.362	0.623	0.539	0.495	0.595	0.571	0.417	**0.952**	0.610
xalan	0.450	**0.541**	0.484	**0.653**	0.5	**0.485**	0.565	0.470	0.473	0.037	0.255	0.481
average	**0.536**	**0.632**	**0.578**	0.358	0.515	0.521	0.431	0.481	0.469	0.227	0.537	0.536

Table 28
Comparison results of the classifiers, in cross-project prediction.

Project	MLP			DT			LR			RF		
	F1	AUC	Accuracy	F1	AUC	Accuracy	F1	AUC	Accuracy	F1	AUC	Accuracy
ant	0.412	0.630	0.566	**0.438**	**0.638**	**0.733**	0.359	0.446	0.348	0.158	0.617	0.778
camel	**0.438**	**0.559**	0.526	0.429	0.511	0.463	0.360	0.485	0.540	0.181	0.466	**0.601**
jedit	**0.473**	**0.610**	0.540	0.060	0.251	0.299	0.444	0.597	0.545	0.185	0.370	**0.590**
lucene	0.559	**0.593**	0.547	**0.606**	0.565	**0.566**	0.448	0.488	0.486	0.228	0.573	0.525
poi	**0.561**	0.529	0.515	0.300	0.437	0.420	0.526	**0.580**	**0.542**	0.112	0.397	0.409
velocity	0.491	0.370	0.430	**0.610**	**0.625**	**0.619**	0.546	0.483	0.504	0.217	0.467	0.309
xalan	**0.329**	**0.639**	0.535	0.147	0.479	0.462	0.207	0.406	0.321	0.080	0.394	**0.692**
average	**0.466**	**0.561**	0.523	0.370	0.501	0.509	0.413	0.498	0.469	0.166	0.469	**0.558**

Table 29
Statistical test results of the classifiers.

Task	Metrics	p-value			Cliff's delta		
		DT vs RF	LR vs RF	MLP vs RF	DT vs RF	LR vs RF	MLP vs RF
within-version	F1	**<0.05**	**<0.05**	**<0.05**	−0.220	−0.333	−0.542
	AUC	**<0.05**	**<0.05**	**<0.05**	−0.850	−0.229	−0.764
	Accuracy	**<0.05**	**<0.05**	**<0.05**	−0.3696	−0.578	−0.8005
		MLP vs DT	MLP vs LR	MLP vs RF	MLP vs DT	MLP vs LR	MLP vs RF
cross-version	F1	>0.05	>0.05	**<0.05**	0.3367	0.2041	0.6122
	AUC	>0.05	>0.05	>0.05	0.3776	0.2143	0.1122
	Accuracy	>0.05	>0.05	>0.05	0.1327	0.2194	0.1429
cross-project	F1	>0.05	>0.05	**<0.05**	0.1916	0.1088	0.689
	AUC	>0.05	>0.05	**<0.05**	0.1735	0.1236	0.2029
	Accuracy	>0.05	>0.05	>0.05	0.0079	0.1888	−0.1429

7.4. Discussion of settings on cross-version and cross-project

In addition to the settings in Section 5, we also experiment with another setting for cross-version and cross-project predictions. Specifically, in cross-version prediction, we use all of the previous versions as training set. For example, we select three versions of each project in our study, so we need to conduct two different runs. In the first run, all the source files in ant-1.4 are used to train a prediction model, and all the source files in ant-1.6 are used to evaluate the model. In the second run, the model is trained using ant-1.4 and ant 1.6, and tested using ant-1.7. Similarly, in cross-project prediction, if the first version of ant (i.e., ant-1.4) is used to evaluate a model, the first versions of the remaining six projects are used to train the model.

In Section 5 settings, the time complexity of CGCN-α is $O(n^2)$ since we need to tune out the optimal parameter α. However, if we use the settings in this subsection, the time complexity of CGCN-α is $O(n^{m-1})$, where m is the number of versions or projects. As the versions or projects increase, the time complexity of the algorithm increases exponentially. Therefore, we select the variant CGCN for experimental comparative analysis in this section.

Tables 30–31 present the performance results of the two settings in cross-version and cross-project predictions. The best value of each row is marked in bold.

(1) Table 30 shows that the setting in this subsection is a bit better than that in Section 5, with the F1, AUC, and Accuracy values improved by 1.7%, 2.1%, and 0.8% on average.

(2) Table 31 shows that the setting in Section 5 is a bit better than that of this subsection, with the F1, AUC, and Accuracy values improved by 18.5%, 10.4%, and 8.8% on average.

The results demonstrate that the setting in this subsection is better than that in Section 5 in cross-version prediction, which indicates that the historical version data is helpful in WPDP. However, the setting in Section 5 is better than this subsection in cross-project prediction, which may be due to the considerable distribution difference between different projects. Therefore, blindly expanding the training data does not usually improve the prediction performance.

7.5. Limitations

In this subsection, we discuss the limitations of our study.

Size of ASTs: The tree-based neural models are vulnerable to gradient vanishing problems, if the transformed ASTs are very large and deep [52–54]. Although transforming ASTs to binary trees can alleviate this problem, it may destroy the original syntactic structure of trees, and weaken the ability of the model to capture syntactic information [55]. In future work, we plan to split the large AST into a set of small trees, or to split the class file into a set of short methods and then convert each method to a small tree.

Length of token sequences: The token sequences extracted from large AST are similar to long texts in Natural Language Processing (NLP). In this study, we make all the lengths equal to the length of the longest sequence, which may result in a waste of computation and make the model difficult to capture long-distance dependencies. In future work, we intend to learn from the long-text processing methods in NLP to handle long token sequences.

Feature weights: In RQ2, parameter α is used to give different weights for internal and external features. Based on our initial intuition, internal features will contribute more to WPDP, and external features will contribute more to CPDP. However, according to the experimental results in Section 6, there is no obvious regularity among the three prediction tasks. In future work, we plan to further expand the dataset and draw statistically significant conclusions. In addition, as shown in Fig. 6, we use a parameter α to assign different weights to $X_{internal}$ and $X_{external}$ in the fusion process of CNN and GCN. The parameter is selected within $[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$. We need to try each value in the list to validate the best value of α during the training process, which has a high computational load. Therefore, in our future work, we intend to design an adaptive fusion module that takes α as a learnable parameter and train it jointly with the model to automatically tune out the optimal value.

Transferability of the model: Due to different scales, functions, and coding rules of software, the data in different projects would show the distribution difference [56]. Even across different versions of the same project, there also can be significant changes, the distribution difference results in poor performance of classifiers in test projects. In future work, we intend to enhance the transferability of the model.

Used weighted digraph: The networks used in our study are undirected graphs. However, the original CDN is a directed graph. In addition, the strength of dependencies between nodes in the network is not the same. Therefore, in the future work, we plan to consider the direction and weight of dependencies in CDN.

8. Conclusion and future work

This work proposes a novel CGCN method, which combines the advantages of CNN and GCN to automatically extract internal semantic information of each file and external structural information between all class files. To generalize our conclusion, we assign different weights to the two types of features, since these two types of features may have different contributions to different projects. We empirically investigate the effectiveness of CGCN on seven open-source projects. The results show that our proposed method produces more stable and better results than baseline models, with stronger feature representation and robustness. Besides, our method is more suitable for within-version prediction. Both CGCN and CGCN-α achieve satisfactory results on the seven projects. However, CGCN-α performs well, but CGCN performs poorly on cross-version and cross-project predictions compared to other baseline methods. Therefore, in our future work, we plan to pay more attention to cross-version and cross-project predictions, and enhance the transferability of the model to improve the performance in cross-version and cross-project predictions.

Table 30
Comparison results of the two settings, in cross-version prediction.

project	version (Tr⇒T)	F1	AUC	Accuracy	version (Tr⇒T)	F1	AUC	Accuracy
ant	1.4⇒1.6	0.357	0.555	0.633	1.4⇒1.6	0.357	0.555	0.633
	1.6⇒1.7	0.481	**0.817**	0.549	1.4, 1.6⇒1.7	**0.528**	0.810	**0.649**
camel	1.2⇒1.4	0.134	0.213	0.166	1.2⇒1.4	0.134	0.213	0.166
	1.4⇒1.6	**0.336**	**0.251**	**0.209**	1.2, 1.4⇒1.6	0.330	0.217	0.201
jedit	3.2⇒4.0	0.485	0.741	0.594	3.2⇒4.0	0.485	0.741	0.594
	4.0⇒4.1	**0.400**	**0.619**	0.679	3.2, 4.0⇒4.1	0.383	0.509	**0.753**
lucene	2.0⇒2.2	0.790	0.834	0.751	2.0⇒2.2	0.790	0.834	0.751
	2.2⇒2.4	0.433	0.477	0.466	2.0, 2.2⇒2.4	**0.671**	**0.858**	**0.673**
poi	1.5⇒2.5	0.912	0.876	0.876	1.5⇒2.5	0.912	0.876	0.876
	2.5⇒3.0	0.755	0.720	0.656	1.5, 2.5⇒3.0	**0.872**	**0.872**	**0.831**
velocity	1.4⇒1.5	0.839	0.790	0.755	1.4⇒1.5	0.839	0.790	0.755
	1.5⇒1.6	**0.681**	**0.870**	**0.797**	1.4, 1.5⇒1.6	0.502	0.716	0.335
xalan	2.4⇒2.5	0.368	0.549	0.537	2.4⇒2.5	0.368	0.549	0.537
	2.5⇒2.6	0.532	0.532	0.431	2.4, 2.5⇒2.6	**0.575**	**0.602**	**0.443**
average		0.536	0.632	0.578		**0.553**	**0.653**	**0.586**

Table 31
Comparison results of the two settings, in cross-project prediction.

project	F1	AUC	Accuracy	project(Tr⇒T)	F1	AUC	Accuracy
ant	**0.412**	0.630	0.566	camel, jedit, lucene, poi, velocity, xalan ⇒ ant	0.308	**0.737**	**0.794**
camel	**0.438**	**0.559**	0.526	ant, jedit, lucene, poi, velocity, xalan ⇒ camel	0.009	0.491	**0.633**
jedit	0.473	**0.610**	**0.540**	ant, camel, lucene, poi, velocity, xalan ⇒ jedit	**0.497**	0.562	0.423
lucene	**0.559**	**0.593**	**0.547**	ant, camel, jedit, poi, velocity, xalan ⇒ lucene	0.398	0.207	0.265
poi	**0.561**	**0.529**	**0.515**	ant, camel, jedit, lucene, velocity, xalan ⇒ poi	0.266	0.348	0.346
velocity	**0.491**	**0.370**	**0.430**	ant, camel, jedit, lucene, poi, xalan ⇒ velocity	0.122	0.086	0.104
xalan	0.329	0.639	**0.535**	ant, camel, jedit, lucene, poi, velocity ⇒ xalan	0.365	0.772	0.479
average	**0.466**	**0.561**	**0.523**		0.281	0.457	0.435

CRediT authorship contribution statement

Chunying Zhou: Methodology, Software, Validation, Formal analysis, Investigation, Writing – original draft, Project administration. **Peng He:** Conceptualization, Data Curation, Writing – review & editing. **Cheng Zeng:** Resources, Visualization, Supervision, Funding acquisition. **Ju Ma:** Software, Validation, Writing – original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

l have shared the link to my data/code at the Manuscript

Acknowledgments

This work is supported by the National Key R&D Program of China (No. 2018YFB1003801), the National Natural Science Foundation of China (Nos. 61832014, 61902114, 61977021), and the Key R&D Program of Hubei Province, China under Grant (Nos. 2021BAA184, 2021BAA188). The authors would like to express their gratitude to EditSprings (https://www.editsprings.cn/) for the expert linguistic services provided.

References

[1] S. Feng, J. Keung, X. Yu, Y. Xiao, M. Zhang, Investigation on the stability of SMOTE-based oversampling techniques in software defect prediction, Inf. Softw. Technol. 139 (2021) 106662.
[2] Z. Xu, J. Liu, X. Luo, Z. Yang, Y. Zhang, P. Yuan, Y. Tang, T. Zhang, Software defect prediction based on kernel PCA and weighted extreme learning machine, Inf. Softw. Technol. 106 (2019) 182–200.
[3] Y. Qu, Q. Zheng, J. Chi, Y. Jin, A. He, D. Cui, H. Zhang, T. Liu, Using K-core decomposition on class dependency networks to improve bug prediction model's practical performance, IEEE Trans. Softw. Eng. 47 (2) (2019) 348–366.
[4] Y. Qu, J. Chi, H. Yin, Leveraging developer information for efficient effort-aware bug prediction, Inf. Softw. Technol. 137 (2021) 106605.
[5] J. Li, P. He, J. Zhu, M.R. Lyu, Software defect prediction via convolutional neural network, in: 2017 IEEE International Conference on Software Quality, Reliability and Security, QRS, IEEE, 2017, pp. 318–328.
[6] S. Wang, T. Liu, J. Nam, L. Tan, Deep semantic feature learning for software defect prediction, IEEE Trans. Softw. Eng. 46 (12) (2018) 1267–1293.
[7] H.K. Dam, T. Tran, T.T.M. Pham, S.W. Ng, J. Grundy, A. Ghose, Automatic feature learning for predicting vulnerable software components, IEEE Trans. Softw. Eng. (2018).
[8] W. Wang, G. Li, B. Ma, X. Xia, Z. Jin, Detecting code clones with graph neural network and flow-augmented abstract syntax tree, in: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2020, pp. 261–271.
[9] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, 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.
[10] L. Chen, W. Ma, Y. Zhou, L. Xu, Z. Wang, Z. Chen, B. Xu, Empirical analysis of network measures for predicting high severity software faults, Sci. China Inf. Sci. 59 (12) (2016) 1–18.
[11] Y. Li, Applying Social Network Analysis to Software Fault-Proneness Prediction (Ph.D. thesis), 2017.
[12] W. Ma, L. Chen, Y. Yang, Y. Zhou, B. Xu, Empirical analysis of network measures for effort-aware fault-proneness prediction, Inf. Softw. Technol. 69 (2016) 50–70.
[13] Y. Qu, H. Yin, Evaluating network embedding techniques' performances in software bug prediction, Empir. Softw. Eng. 26 (4) (2021) 1–44.
[14] Y. Qu, T. Liu, J. Chi, Y. Jin, D. Cui, A. He, Q. Zheng, node2defect: Using network embedding to improve software defect prediction, in: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2018, pp. 844–849.
[15] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, S.Y. Philip, A comprehensive survey on graph neural networks, IEEE Trans. Neural Netw. Learn. Syst. 32 (1) (2020) 4–24.
[16] C. Zeng, C.Y. Zhou, S.K. Lv, P. He, J. Huang, GCN2defect : Graph convolutional networks for smotetomek-based software defect prediction, in: 2021 IEEE 32nd International Symposium on Software Reliability Engineering, ISSRE, 2021, pp. 69–79.
[17] R. Mo, S. Wei, Q. Feng, Z. Li, An exploratory study of bug prediction at the method level, Inf. Softw. Technol. 144 (2022) 106794.
[18] X. Yu, J. Keung, Y. Xiao, S. Feng, F. Li, H. Dai, Predicting the precise number of software defects: Are we there yet? Inf. Softw. Technol. 146 (2022) 106847.
[19] K.E. Bennin, J. Keung, P. Phannachitta, A. Monden, S. Mensah, Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction, IEEE Trans. Softw. Eng. 44 (6) (2017) 534–550.

[20] X. Yu, M. Wu, Y. Jian, K.E. Bennin, M. Fu, C. Ma, Cross-company defect prediction via semi-supervised clustering-based data filtering and MSTrA-based transfer learning, Soft Comput. 22 (10) (2018) 3461–3472.
[21] Z. Xu, S. Pang, T. Zhang, X.-P. Luo, J. Liu, Y.-T. Tang, X. Yu, L. Xue, Cross project defect prediction via balanced distribution adaptation based transfer learning, J. Comput. Sci. Tech. 34 (5) (2019) 1039–1062.
[22] T. Zhou, X. Sun, X. Xia, B. Li, X. Chen, Improving defect prediction with deep forest, Inf. Softw. Technol. 114 (2019) 204–216.
[23] P. He, B. Li, X. Liu, J. Chen, Y. Ma, An empirical study on software defect prediction with a simplified metric set, Inf. Softw. Technol. 59 (2015) 170–190.
[24] K. Zhao, Z. Xu, T. Zhang, Y. Tang, M. Yan, Simplified deep forest model based just-in-time defect prediction for android mobile apps, IEEE Trans. Reliab. 70 (2) (2021) 848–859.
[25] K. Zhao, Z. Xu, M. Yan, L. Xue, W. Li, G. Catolino, A compositional model for effort-aware Just-In-Time defect prediction on android apps, IET Softw. 16 (3) (2022) 259–278.
[26] Z. Xu, L. Li, M. Yan, J. Liu, X. Luo, J. Grundy, Y. Zhang, X. Zhang, A comprehensive comparative study of clustering-based unsupervised defect prediction models, J. Syst. Softw. 172 (2021) 110862.
[27] X. Yu, J. Liu, J.W. Keung, Q. Li, K.E. Bennin, Z. Xu, J. Wang, X. Cui, Improving ranking-oriented defect prediction using a cost-sensitive ranking SVM, IEEE Trans. Reliab. 69 (1) (2019) 139–153.
[28] X. Yu, K.E. Bennin, J. Liu, J.W. Keung, X. Yin, Z. Xu, An empirical study of learning to rank techniques for effort-aware defect prediction, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2019, pp. 298–309.
[29] Z. He, F. Peters, T. Menzies, 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.
[30] A.V. Phan, M. Le Nguyen, 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.
[31] Z. Xu, S. Li, J. Xu, J. Liu, X. Luo, Y. Zhang, T. Zhang, J. Keung, Y. Tang, LDFR: Learning deep feature representation for software defect prediction, J. Syst. Softw. 158 (2019) 110402.
[32] B. Perozzi, R. Al-Rfou, S. Skiena, Deepwalk: Online learning of social representations, in: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2014, pp. 701–710.
[33] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, Q. Mei, Line: Large-scale information network embedding, in: Proceedings of the 24th International Conference on World Wide Web, 2015, pp. 1067–1077.
[34] A. Grover, J. Leskovec, node2vec: Scalable feature learning for networks, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 855–864.
[35] W. Hamilton, Z. Ying, J. Leskovec, Inductive representation learning on large graphs, Adv. Neural Inf. Process. Syst. 30 (2017).
[36] T.N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, 2016, arXiv preprint arXiv:1609.02907.
[37] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, Y. Bengio, Graph attention networks, 2017, arXiv preprint arXiv:1710.10903.
[38] C. Lin, Z. Ouyang, J. Zhuang, J. Chen, H. Li, R. Wu, Improving code summarization with block-wise abstract syntax tree splitting, in: 2021 IEEE/ACM 29th International Conference on Program Comprehension, ICPC, IEEE, 2021, pp. 184–195.
[39] B. Guo, C. Zhang, J. Liu, X. Ma, Improving text classification with weighted word embeddings via a multi-channel TextCNN model, Neurocomputing 363 (2019) 366–374.
[40] X. Ma, J. Keung, Z. Yang, X. Yu, Y. Li, H. Zhang, CASMS: Combining clustering with attention semantic model for identifying security bug reports, Inf. Softw. Technol. 147 (2022) 106906.
[41] Z. Tan, J. Chen, Q. Kang, M. Zhou, A. Abusorrah, K. Sedraoui, Dynamic embedding projection-gated convolutional neural networks for text classification, IEEE Trans. Neural Netw. Learn. Syst. (2021).
[42] C.-H.H. Yang, J. Qi, S.Y.-C. Chen, P.-Y. Chen, S.M. Siniscalchi, X. Ma, C.-H. Lee, Decentralizing feature extraction with quantum convolutional neural network for automatic speech recognition, in: ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP, IEEE, 2021, pp. 6523–6527.
[43] S. Yun, S.J. Oh, B. Heo, D. Han, J. Choe, S. Chun, Re-labeling imagenet: from single to multi-labels, from global to localized labels, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 2340–2350.
[44] P. He, B. Li, Y. Ma, L. He, Using software dependency to bug prediction, Math. Probl. Eng. 2013 (2013).
[45] P. He, P. Wang, B. Li, S.-w. Hu, An evolution analysis of software system based on multi-granularity software network, Acta Electon. Sin. 46 (2) (2018) 257.
[46] N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer, SMOTE: synthetic minority over-sampling technique, J. Artificial Intelligence Res. 16 (2002) 321–357.
[47] G.M. Weiss, Foundations of imbalanced learning, in: Imbalanced Learning: Foundations, Algorithms, and Applications, Wiley Online Library, 2013, pp. 13–41.
[48] M. Zeng, B. Zou, F. Wei, X. Liu, L. Wang, Effective prediction of three common diseases by combining SMOTE with tomek links technique for imbalanced medical data, in: 2016 IEEE International Conference of Online Analysis and Computing Science, ICOACS, IEEE, 2016, pp. 225–228.
[49] Y. Zhao, Y. Wang, Y. Zhang, D. Zhang, Y. Gong, D. Jin, ST-TLF: Cross-version defect prediction framework based transfer learning, Inf. Softw. Technol. 149 (2022) 106939.
[50] K.E. Bennin, A. Tahir, S.G. MacDonell, J. Börstler, An empirical study on the effectiveness of data resampling approaches for cross-project software defect prediction, IET Softw. 16 (2) (2022) 185–199.
[51] N. Cliff, Ordinal Methods for Behavioral Data Analysis, Psychology Press, 2014.
[52] P. Le, W. Zuidema, Quantifying the vanishing gradient and long distance dependency problem in recursive neural networks and recursive LSTMs, 2016, arXiv preprint arXiv:1603.00423.
[53] Y. Zhen, J.W. Keung, Y. Xiao, X. Yan, J. Zhi, J. Zhang, On the significance of category prediction for code-comment synchronization, ACM Trans. Softw. Eng. Methodol. (2022).
[54] F. Zhang, X. Yu, J. Keung, F. Li, Z. Xie, Z. Yang, C. Ma, Z. Zhang, Improving Stack Overflow question title generation with copying enhanced CodeBERT model and bi-modal information, Inf. Softw. Technol. 148 (2022) 106922.
[55] X. Zhu, P. Sobihani, H. Guo, Long short-term memory over recursive structures, in: International Conference on Machine Learning, PMLR, 2015, pp. 1604–1612.
[56] S. Herbold, A. Trautsch, J. Grabowski, A comparative study to benchmark cross-project defect prediction approaches, IEEE Trans. Softw. Eng. 44 (9) (2017) 811–833.