Effective Prediction of Bug-Fixing Priority via Weighted Graph Convolutional Networks

Sen Fang, You-shuai Tan, Tao Zhang, Zhou Xu, and Hui Liu

***Abstract*—With the increasing number of software bugs, bug fixing plays an important role in software development and maintenance. To improve the efficiency of bug resolution, developers utilize bug reports to resolve given bugs. Especially, bug triagers usually depend on bugs' descriptions to suggest priority levels for reported bugs. However, manual priority assignment is a time-consuming and cumbersome task. To resolve this problem, recent studies have proposed many approaches to automatically predict the priority levels for the reported bugs. Unfortunately, these approaches still face two challenges that include words' nonconsecutive semantics in bug reports and the imbalanced data. In this article, we propose a novel approach that graph convolutional networks (GCN) based on weighted loss function to perform the priority prediction for bug reports. For the first challenge, we build a heterogeneous text graph for bug reports and apply GCN to extract words' semantics in bug reports. For the second challenge, we construct a weighted loss function in the training phase. We conduct the priority prediction on four open-source projects, including Mozilla, Eclipse, Netbeans, and GNU compiler collection. Experimental results show that our method outperforms two baseline approaches in terms of the F-measure by weighted average of 13.22%.**

***Index Terms*—Bug report, graph convolutional network (GCN), priority prediction.**

I. Introduction

BUG resolution plays an important role in software maintenance. Unfortunately, due to a large number of bugs appearing in software products, bug resolution has become a time-consuming and difficult task [1]. For example, according to our investigation, there are more than 10,000 bug reports submitted to Mozilla project from January 1, 2021, to March 8, 2021, which means that Mozilla project can receive more than 150 new bug reports a day on average. To locate bugs and improve the next release, most of large-scale open-source software project teams utilize bug tracking systems (e.g., Bugzilla[1] and Jira[2]) to manage bugs [2] by submitting and analyzing bug reports. Therefore, more and more bug reports are submitted to bug tracking systems [3]. As a result, some serious bugs cannot be fixed in time due to the heavy workload and limited time [4]. To avoid this problem, bug triager usually prioritize the bug-fixing process, which can help them fix the bugs with the highest priority. Specifically, as an experienced developer, bug triager first determines if a new bug is an enhancement or a valuable new problem according to the content of its corresponding bug report. If this bug is, the bug triager will prioritize it and assign a bug fixer to resolve it. In Bugzilla, the priority is defined from P1 to P5, where P1 stands for the highest priority and P5 stands for the lowest one. Although manual priority assignment can get high-quality classification, it is tedious and requires a lot of manpower. Therefore, how to build an effective model for automated priority prediction become a big problem.

In order to overcome this problem, some automated approaches have been proposed to perform the priority prediction [2], [5], [6]. Tian *et al.* [2] proposed a linear regression model named DRONE which makes full use of factors in bug reports and utilizes thresholding approach to deal with imbalanced data. Umer *et al.* [5] used the convolutional neural network (CNN) to implement an automated priority assignment model named cPur, where CNN is adept at capturing syntactic and semantic information in local consecutive word sequences. Even though these approaches realize the bug priority prediction, they still face two challenges. First, the prediction results of DRONE are still influenced by imbalanced data because it cannot effectively predict some priority classes with small-sized samples. Second, CNN ignores global words' co-occurrence in documents which contain long-distance and nonconsecutive semantics [7].

To overcome the above drawbacks, we propose a novel framework named Ppriority prediction by using weighted graph convolutional networks (PPWGCN) to realize automated priority prediction, where graph convolutional network (GCN) for text classification was first proposed by Yao *et al.* [8]. Our approach considers five elements of bug reports that may help to recognize the priority levels (P1–P5). For each given bug, these elements include the severity of the bug (severity), the

Manuscript received November 13, 2020; revised March 8, 2021; accepted April 16, 2021. Date of publication May 19, 2021; date of current version June 1, 2021. This work was supported in part by the Science and Technology Development Fund of Macau, Macau SAR under Grant 0047/2020/A1, in part by the China Postdoctoral Science Foundation, China under Grants 2017M621247 and 2020M673137, in part by the Faculty Research Grant Projects of MUST, Macau SAR under Grant FRG-20-008-FI, and in part by the Natural Science Foundation of Heilongjiang Province under Grant LH2019F008. Associate Editor: R. Gao. *(Sen Fang and You-shuai Tan contributed equally to this work.) (Corresponding author: Tao Zhang.)*

Sen Fang, You-shuai Tan, and Tao Zhang are with the Faculty of Information Technology, Macau University of Science and Technology, Macau 999078, China (e-mail: fangsen1996@gmail.com; tanyoushuai@hrbeu.edu.cn; tazhang@must.edu.mo).

Zhou Xu is with the School of Big Data and Software Engineering, Chongqing University, Chongqing 400030, China (e-mail: zhouxullx@cqu.edu.cn).

Hui Liu is with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China (e-mail: liuhui08@bit.edu.cn).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TR.2021.3074412.

Digital Object Identifier 10.1109/TR.2021.3074412

[1][Online]. Available: https://www.bugzilla.org/

[2][Online]. Available: https://www.atlassian.com/software/jira

product which is affected by the bug (product), the component which the bug affects (component), the particular information of the bug (description), and the summary of the bug (summary). Then, we apply natural language processing (NLP) techniques to preprocess bug reports and use the preprocessed data to construct a heterogeneous graph which contains word nodes and document nodes. By completing the above steps, we transform the priority prediction problem into a node classification problem. Next, to capture global word co-occurrence information and make good priority prediction, we utilize GCN [9] to model the heterogeneous graph. The reason is that GCN can propagate the priority class-label information to the entire graph well, thus our heterogeneous graph preserves global word co-occurrence information. Since bug reports with different priority levels are quite imbalanced (see Section IV), we construct a weighted cross entropy loss function by introducing label penalty and use it to deal with the imbalanced problem in bug reports.

To evaluate the effectiveness of our approach, we choose state-of-the-art approaches-DRONE [2] and cPur [5] as baselines. We conduct experiments on four open-source repositories, including Mozilla,[3] Eclipse,[4] Netbeans,[5] and GNU compiler collection (GCC).[6] The experimental results show that our method outperforms two baselines approaches in terms of the F-measure by weighted average of 13.22%. Moreover, similar to the previous work [8], our approach can still perform well with less training data (see Section IV).

To help researchers reproduce our approach quickly, we open all data, source code, and results at GitHub[7].

Our contributions are summarized as follows.

1) To the best of our knowledge, we are the first to leverage the global words' co-occurrence in predicting the priority of bug reports.
2) A novel approach is proposed to handle the imbalance of bug reports.
3) We conduct the extensive experiments to evaluate the performance of our approach on four large-scale open-source projects, including Mozilla, Eclipse, Netbeans, and GCC. The experimental results illustrate that our approach outperforms baseline approaches.

The remainder of this article is as follows. Section II describes the background and our motivations. Section III introduces the details of our approach and uses a bug report as an example. Section IV shows how to perform experiments and analyzes the results. In Section V, we discuss several threats to our approach. We present some related works in Section VI. In Section VII, we conclude the article and discuss future work.

II. Background and Motivations

In this section, we introduce background knowledge, including bug reporting, priority prediction, and GCN. After that, we explain the motivation of our work.

[3][Online]. Available: https://bugzilla.mozilla.org/
[4][Online]. Available: https://bugs.eclipse.org/
[5][Online]. Available: https://bz.apache.org/netbeans/
[6][Online]. Available: https://gcc.gnu.org/
[7][Online]. Available: https://github.com/TanYoushuai123/PPWGCN

TABLE I
Main Elements in Bugzilla Bug Reports

Element	Description
Summary	It succinctly describes what a reported bug is.
Product	A specific project where the bug appears.
Reporter	The person who reported the bug.
Component	A second-level category indicates a specific component where the bug appears; it belongs to the Product.
Priority	It indicates that how soon the bug should be fixed.
Severity	It indicates that how severe the bug is, or whether it is an enhancement.
Assignee	The person in charge of resolving the bug.
Comment	Developers can free post their suggestions how to fix the reported bug.
Description	It describes the reported bug in detail (*e.g.*, how is the bug produced?).

A. Bug Reports

Bug tracking systems are established for developers to get the feedback which helps to fix bugs [10]. Although both users and developers utilize these systems to submit their feedback in form of the bug report, different bug tracking systems have their own ways to define the priority for the bug report [11]. Thus, to maintain the consistency, we only focus on bug reports which are managed by Bugzilla, because it is one of the most popular bug tracking systems.

To be more visualized, Fig. 1 illustrates an eclipse bug report which contains basic elements, such as description, summary, comment, product, component, priority, severity, assignee, and reporter. We describe the main elements of bug reports extracted from Bugzilla in Table I. We also explain the basic concept of each one in this table helping readers easily understand the contents of bug reports. Especially, only name information is included in "Reporter" and "Assignee" elements in the bug report. Considering that we use four open-source projects as datasets in the experiment, names in these two elements are full of randomness. Thus, it is difficult to extract meaningful semantic information from them. Besides, the information in "comment" element in the bug report is not always related to this bug, such as the example in Table 1. Therefore, we use the remaining five elements ("Summary," "Product," "Component," "Severity," and "Description") of the bug report to achieve bug report priority classification, all of which have sufficient text information and are helpful to train PPWGCN.

B. Priority Prediction

When a reporter submits a bug report, developers work together to fix the submitted bug. First of all, a bug triager determines whether the new bug is an enhancement or a new

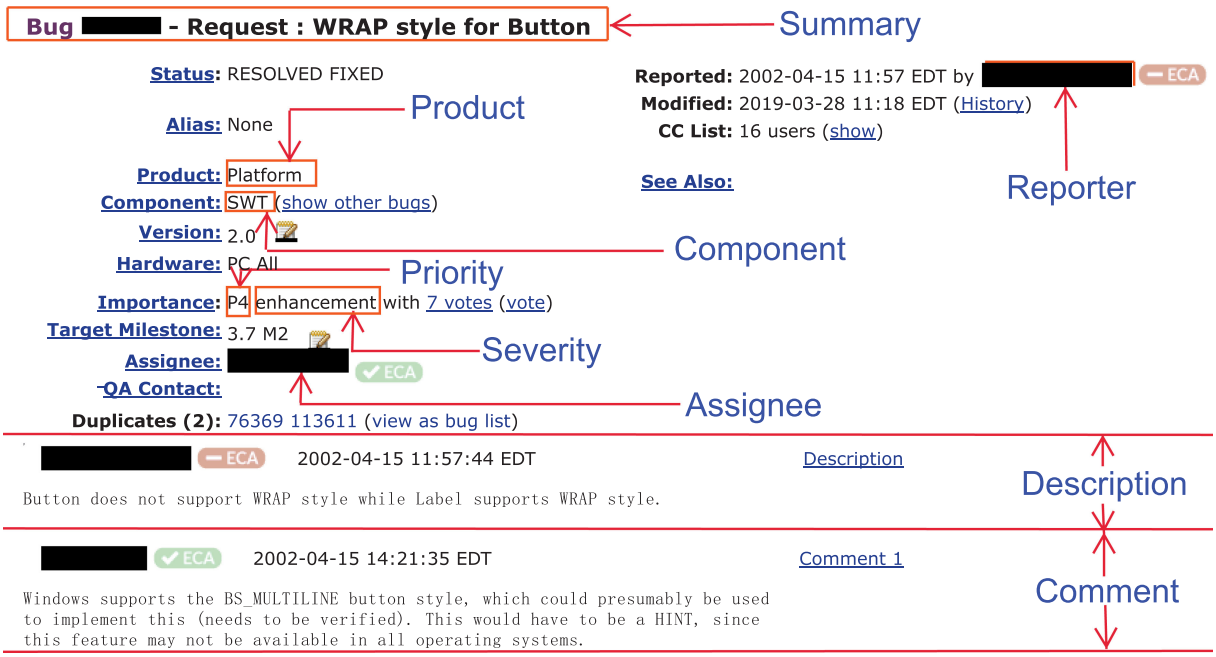

Fig. 1. Example of Eclipse bug report. We pixelate sensitive information such as bug report id and name.

problem that should be resolved. Then the given bug is assigned to a developer to resolve it. However, developers are always overwhelmed with massive bug reports [3]. If all the bug reports have the same priority, the bug-fixing process for several serious problems may be postponed. Thus, developers prioritize the bug reports to raise efficiency. In Bugzilla, priority labels are classified as P1, P2, P3, P4, and P5, where P1 represents the highest priority while P5 stands for the lowest one. For example, Fig. 1 describes an issue that the button does not support WRAP. Obviously the developer (i.e., the triager) thinks that this is a minor problem so that the priority level is set to P4. Thus, the reported issue will be resolved later than other more serious bugs with higher priority level, i.e., P1, P2, and P3.

C. Graph Convolutional Networks

Many scholars have paid attention to GCN recently [12]–[14]. It is difficult to generalize well-established neural models (e.g., RNN, CNN) on arbitrarily structured graphs. To address this problem, several papers have constructed parameterized filters using graph convolutions from spectral graph theory to perform in a neural network [9], [15], [16]. Kipf and Welling [9] proposed a novel graph neural network model based on framework of spectral graph convolutions, called GCN. This model is simple and allows for high predictive accuracy and fast training, which achieves state-of-the-art classification results on many benchmark graph datasets. Moreover, GCN has been applied to NLP tasks, such as machine translation [17], relation classification [18], and semantic role labeling [19].

Can GCN be used for text classification? Yao *et al.* [8] proposed a novel model for text classification called text GCN. They utilized the corpus to build a heterogeneous graph and adopt GCN to perform a nodes classification, which outperforms a lot of baseline approaches.

D. Motivation

As the number of bug reports increases, this may overwhelm developers [3]. Thus, a bug report with a clear priority class-label can help developers to solve the corresponding bug quickly [2]. However, it is boring and bored for developers to manually label the priority for each bug report. Therefore, several approaches have been proposed to automatically predict the priority of bug reports [2], [5], [6]. All these models can only achieve good accuracy for the priority class with large-sized samples. But for the priority class with small-sized samples, they almost cannot make accurate predictions [2]. Except for the imbalance of bug reports' distribution, another reason is that these approaches pay more attention to the semantic information in the local consecutive words of the bug report, but may ignore the long-distance and nonconsecutive semantic information from global word co-occurrence. In the text classification task, a crucial step is the text representation [8]. Hence, in the bug report priority classification, a sufficient text representation for the bug report can dramatically improve the performance of the model. Compared to local semantic information, global semantic information can help models to understand a bug report completely, which is beneficial for models to learn the semantic representation of the bug report [20]. Besides, lacking of the global semantic information makes models cannot accurately recognize the priority class-labels that have a small-sized samples in many bug reports [8]. Therefore, to construct a more effective model to perform the priority prediction of bug reports is meaningful.

Driven by the above motivation and the success of GCN in text classification, there is an idea that occurred in our mind—is GCN suitable for the priority classification of bug reports? In our view, we can regard each bug report as a document and apply GCN to realize the priority classification for each document. Compared with previous model [2], [5], [6], GCN can extract

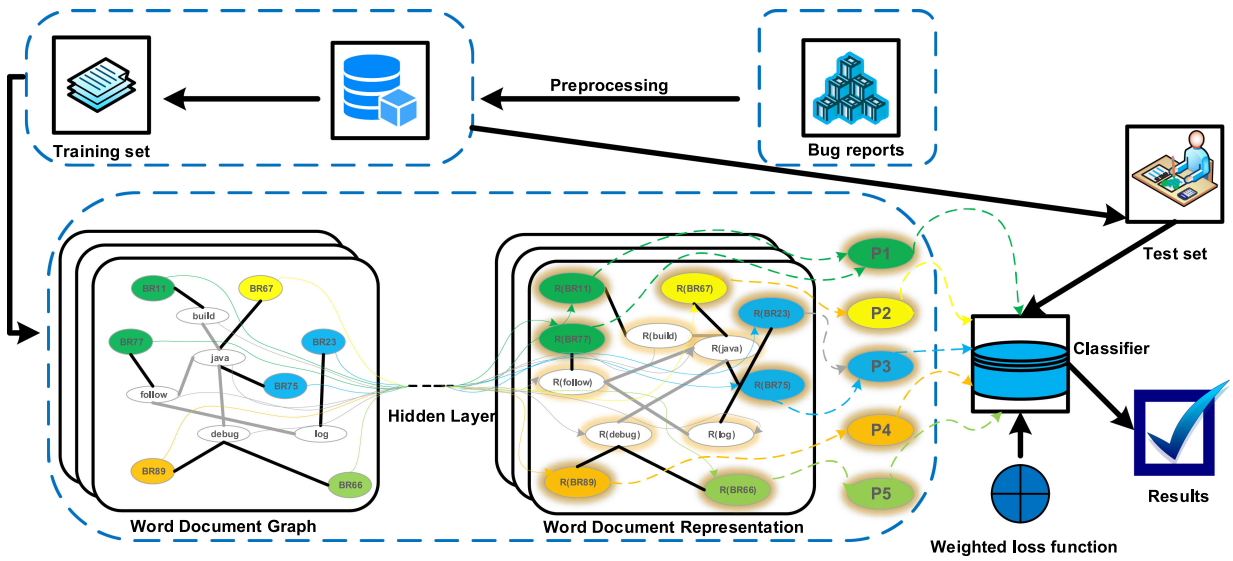

Fig. 2. Schematic of PPWGCN. Nodes begin with "BR" are bug reports while others are words. Gray straight lines are word-word edges and black straight lines are word-document edges. $R(x)$ represents the word embedding or document embedding for x (depending on x is a word or document). Curves in different colors mean the message flows in the GCN.

more sufficient global semantic information because it models all of the words and documents in a heterogeneous graph. As we all know, global semantic information is helpful for the model to understand the deep meaning of documents [21], [22]. Furthermore, for the imbalanced data, we think this problem can be alleviated by introducing the label penalty since it gives a relatively small weight to the priority class-labels with large-sized samples.

In this article, we actively explore how to use GCN to the priority classification of bug reports. We propose our approach and conduct a series of experiments to verify the effectiveness of our proposed model.

III. Methodology

In this section, we describe the details of the proposed novel framework PPWGCN for predicting the priority levels of bug reports. We present here an overall framework of PPWGCN, and we also describe it step by step.

A. Overview

To improve the performance of automated priority prediction, we present a novel approach PPWGCN to conduct this work. Fig. 2 shows the framework of our approach. The brief introduction for each step of this approach is as follows.

1) We take the five elements of a bug report as a document and employ the preprocessing method to reduce memory consumption and improve the performance of the proposed algorithm, which includes rare words removal and stop word removal.

2) We build a heterogeneous graph whose nodes represent both documents and words. We utilize pointwise mutual information (PMI) to calculate the word-word edges and TF-IDF to get the word-document edges.

3) We feed this graph into a GCN to train a classifier with the supervision of a weighted loss function. When completing training process, we use this trained model to predict the priority levels of the bug reports in our test dataset.

In the following sections, we describe the details of each step in this framework. For easy to understand, we also take the bug report shown in Fig. 1 as an example.

B. Preprocessing

We preprocess bug reports from four open-source projects (i.e., Mozilla, Eclipse, Netbeans, and GCC) to verify the efficiency of the proposed algorithm.

We describe the process by introducing the case. In Fig. 1, notes that the bug report contains all five elements (i.e., description, summary, severity, component, and product). We first combine them into one document, then adopt a preprocess to clean it. We get our raw data as—*button, does, not, support, wrap, style, while, label, supports, wrap, style, request, wrap, style, for, button, enhancement, swt, platform.*

Then, we employ stop word removal to remove some words (e.g., not, are, for) that nearly cannot make any contribution to the performance of the algorithm. Moreover, rare words may reduce the accuracy and efficiency of models in NLP domain [23], [24]. Thus, we remove the word whose occurrence frequency is less than five. Finally, the abovementioned raw data are changed to the following one—*button, support, wrap, style, label, supports, wrap, style, request, wrap, style, button, enhancement, swt, platform.*

C. Text Graph Convolutional Network for Priority Prediction

We first build a heterogeneous graph $G = (V, E)$ after preprocessing bug reports, where V and E represent nodes and edges, respectively. We take both words and documents (i.e., bug reports) as nodes. Thus, the number of nodes contains two parts: one is the sum of the number of documents in the corpus (i.e., open-source project), and another one is the size

of the vocabulary [i.e., the number of unique words (UWs)]. In our heterogeneous graph, there are three different types of edges, including document–document edge, word-word edge, and word-document edge. For document-document edges, we generally set the corresponding values to zero. For word-word edges, we utilize a sliding window on every document to gather the word frequency of co-occurrence, which captures the global word co-occurrence information. PMI is a useful measure for word associations, thus, we use it to calculate weights of word-word edges. The PMI value of two words *(i, j)* is calculated as

$$\text{PMI}(i,j) = \log \frac{p(i,j)}{p(i)p(j)} \tag{1}$$

$$p(i,j) = \frac{\#W(i,j)}{\#W} \tag{2}$$

$$p(i) = \frac{\#W(i)}{\#W} \tag{3}$$

where $\#W(i,j)$ stands for the number of specific sliding windows that contain both i and j, $\#W$ is the total number of specific windows in all the documents, and $\#W(i)$ is the number of specific sliding windows in all the documents which contain the word i. Positive PMI values indicate strong semantic corrections of words in all documents, while negative PMI values imply weak corrections. For word-document edges, we use TF-IDF to compute the relevancy between a word and a document. We define it as follows:

$$\text{TF-IDF}_{i,j} = tf_{i,j} \times \left(1 + \log \frac{N}{1 + df(j)}\right) \tag{4}$$

where i is a document and j is a word, $tf_{i,j}$ is the number of times that j appears in i, IDF is logarithmically scaled inverse fraction of the total number of documents that contain j. Formally, we can define the edges between two nodes as

$$A_{i,j} = \begin{cases} \text{PMI}(i,j) & i,j \text{ are words, } \text{PMI}(i,j) > 0 \\ \text{TF-IDF}_{i,j} & i \text{ is document, } j \text{ is word} \\ 1 & i = j \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

After building this heterogeneous graph, we feed this graph into a simple classifier model with two GCN layers

$$Z = softmax\left(\tilde{A}\, ReLU\left(\tilde{A}XW_0\right)W_1\right) \tag{6}$$

where $\tilde{A} \in \mathbb{R}^{n\times n} = D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ is the normalized symmetric adjacency matrix, which is used to represents the relationship between two nodes. Thus, we utilize $\tilde{A}$ for propagation in our GCN framework. $A \in \mathbb{R}^{n\times n}$ is an adjacency matrix. To capture self-information, the diagonal elements of A are defined as 1, which explains that the condition of $i = j$. $D \in \mathbb{R}^{n\times n}$ is a degree matrix, where $D_{ii} = \sum_j A_{ij}$. $Z \in \mathbb{R}^{n\times m_2}$ is the prediction made by our model; n is the total number of words and documents; m_2 is the number of classes (i.e., five priority levels). In contrast to single-layer GCN, two-layer GCN enables information to pass among nodes that have a maximum distance of two steps. Therefore, the information can swap between documents though we do not build document–document edges directly [8]. For the first layer

$$L^{(1)} = ReLU\left(\tilde{A}XW_0\right) \tag{7}$$

where $ReLU$ is an activation function which is defined as

$$ReLU(x) = \max(0, x). \tag{8}$$

$X \in \mathbb{R}^{n\times n}$ is our feature matrix which is an identity matrix. Our input is a one-hot vector containing all words and documents. $W^0 \in \mathbb{R}^{n\times m_1}$ is a weight matrix, where m_1 is the first hidden size. For the second layer

$$L^{(2)} = softmax\left(\tilde{A}L^{(1)}W^1\right) \tag{9}$$

where $W^1 \in \mathbb{R}^{m_1\times m_2}$ is the model parameter. The softmax is defined as

$$softmax\,(x_i) = \frac{exp\,(x_i)}{\mathcal{Z}} \tag{10}$$

$$\mathcal{Z} = \sum_i exp\,(x_i) \tag{11}$$

where x_i is one of the five output values (i.e., P1, P2, P3, P4, and P5). To deal with the imbalanced data, we adjust the loss function by introducing label penalty. Our loss function is defined as

$$\mathcal{L} = -\sum_{d\in\mathcal{Y}_D}\sum_{f=1}^{F} Y_{df} \ln Z_{df} W_f \tag{12}$$

where $\mathcal{Y}_D$ is the training dataset, F is the number of priority labels (which is equal to 5), Y is a label indicator matrix, and W_f is weight of this label. The weight parameter is defined as

$$W_f = \begin{cases} 0 & N_f = 0 \\ \frac{\max N_i}{N_f} & N_f \neq 0, i \subseteq F \end{cases} \tag{13}$$

where N_f is number of the label f in this data.

Although document and document have no direct edges, the message of the bug report can be passed to any other bug reports after two propagations, which means that this two-layer GCN can effectively make the features of vertices in the same priority label similar and keep simple. Therefore, we believe that our approach is effective for priority prediction of bug reports.

IV. Experiment Setup

In this section, we describe the experimental setup in detail.

A. Research Questions (RQs)

In this paper, we utilize two-layer GCN to implement the priority prediction. We answer three RQs to verify the effectiveness of our approach.

RQ1: How effective is PPWGCN as compared with two baseline methods—namely, DRONE and cPur?

Tian *et al.* [2] and Umer *et al.* [5] proposed competitive DRONE and cPur for the priority prediction of bug report, respectively. Therefore, to verify the effectiveness of our proposed model, we choose DRONE and cPur as baselines to compare with our proposed model.

RQ2: How effective is PPWGCN when we use different ratio of the training dataset to train?

Yao *et al.* [8] found GCN can perform quite well with a low ratio of the training dataset. For the priority prediction, there are many bug reports without corresponding labels. Thus, this characteristic of GCN can make contributions to the priority prediction. The answer to this research question can verify whether PPWGCN is equally effective when giving it a few bug reports to train.

RQ3: Can our weighted loss function cope with the imbalanced data?

Our data is quite imbalanced, which is the same as the experimental data in other research works [2], [5]. Imbalanced data present an obstacle to experiments [25]. Specifically, the model tends to learn the features of the priority classes that have large-sized samples [26], thus, the prediction accuracy of these priority classes may be nearly 100%. Meanwhile, the accuracy of the priority classes with small-sized samples may be close to 0. To alleviate the influence of the imbalanced data, we build a weighted loss function by introducing label penalty to deal with this problem. The answer to this research question verifies whether the weighted loss function is helpful for our approach.

B. Experiment Setting

1) Dataset: To demonstrate the effectiveness of our approach, we perform a series of experiments on bug reports from four large-scale open-source projects, including Mozilla, Eclipse, Netbeans, and GCC. We collect the bug reports from February 2000 to September 2020, and treat the bug reports with the same priority level as a group. However, we found that the data are quite imbalanced, as shown in Table II. For example, the number of P3 is 90,026 in Eclipse while P4 only has 2828 entities. Besides, in Table III, we show some other information about the data we collected, including the number of UWs and the average length of the bug reports (BRs). We select 50% of data as our training dataset and the other 50% for testing, which is the same as DRONE. We split our dataset 30 times to alleviate the impact of random partition.

2) Evaluation Measures: To measure the accuracy of PPWGCN and analyze results of our experiments, we choose the F-measure (F) as our evaluation measures [27]. Precision (P), recall (R), and F-measure for a priority P_i can be defined as

$$P(P_i) = \frac{TP}{TP + FP} \tag{14}$$

$$R(P_i) = \frac{TP}{TP + FN} \tag{15}$$

$$F(P_i) = \frac{2 * P * R}{P + R} \tag{16}$$

where TP (true positive) is the number of bug reports with class-label P_i which are predicted correctly, FP (false positive) means the number of bug reports with other class-labels which are predicted as class-label P_i, FN (false negative) is the number of bug reports with class-label P_i which are predicted as other class-labels.

3) Parameter Settings: For GCN in PPWGCN, we set the embedding size of convolution layer as 200 and set the window size as 20. Besides, we set the learning rate as 0.02 and dropout rate as 0.5. We train our PPWGCN for a maximum of 200 epochs. The reason is that Yao *et al.* [8] experimented with other settings and found that small changes did not improve the results much. Moreover, through comparative experiments on five different types of datasets, Yao *et al.* [8] also found that the above settings for these two parameters (learning rate and dropout rate) are more suitable. Thus, we follow their settings.

TABLE II
DATASET (BR: BUG REPORT). BOLD ENTITIES DENOTE THE PRIORITY LABEL WITH THE MOST BRS

Project	Priority	Number of BRs
Mozilla	P1	**30,247**
	P2	9,818
	P3	13,000
	P4	4,974
	P5	3,771
Total		61,810
Eclipse	P1	4,850
	P2	9,964
	P3	**90,026**
	P4	2,828
	P5	3,270
Total		110,938
Netbeans	P1	11,946
	P2	**18,934**
	P3	18,399
	P4	2,873
	P5	0
Total		52,152
GCC	P1	3,905
	P2	12,418
	P3	**16,164**
	P4	384
	P5	241
Total		33,112

TABLE III
STATISTICS OF BUG REPORTS IN EACH PROJECT

Project	Number of UWs	Average Length of BRs
Mozilla	68,314	177.59
Eclipse	56,742	168.82
Netbeans	47,928	356.03
GCC	38,096	247.05

V. EXPERIMENTAL RESULTS

A. Answer to RQ1: Performance Comparison With Baselines

Table IV shows the results of performance comparison. Especially, WAvg denotes weighted average, where the weight is calculated by the number of each priority label divides by the total number of the priority label.

We analyze the results as follows.

TABLE IV
PERFORMANCE COMPARISON WITH BASELINES. BOLD ENTITIES DENOTE THE BEST WEIGHTED AVERAGE F SCORE

Project	Priority	Our approach			DRONE			cPur		
		P%	R(%)	F(%)	P%	R(%)	F(%)	P%	R(%)	F(%)
Mozilla	P1	76.22	59.15	66.59	100	51.12	67.65	48.94	100	65.71
	P2	30.82	37.75	33.84	21.77	25.66	23.56	0	0	0
	P3	56.19	64.85	60.17	16.99	31.47	22.06	0	0	0
	P4	37.00	53.26	43.62	5.38	34.30	9.31	0	0	0
	P5	61.27	66.01	63.52	100	1.11	2.19	0	0	0
	WAvg	60.73	56.89	**58.00**	62.50	38.54	42.42	23.95	48.94	32.16
Eclipse	P1	16.70	34.82	22.49	0	0	0	0	0	0
	P2	19.34	32.06	24.08	0.55	1.90	0.86	0	0	0
	P3	88.89	67.33	76.58	81.45	45.88	58.70	81.15	100	89.59
	P4	10.85	29.94	15.87	0.34	0.61	0.44	0	0	0
	P5	15.10	37.59	21.47	0.03	3.51	0.07	0	0	0
	WAvg	75.32	60.91	66.33	66.16	37.52	47.73	65.85	81.15	**72.07**
Netbeans	P1	47.98	50.23	49.02	42.59	70.21	53.02	0	0	0
	P2	51.12	46.67	48.73	95.46	72.35	82.32	40.54	80.61	53.95
	P3	62.52	54.86	58.40	0	0	0	59.23	46.71	52.23
	P4	18.79	40.21	25.55	0	0	0	0	0	0
	P5	0	0	0	0	0	0	0	0	0
	WAvg	52.62	50.02	**50.93**	44.41	42.35	42.03	35.61	45.74	38.01
GCC	P1	31.31	43.62	36.41	59.49	98.85	74.28	0	0	0
	P2	61.88	57.79	59.73	0	0	0	0	0	0
	P3	68.99	61.30	64.87	95.49	96.12	95.81	48.81	100	65.60
	P4	10.65	29.13	15.51	0	0	0	0	0	0
	P5	6.72	17.41	9.64	0	0	0	0	0	0
	WAvg	60.75	57.21	**58.61**	53.63	58.58	55.53	23.83	48.82	32.02

1) For Mozilla, we obtain the F-measure values of 66.59%, 33.84%, 60.17%, 43.62%, and 63.52% for P1, P2, P3, P4, and P5 priority labels, respectively, and the weighted average F-measure value is 58.00%. The F-measure value is highest for priority label P1 while it is lowest for priority label P2.

By comparing with the F-measure values of DRONE, we find that DRONE performs better when predicting the priority labels P1. Our approach can improve the weighted average F-measure value by 15.58%.

By comparing with the F-measure values of cPur, we find that cPur can only predict the priority labels P1. One major reason is that the multi-classification task is challengeable and the imbalanced data make the neural networks learn to predict priority class labels that have large-sized samples. Our approach not only can improve the weighted average F-measure value by 25.84% but can make predictions for each priority label because our approach can fully capture document-word relations, global word-word relations, and document-document relations.

Thus, our approach performs better than DRONE and cPur for Mozilla.

2) For Eclipse, we obtain the F-measure values of 22.49%, 24.08%, 76.58%, 15.87%, and 21.47% for P1, P2, P3, P4, and P5 priority labels, respectively, and the weighted average F-measure value is 66.33%. The F-measure value is highest for priority label P3 while it is lowest for priority label P4.

By comparing with the F-measure values of DRONE, we find that our approach can predict all the five priority labels while DRONE only predicts the priority labels P2, P3, P4, and P5. Our approach can improve the weighted average F-measure value by 18.60%.

By comparing with F-measure values of cPur, we find that cPur can only predict the priority label P3. Although the weighted average F-measure value of our approach is a little lower than cPur, our approach can make predictions for each priority label, which is more valuable in practice.

Therefore, our approach performs better than DRONE and cPur for Eclipse.

3) For Netbeans, we obtain the F-measure values of 49.02%, 48.73%, 58.40%, 25.55%, and 0% for P1, P2, P3, P4, and P5 priority labels, respectively, and the average F-measure value is 36.34%. The F-measure value is the highest for priority label P3 while it is the lowest for priority label P5.

By comparing with the F-measure values of DRONE, we find that our approach can predict four priority labels while DRONE only predicts the priority labels P1 and P2. Our approach can improve the weighted average F-measure value by 8.90%.

By comparing with F-measure values of cPur, we find that cPur can predict only two priority labels P2 and P3. Our approach can improve the weighted average F-measure value by 12.92%.

Thus, our approach performs better than DRONE and cPur for Netbeans.

4) For GCC, we obtain the F-measure values of 36.41%, 59.73%, 64.87%, 15.51%, and 9.64% for P1, P2, P3, P4, and P5 priority labels, respectively, and the average F-measure value is

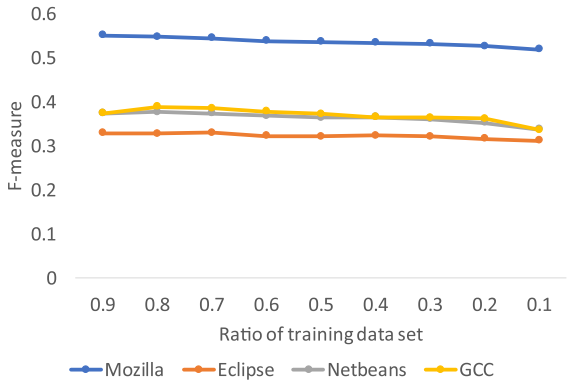

Fig. 3. Vary ratio of the training dataset on the F-measure values for Mozilla, Eclipse, and GCC.

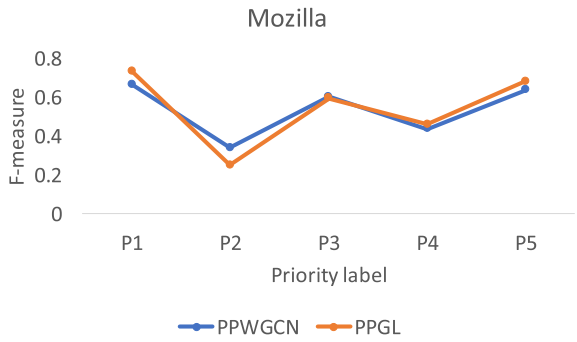

Fig. 4. Mozilla: Performance comparison of using weighted loss function and general loss function.

37.23%. The F-measure value is the highest for priority label P3 while it is the lowest for priority label P5.

By comparing with the F-measure values of DRONE, we find that our approach can predict all five priority labels while DRONE only predicts the priority labels P1 and P3. Our approach can improve the weighted average F-measure value by 3.08%.

By comparing with the F-measure values of cPur, we find that cPur can predict only the priority label P3. Our approach can improve the weighted average F-measure value by 26.59%.

Therefore, our approach performs better than DRONE and cPur for GCC.

Compared to baselines, our approach considers five elements of bug reports and can capture global co-occurrence information by building a heterogeneous graph. Meanwhile, GCN can effectively make the features of vertices in the same priority label similar and implements classification as a special Laplacian smoothing, which mixes the features of a vertex and its nearby neighbors [28]. Moreover, we utilize the weighted loss function to handle the imbalanced data. Therefore, our approach can obtain the relative balanced prediction results while keeping a high F-measure for the class-label with large-sized samples.

According to the above-mentioned experimental results, we answer to RQ1 as follows:

Answer to RQ1: Our approach is more effective than DRONE and cPur.

B. Answer to RQ2: Effectiveness Evaluation

To answer RQ2, we employ our approach when utilizing different ratios of the training dataset (ratios = 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, and 0.1). We show the results of the priority prediction for Mozilla, Eclipse, Netbeans, and GCC datasets in Fig. 3.

We analyze the results as follows.

1) For Mozilla, we obtain average F-measure values of 55.06%, 54.71%, 54.38%, 53.74%, 53.55%, 53.35%, 53.13%, 52.60%, and 51.86% for different ratios of the training dataset. We find that the average F-measure value is decreased as we decrease the ratio of training dataset. As a comparison, the average F-measure values of two baselines are 24.95% and 13.14%. Our approach with 10% ratio of the training dataset can still improve average F-measure values by 26.91% and 38.72%.

2) For Eclipse, we obtain average F-measure values of 32.82%, 32.68%, 32.91%, 32.18%, 32.10%, 32.28%, 32.06%, 31.54%, and 31.07% for different ratios of the training dataset. Generally, the average F-measure value is decreased as we decrease the ratio of training dataset. By contrast, the average F-measure values of two baselines are 12.01% and 17.92%. Our approach with 10% ratio of the training dataset can still improve average F-measures by 19.06% and 13.15%.

3) For Netbeans, we obtain average F-measure values of 37.34%, 37.59%, 37.35%, 36.75%, 36.34%, 36.44%, 36.04%, 35.20%, and 33.67% for different ratios of the training dataset. In general, the tendency of the F-measure on Netbeans is same with Mozilla and Eclipse. As a comparison, the average F-measure values of two baselines are 27.07% and 21.24%. Our approach with 10% ratio of the training dataset can still improve average F-measure values by 6.60% and 12.43%.

4) For GCC, we obtain average F-measure values of 37.33%, 38.77%, 38.51%, 37.74%, 37.23%, 36.50%, 36.37%, 36.10%, and 33.52% for different ratios of the training dataset. For these four datasets, we observe that there are same tendencies of the F-measure value as we decrease the ratio of training dataset. By contrast, the average F-measure values of two baselines are 34.02% and 13.12%. Our approach with 20% ratio of the training dataset can still improve average F-measure values by 2.08% and 22.98%.

Since our approach builds a graph that contains all bug reports, it could fully capture the global word-word relations. Thus, our approach can still perform well with a low ratio of the training dataset. According to the above analysis, we answer to RQ2 as follows:

Answer to RQ2: Although the average F-measure value is decreased as we decrease the ratio of the training dataset, PPWGCN can still perform well with a low ratio of the training dataset.

C. Answer to RQ3: Performance Analysis

To answer RQ3, we perform extra experiments that use general loss function and compare their performance with our proposed approach. We show the comparison results in Figs. 4–7.

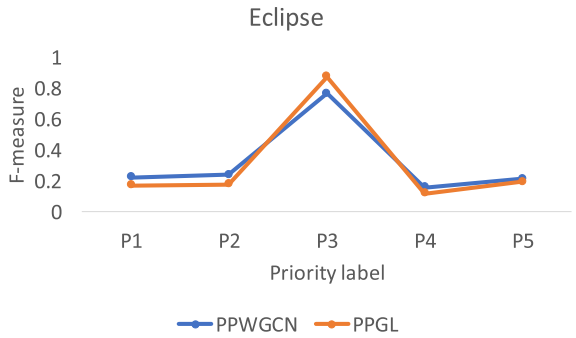

Fig. 5. Eclipse: Performance comparison of using weighted loss function and general loss function.

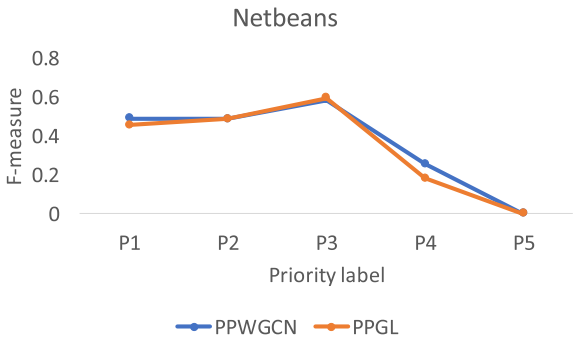

Fig. 6. Netbeans: Performance comparison of using weighted loss function and general loss function.

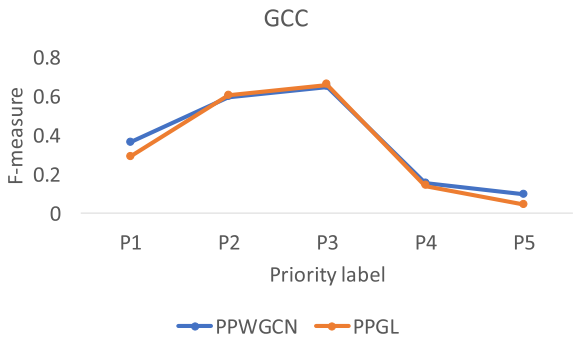

Fig. 7. GCC: Performance comparison of using weighted loss function and general loss function.

We name the method that uses common loss function with PPGL (i.e., the priority prediction with general loss function).

We analyze the experimental results as follows.

1) For Mozilla, PPGL obtains the F-measure values of 73.45%, 24.97%, 59.54%, 46.14%, and 68.16% for P1, P2, P3, P4, and P5 priority labels, respectively. The F-measure value is highest for priority label P1 and while it is lowest for priority label P2. The average F-measure value is 54.45%.

We use variance score to demonstrate whether the weighted loss function can be helpful to our approach. PPWGCN obtains the F-measure values of 66.59%, 33.84%, 60.17%, 43.62%, and 63.52% for P1, P2, P3, P4, and P5 priority labels, respectively. The variance score of PPWGCN and PPGL are 0.0200 and 0.0378, respectively, and thus, PPWGCN can improve the variance score by 0.0178.

2) For Eclipse, PPGL obtains the F-measure values of 17.05%, 17.94%, 87.51%, 11.76%, and 19.5% for P1, P2, P3, P4, and P5 priority labels, respectively. The F-measure value is highest for priority label P3 and while it is lowest for priority label P4. The average F-measure value is 30.75%.

We use variance score to demonstrate whether the weighted loss function can be helpful to our approach. PPWGCN obtains the F-measure values of 22.49%, 24.08%, 76.58%, 15.87%, and 21.47% for P1, P2, P3, P4, and P5 priority labels, respectively. The variance score of PPWGCN and PPGL are 0.0628 and 0.1015, respectively, and thus, PPWGCN can improve the variance score by 0.0387.

3) For Netbeans, PPGL obtains the F-measure values of 45.56%, 48.64%, 59.30%, 18.21%, and 0% for P1, P2, P3, P4, and P5 priority labels, respectively. The F-measure value is highest for priority label P3 and while it is lowest for priority label P5. The average F-measure value is 34.34%.

We use variance score to demonstrate whether the weighted loss function can be helpful to our approach. PPWGCN obtains the F-measure values of 49.02%, 48.73%, 58.40%, 25.55%, and 0% for P1, P2, P3, P4, and P5 priority labels, respectively. The variance score of PPWGCN and PPGL are 0.0559 and 0.0598, respectively, and thus, PPWGCN can improve the variance score by 0.0039.

4) For GCC, PPGL obtains the F-measure values of 28.93%, 60.38%, 65.92%, 14.06%, and 4.47% for P1, P2, P3, P4, and P5 priority labels, respectively. The F-measure value is highest for priority label P3 and while it is lowest for priority label P5. The average F-measure value is 34.75%.

We use variance score to verify whether the weighted loss function can be helpful to our approach. PPWGCN obtains the F-measure values of 36.41%, 59.73%, 64.87%, 15.51%, and 9.64% for P1, P2, P3, P4, and P5 priority labels, respectively. The variance score of PPWGCN and PPGL are 0.0626 and 0.0752, respectively, and thus, PPWGCN can improve the variance score by 0.0126.

Moreover, PPWGCN can improve the average F-measure value by 0.25%, 2.00%, and 2.48% for Eclipse, Netbeans, and GCC, respectively. PPGL only improve the average F-measure value by 0.9% for Mozilla. The reason is that we set weights for different labels when they implement the backpropagation, thus, the model reduces the attention to priority class with the large-sized samples. According to the analysis above, we answer to RQ3 as follows:

Answer to RQ3: The variance scores show that the weighted loss function could relieve the class imbalanced problem of our datasets.

VI. Threats to Validity

In this section, we discuss several possible threats to our approach, which includes external ones and internal ones, we explain as follows. Moreover, several bug reports from open-source projects are with poor quality and priority class-labels may be invalid.

1) External threats: We collect bug reports from four open-source bug repositories to employ our experiments. But we cannot ensure that our approach is still effective in other large-scale open-source projects and commercial projects. Moreover, we also only used bug reports managed by Bugzilla to conduct our experiments. Since different bug tracking systems have various life cycles [29] of bugs, we are not sure whether our approach is still effective in bug reports managed by other bug tracking systems. Another external threat is that several bug reports from open-source projects are of poor quality and priority class-labels, thus they may be invalid. In the future, we will implement our approach for the open-source projects managed by other bug tracking systems in order to demonstrate the effectiveness of the proposed approach.
2) Internal threats: We utilize description, summary, severity, component, and product of bug reports to build our dataset and use them to train our model. However, we do not consider developers' characteristics which may affect the quality of priority labels.

VII. Related Work

In this section, we introduce the previous studies related to the priority prediction of bug reports. Then, we introduce the some tasks related to the priority prediction such as severity prediction of bug reports and other bug management tasks such as bug localization, bug summarization, and duplicate bug report detection.

A. Priority Prediction

Abdelmoez *et al.* [30] utilized Naïve Bayesian and considered their mean time to realize the priority prediction. They employed their approach on three large open-source projects, including Mozilla, Eclipse, and GNOME.

Tian *et al.* [2] proposed a linear regression model named DRONE. They made good use of features in bug reports and utilized threading approach to handle imbalanced data, thus DRONE achieves the average F-measure to 29%.

Alenezi and Banitaan [31] performed Naïve Bayesian, random forest, and decision tree on the priority prediction, respectively. They considered two features, including term frequency weighted words of bug reports and classification of bug report attributes, where the results show that the second feature performs better than the first one. Decision trees and random forests are better than Naïve Bayesian.

Umer et al. [5] used a CNN-based model to predict the labels of bug reports. They first converted each word into a vector using a word2vec model. Then, they passed both the vectors and emotions of bug reports to the CNN-based classifier.

Our work is different from previous studies. We consider five elements of bug reports and develop a novel framework based on a weighted GCN.

There are some research studies helpful for the bug report priority prediction, such as software fault-proneness prediction. For example, Li *et al.* [32] integrate developer distribution relation, module dependency relation, and developer collaboration relation, and then they used a trirelation network to achieve software fault-proneness prediction. We find that, distinguishing modules that are prone to failure is helpful for us to predict bug report priority because these modules may have high priority (worth further studying in the future).

B. Severity Prediction

The severity prediction is different from the priority prediction. Users assign the severity class-label of a bug report while developers provide the priority labels [2]. To the best of our knowledge, more and more researchers pay attention to severity prediction.

Menzies and Marcus [33] were the first to predict the severity labels of bug reports. They performed their approach on bug reports from NASA. They first converted descriptions of the bug reports into tokens. Then they preprocessed these tokens by performing stemming and removing stop words. The important tokens from training data were chosen to feed into an algorithm called RIPPER [34].

Lamkanfi *et al.* [35] first performed severity prediction on open-source repositories, which extended the work of Menzies and Marcus. They only predicted five severity labels, including blocker, minor, critical, major, and trivial. Then, the five categories were classified as two groups—severe and nonsevere. Blocker, critical, and major were in the severe group while the nonsevere included minor and trivial.

Lamkanfi *et al.* [36] explored several other machine learning algorithms to predict severity labels of bug reports, including Naïve Bayesian, Naïve Bayesian multinomial, SVM, and 1-nearest neighbor. Based on experimental results, they found Naïve Bayesian multinomial is a good choice for severity prediction. Especially, using Naïve Bayesian needs to satisfy a basic assumption: sample attributes are independent of each other [37]. For the severity prediction of the bug report, severity bug reports, and nonseverity bug reports are different in their summary, which means sample attributions of the summary in these two types of bug reports are independent. Hence, Naïve Bayesian performs well in the severity prediction of bug reports. However, in the priority prediction task of bug reports, there are five priority class labels for bug reports. For bug reports with adjacent priority class labels, they may have similar content. A representative example is that if two bug reports with adjacent priority class labels are generated by the same product, their "component" element may be identical, which violates the basic assumption of Naïve Bayesian because bug reports with different priority classes labels have the dependent sample attributions. Even if there are some bug reports from different products, they also may have identical elements, i.e., the "severity" element. Therefore, Naïve Bayesian methods are not suitable for the bug report priority prediction.

Yang *et al.* [38] performed feature selection schemas such as chi-square, correlation coefficient, and information gain to choose the suitable features. Then they put them into the Naïve Bayesian multinomial. The experimental results showed that these features selection schemes perform well for severity prediction.

Zhang *et al.* [29] utilized an enhanced version of REP to capture top-K nearest neighbors of new bug reports. Then they developed a novel classification algorithm by considering the textual similarities between the given bug report and the neighbors.

Recently, Tan *et al.* [39] utilized logistic regression, a simple machine learning algorithm, to predict the severity labels of bug reports. They used BM25 to enrich their bug reports from Mozilla, Eclipse, and GCC with data collected from Stack Overflow.

Our work is different from severity prediction. Severity label is assigned by a reporter who describes the details of a given bug while priority label is assigned by a bug triager who is responsible for assigning the bug to an appropriate bug fixer. [2]. Therefore, there are different features which are adopted in the priority prediction and severity prediction tasks respectively so that it is necessary to design the different approaches to conduct these two different prediction tasks.

C. Other Bug Management Tasks

Besides the priority prediction and severity prediction of bug reports, bug management tasks include bug localization, bug summarization, duplicate bug report detection, etc.

Bug localization is aimed at automatically locating the new bug to reduce the work of developers. Lukins *et al.* [40] used LDA to find the new bug in the source code file. They took the report as a query and then performed LDA to retrieve the source file to concern the localization of the bug. Rao and Kak [41] explored several IR-models when employing the bug localization task, including LDA, cluster-based document model, latent semantic analysis model, vector space model, and unigram model. Zhou *et al.* [42] proposed a method named BugLocator. They first ranked all files according to textual similarity between the source code and the new bug report with a revised vector space model. Moreover, they also ranked the relevant files based on the similar historical bug reports. At last, BugLocator can locate the given bug by combining the two ranks. Kim *et al.* [43] proposed a recommendation model with two phases. They utilized Naïve Bayesian to filter out the useless bug reports and then predicted the buggy file for the given bug report in this model. Saha *et al.* [44] performed AST to program structures of source code files. Then they used Okapi BM25 to compute the similarity between constructs of candidate buggy files and the given bug report. Zamani *et al.* [45] proposed a feature location method based on a novel term-weighting technique which considered how recently one term has been in use in the repositories.

Bug report summarization aims to generate a summary of a bug report by using an automated approach. Rastkar *et al.* [46] used three supervised classification algorithms to generate bug summarizations, which included bug report classifier (BRC), email and meeting classifier (EMC), and email classifier (EC). Mani *et al.* [47] utilized four unsupervised approaches to perform bug report summarization, which included Diverse Rank (DivRank), Grasshopper, maximum marginal relevance (MMR), and Centroid. Jiang *et al.* [48] adopted byte-level N-grams to capture the authorship characteristics of developers. Then they performed the authorship characteristics to collect similar bug reports to employ the bug report summarization task. Najam *et al.* [49] summarized the source code fragment with small-scale crowdsourcing based features. The experimental results indicate that this approach performs better than the existing code fragment classifiers.

Users from different parts of the world may encounter the same bug but submit various bug reports. The goal of duplicate bug report detection is to recognize the duplicate bug reports. A number of approaches have been proposed to detect duplicate bug reports. Most of them rely on the good similarity measure to find bug reports that are similar. They are worked by Sun *et al.* [50], Jalbert and Weimer [51], Wang *et al.* [52], Runeson *et al.* [53], and many more. These studies captured various elements of bug reports and then converted them into vectors. Then these vectors can be used to calculate a similar score between two bug reports. Many of the studies collected the important word tokens appearing in the bug reports via making use of the concepts of inverse document frequency and term frequency. Wang *et al.* [52] utilized execution traces to detect duplicate bug reports. Jalbertand Weimer [51] considered other elements of bug reports (e.g., product) to compute the similarity of two bug reports.

VIII. Conclusion

In this article, we propose a novel framework named PP-WGCN to perform automated priority prediction of bug reports. We extract five elements (i.e., description, summary, severity, component, and product) of bug reports and preprocess them with simple NLP technologies. Finally, we feed the preprocessed bug reports into a classifier based on GCN with a weighted loss function to realize the priority prediction. To verify the effectiveness of our approach, we perform our experiments on four open-source bug repositories, including Mozilla, Eclipse, Netbeans, and GCC. The experimental results show that our approach can outperform two cutting-edge approaches. In addition, our approach can still perform well with a low ratio of the training data. We list two main reasons why our approach performs well: 1) our approach considers five elements of bug reports and capture the global word-word relations; and 2) this GCN model can effectively capture the correlations between two bug reports.

In the future, we plan to further solve the problem of the unbalanced data, including collecting more bug reports to balance our data and attempting more effective approaches. Then, we also plan to develop a complete tool for the priority prediction of bug reports. Beside, we will actively explore the potential of GCN that uses in other software engineering tasks such as bug summarization, code search, and duplicate bug report detection.

References

[1] X. Xia, D. Lo, X. Wang, and B. Zhou, "Accurate developer recommendation for bug resolution," in *Proc. 20th Work. Conf. Reverse Eng.*, 2013, pp. 72–81.

[2] Y. Tian, D. Lo, X. Xia, and C. Sun, "Automated prediction of bug report priority using multi-factor analysis," *Empirical Softw. Eng.*, vol. 20, no. 5, pp. 1354–1383, 2015.

[3] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *Proc. OOPSLA Workshop Eclipse Technol. eXchange*, 2005, pp. 35–39.

[4] X. Xia, D. Lo, M. Wen, E. Shihab, and B. Zhou, "An empirical study of bug report field reassignment," in *Proc. Softw. Evol. Week-IEEE Conf. Softw. Maintenance, Reeng., Reverse Eng.*, 2014, pp. 174–183.

[5] Q. Umer, H. Liu, and I. Illahi, "CNN-based automatic prioritization of bug reports," *IEEE Trans. Rel.*, vol. 69, no. 4, pp. 1341–1354, Dec. 2020.

[6] Q. Umer, H. Liu, and Y. Sultan, "Emotion based automated priority prediction for bug reports," *IEEE Access*, vol. 6, pp. 35 743–35752, 2018.

[7] H. Peng *et al.*, "Large-scale hierarchical text classification with recursively regularized deep graph-CNN," in *Proc. World Wide Web Conf.*, 2018, pp. 1063–1072.

[8] L. Yao, C. Mao, and Y. Luo, "Graph convolutional networks for text classification," in *Proc. AAAI Conf. Artif. Intell.*, 2019, vol. 33, pp. 7370–7377.

[9] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Prof. 5th Int. Conf. Learn. Representation*, 2016.

[10] X. Kong, L. Zhang, W. E. Wong, and B. Li, "Experience report: How do techniques, programs, and tests impact automated program repair?" in *Proc. IEEE 26th Int. Symp. Softw. Rel. Eng.*, 2015, pp. 194–204.

[11] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry, "Are these bugs really" normal"?" in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories*, 2015, pp. 258–268.

[12] H. Cai, V. W. Zheng, and K. C.-C. Chang, "A comprehensive survey of graph embedding: Problems, techniques, and applications," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 9, pp. 1616–1637, Sep. 2018.

[13] P. W. Battaglia *et al.*, "Relational inductive biases, deep learning, and graph networks," 2018, *arXiv:1806.01261*.

[14] J. Yin, J. Liu, and M. Yuan, "Predicting emerging trends of keywords based on graph neural network,"*Int. J. Performability Eng.*, vol. 16, no. 12, pp. 1957–1964, 2020.

[15] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," in *Prof. 2nd Int. Conf. Learn. Representation*, 2013.

[16] M. Henaff, J. Bruna, and Y. LeCun, "Deep convolutional networks on graph-structured data," 2015, *arXiv:1506.05163*.

[17] J. Bastings, I. Titov, W. Aziz, D. Marcheggiani, and K. Sima'an, "Graph convolutional encoders for syntax-aware neural machine translation," in *Prof. Conf. Empir. Methods Nat. Lang. Process.*, 2017, pp. 1957–1967.

[18] Y. Li, R. Jin, and Y. Luo, "Classifying relations in clinical narratives using segment graph convolutional and recurrent neural networks (Seg-GCRNS)," *J. Amer. Med. Inform. Assoc.*, vol. 26, no. 3, pp. 262–268, 2019.

[19] D. Marcheggiani and I. Titov, "Encoding sentences with graph convolutional networks for semantic role labeling," in *Prof. Conf. Empir. Methods Nat. Lang. Process.*, 2017, pp. 1506–1515,.

[20] H. Peng *et al.*, "Large-scale hierarchical text classification with recursively regularized deep graph-CNN," in *Proc. World Wide Web Conf.*, 2018, pp. 1063–1072.

[21] M. Bai, W. Luo, K. Kundu, and R. Urtasun, "Exploiting semantic information and deep matching for optical flow," in *Proc. Eur. Conf. Comput. Vis.*, 2016, pp. 154–170.

[22] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding," in *Prof. Conf. North Amer. Assoc. Comput. Linguist.: Human Lang. Technol.*, 2019, pp. 4171–4186

[23] H. Mi, Z. Wang, and A. Ittycheriah, "Vocabulary manipulation for neura machine translation," in *Prof. 54th Annu. Meeting Assoc. Comput. Linguist.*, 2016, pp. 2016–2021.

[24] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*.

[25] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 9, pp. 1263–1284, Sep. 2009.

[26] Y. Sun, A. K. Wong, and M. S. Kamel, "Classification of imbalanced data: A review," *Int. J. Pattern Recognit. Artif. Intell.*, vol. 23, no. 4, pp. 687–719, 2009.

[27] C. Goutte and E. Gaussier, "A probabilistic interpretation of precision, recall and f-score, with implication for evaluation," in *Proc. Eur. Conf. Inf. Retrieval*, 2005, pp. 345–359.

[28] Q. Li, Z. Han, and X.-M. Wu, "Deeper Insights Into Graph Convolutional Networks for Semi-Supervised Learning," in *Prof. AAAI Conf. Artif. Intell.*, 2018, pp. 3538–3545.

[29] T. Zhang, J. Chen, G. Yang, B. Lee, and X. Luo, "Towards more accurate severity prediction and fixer recommendation of software bugs," *J. Syst. Softw.*, vol. 117, pp. 166–184, 2016.

[30] W. Abdelmoez, M. Kholief, and F. M. Elsalmy, "Bug fix-time prediction model using naïve Bayes classifier," in *Proc. 22nd Int. Conf. Comput. Theory Appl.*, 2012, pp. 167–172.

[31] M. Alenezi and S. Banitaan, "Bug reports prioritization: Which features and classifier to use?" in *Proc. 12th Int. Conf. Mach. Learn. Appl.*, 2013, vol. 2, pp. 112–116.

[32] Y. Li, W. E. Wong, S.-Y. Lee, and F. Wotawa, "Using tri-relation networks for effective software fault-proneness prediction," *IEEE Access*, vol. 7, pp. 63066–63080, 2019.

[33] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2008, pp. 346–355.

[34] W. W. Cohen, "Fast effective rule induction," in *Proc. Mach. Learn.*, 1995, pp. 115–123.

[35] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Proc. 7th IEEE Work. Conf. Mining Softw. Repositories*, 2010, pp. 1–10.

[36] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *Proc. 15th Eur. Conf. Softw. Maintenance Reeng.*, 2011, pp. 249–258.

[37] J. D. Rennie, L. Shih, J. Teevan, and D. R. Karger, "Tackling the poor assumptions of naive Bayes text classifiers," in *Proc. 20th Int. Conf. Mach. Learn.*, 2003, pp. 616–623.

[38] C.-Z. Yang, C.-C. Hou, W.-C. Kao, and X. Chen, "An empirical study on improving severity prediction of defect reports using feature selection," in *Proc. 19th Asia-Pacific Softw. Eng. Conf.*, 2012, vol. 1, pp. 240–249.

[39] Y. Tan, S. Xu, Z. Wang, T. Zhang, Z. Xu, and X. Luo, "Bug severity prediction using question-and-answer pairs from stack overflow," *J. Syst. Softw.*, vol. 165, 2020, Art. no. 110567.

[40] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Source code retrieval for bug localization using latent dirichlet allocation," in *Proc. 15th Work. Conf. Reverse Eng.*, 2008, pp. 155–164.

[41] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: A comparative study of generic and composite text models," in *Proc. 8th Work. Conf. Mining Softw. Repositories*, 2011, pp. 43–52.

[42] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 14–24.

[43] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE Trans. Softw. Eng.*, vol. 39, no. 11, pp. 1597–1610, Nov. 2013.

[44] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2013, pp. 345–355.

[45] S. Zamani, S. P. Lee, R. Shokripour, and J. Anvik, "A noun-based approach to feature location using time-aware term-weighting," *Inf. Softw. Technol.*, vol. 56, no. 8, pp. 991–1011, 2014.

[46] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: A case study of bug reports," in *Proc. ACM/IEEE 32nd Int. Conf. Softw. Eng.*, 2010, vol. 1, pp. 505–514.

[47] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey, "AUSUM: Approach for unsupervised bug report summarization," in *Proc. ACM SIGSOFT 20th Int. Symp. Foundations Softw. Eng.*, 2012, pp. 1–11.

[48] H. Jiang, J. Zhang, H. Ma, N. Nazar, and Z. Ren, "Mining authorship characteristics in bug repositories," *Sci. China Inf. Sci.*, vol. 60, no. 1, 2017, Art. no. 012107.

[49] N. Nazar, H. Jiang, G. Gao, T. Zhang, X. Li, and Z. Ren, "Source code fragment summarization with small-scale crowdsourcing based features," *Front. Comput. Sci.*, vol. 10, no. 3, pp. 504–517, 2016.

[50] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*-Volume 1, 2010, pp. 45–54.

[51] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *Proc. IEEE Int. Conf. Dependable Syst. Netw. With FTCS DCC*, 2008, pp. 52–61.

[52] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 461–470.

[53] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 499–510.