An Exploratory Study of Bug Prioritization and Severity Prediction based on Source Code Features

Chun Ying Zhou
School of Computer Science and Information Engineering
Hubei University
Wuhan, China
zcy9838@stu.hubu.edu.cn

Cheng Zeng
School of Computer Science and Information Engineering
Hubei University
Wuhan, China
zc@hubu.edu.cn

Peng He*
School of Computer Science and Information Engineering
Hubei University
Wuhan, China
penghe@hubu.edu.cn

***Abstract*—Software systems generate a large number of bugs during their lifecycles. Managing and assigning these bug reports is a challenging task. Building prediction models for the priority or severity levels of bugs through bug reports can help developers prioritize highly urgent bugs. Traditional prediction models are based on the textual description information in bug reports. However, most of the description is little or no. According to the bug report, developers need to fix the corresponding source code files. If the corresponding source code file is a core module in a software system, the report is likely to have high-level assignment rights. Therefore, in this paper, we investigate the effect of using the source code file feature sets on classification performance. In addition, we evaluate the effect of different sampling methods on the data, namely SMOTE, RUS, SMOTEEN, Adaboost, and GAN. Extensive experiments were conducted on five open-source projects. The experimental results show that the source code file feature sets do not perform as well as the textual description features in bug reports. Besides, over-sampling methods do not alleviate the data imbalance problem in the case of insufficient data, while GAN performs best in the case of sufficient data.**

***Keywords*—bug priority; bug severity; bug report; source code; data imbalance**

I. INTRODUCTION

Throughout the software lifecycle, developers will receive a large number of bug reports. Once a bug report is committed and confirmed, the bug needs to be fixed in a timely manner. Different analyses can be performed based on the bug reports, including Bug Triage, Bug Localization, Bug-fix Time Estimation, predict defect attributes etc [1]. A bug report contains several attributes, such as *bug id*, *summary*, *description*, *reporter*, *created date*, *fix version*, *status*, *priority*, *severity*, in which the priority levels range from P1 (most important) to P5 (least important), and the severity is categorized into *blocker*, *critical*, *major*, *normal*, *minor*, *enhancement*, and *trivial*. Priority is assigned from the developer's point of view, which indicates the urgency of fixing bugs. Severity is assigned from the user's point of view, which indicates the degree of impact on the use of software function [2].

Existing studies analyzed the text of bug reports by using machine learning methods to automatically predict priority or severity [3]. Given the excellence of deep learning in the field of natural language processing, researchers also explored various neural networks to further extract semantic information from bug reports [1, 6-9]. Unfortunately, if the bug reports provide insufficient or misleading information, the performance of the predictor will be greatly affected. Therefore, in addition to the textual features of bug reports, whether there are other appropriate features for bug prioritization and severity prediction becomes an open challenge. To this end, we will further consider the information of source code files. We assume that if a source code file is a core file in the project, then it has a higher level of importance. Once the file is defective, it is more likely to have a higher priority or severity. Leveraging neural networks to capture semantic and syntactic features from source files is widely used for bug localization [4] and defect prediction [5].

In addition, during bug repair, the co-change relationships between source files have been proved to be potentially valuable [10]. We assume that if two files are associated with a bug report at the same time, there will be some correlation between these two files. More occurrences together indicate a stronger relationship. Hence, we construct a co-occurrence network (COON) between source code files, and extract relational features of the source files by network embedding learning.

The study aims at investigating the feasibility of source code feature sets mentioned above on bug prioritization and severity prediction, and analyzing the impact of different features on this task. To simplify the subsequent presentation, we use *FSet-1* to represent the feature set learned from the textual information of the bug reports, *FSet-2* to represent the semantic feature set of the source code files associated with the bug reports, and *FSet-3* to represent the relational feature set learned from the co-occurrence network of the source code files. Extensive experiments were conducted on five open-source projects to answer the following research questions:

RQ1: Which type of feature set performs better?

RQ2: Is it helpful to consider the importance of source code files?

RQ3: Do the combined feature sets achieve better results?

RQ4: Is the impact of sampling methods obvious?

The remainder of this paper is organized as follows. The related work is presented in Section II. The method is detailed in Section III. The evaluation and analysis are presented in Section IV. Finally, the conclusion is drawn in Section V.

II. RELATED WORK

Most of the existing models predict various attributes based on textual analysis of bug reports (e.g., summary and

Corresponding author: Peng He (penghe@hubu.edu.cn)

DOI reference number: 10.18293/SEKE2022-102

description). Tian et al. [2] treated priority prediction as a linear regression problem rather than a classification problem. The priority level is an ordinal value rather than a categorical value, while classification will make a large difference between levels. Sharma et al. [3] evaluated the performance of different machine learning techniques such as SVM, Naive Bayes, and K-Nearest Neighbors in the priority prediction. Kumari et al. [6] took care of uncertainty by using entropy-based measures. Umer et al. [7] and Ramay et al. [8] proposed an automatic emotion-based prediction method for sentiment analysis of bug reports. Bani-Salameh et al. [9] used a five-layer RNN-LSTM neural network model for bug priority prediction.

If the reporter provided an insufficient description of the bug, it is difficult to learn valuable features from it. As far as we know, few studies have explored bug prediction and severity in terms of the feature sets of source code. In fact, a bug report is often associated with at least one source code file that needs to be fixed. Source code files can be used to indirectly learn about potential bug features. The source code files are converted into Abstract Syntax Trees (ASTs). Then a tree-based neural network is used to extract semantic information. Compared to the source code, the ASTs ignore unnecessary details but still retain lexical and syntactic structure information [11].

With the wide use of complex networks in software engineering, a co-occurrence network is proposed in this paper inspired by this research trend. Considering that source code files are not independent, when two files are associated with a bug report at the same time, there is an association relationship between them, and a complex network is constructed based on this correlation. In fact, real-world networks are often more complex in that not only topological information is available, but also each node and edge has attributes. However, traditional network embedding methods are unsupervised and cannot utilize node attribute information. With the development of Graph Neural Networks (GNNs), a group of models has emerged specifically for learning graph structure data [12], which can smoothly incorporate node and edge attributes while learning the network structure to generate better robust representation.

Unlike existing studies, we not only perform textual analysis of bug reports, but also consider semantic information of source code files and relational information based on co-occurrence network. Therefore, the feature sets in this paper are divided into three categories: textual features of bug reports; semantic features of source code files; and relational features of co-occurrence networks. These three sets of features are then combined and applied to bug prioritization and severity prediction.

III. APPROACH

The workflow (cf. Figure. 1) comprises three parts: (1) *FSet-1* generation: extract the summary and description of bug reports, then use CNN to learn the textual features; (2) *FSet-2* generation: convert source code files to ASTs, then use CNN to extract the code semantic features; (3) *FSet-3* generation: construct a co-occurrence network between source code files, then use GNN to learn the structural features. Finally, the three feature sets are combined in different ways and fed into the classifier for prediction.

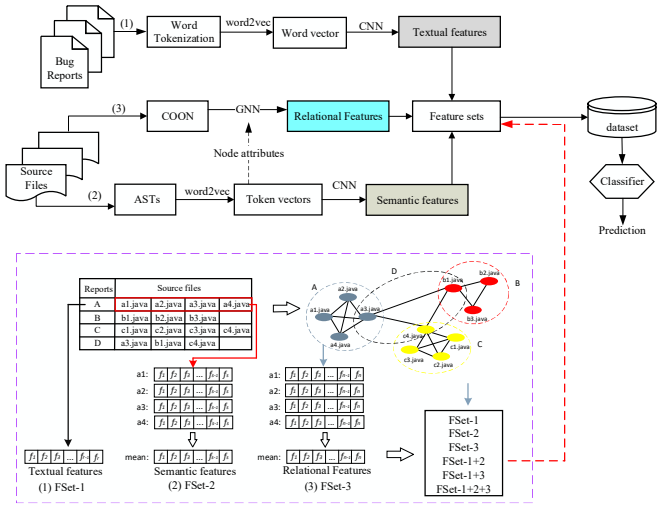

Figure 1. Experimental workflow.

A. FSet-1 generation

The summary and description of each bug report are combined into a document, then all textual information is preprocessed. First, tokenize each word in the text sequences. Second, remove words without real meaning, such as "the", "and", "this", "that", etc. Finally, stem each word into basic words. For example, "working" and "worked" will be converted into "work". In addition, we further processed the tokens as treated in [4]. Since developers usually use compound words to name classes and methods. Therefore, according to CamelCase naming rules, compound words are split into separate real words. For example, "WorkbenchActionBuilder" is split into "Workbench", "Action" and "Builder". After preprocessing, each token is converted into a word vector using *word2vec* and then fed into CNN to extract textual features for bug reports.

B. FSet-2 generation

In this part, the open-source python package javalang[1] is adopted to parse the source code files into ASTs. Three types of nodes were selected as in [13]: (1) nodes of method invocations and class instance creations, (2) declaration nodes, and (3) control-flow nodes. After parsing, each Java file is converted into a token sequence. Since CNN requires input integer vectors, each token is mapped to a unique integer. That is, the token sequence is converted into an integer vector. Since the length of the token sequence of each file is unequal, the dimensions of the converted integer vectors will be different. To keep the same dimension of each file vector, 0 is added at the end of the integer vector, which is equal to the length of the longest vector. Note that adding 0 will not affect the result. Moreover, some uncommon tokens are filtered out and only the tokens that appear more than three times are encoded.

Suppose that there are n Java source code files associated with all bug reports for project P, $P = \{f_1, f_2, \dots, f_n\}$. After the above processing, the token sequence $f_i = \{t_{i1}, t_{i2}, \dots, t_{im}\}$ is extracted, then each token is mapped to an integer, i.e., f_i is converted to a fixed-length integer vector $f_i \in \mathbb{R}^l, i \in [1, n]$. In the embedding layer, f_i is converted to a real-valued vector matrix $X_i = \{\boldsymbol{x}_{i1}, \boldsymbol{x}_{i2}, \dots, \boldsymbol{x}_{il}\}, X_i \in \mathbb{R}^{l \times d}$, where $\boldsymbol{x}_{ij} \in \mathbb{R}^d$ is the embedding vector corresponding to the j-th token t_{ij} of f_i.

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

Since the source code files associated with the bug reports are defective files, clean source code files are also needed to be fed into the CNN for training together as positive instances. Therefore, positive instances are randomly selected from the remaining files in the project, keeping the same number of negative instances. After convolution and pooling layers, the *FSet-2* is generated.

C. FSet-3 generation

Figure 1 also shows the construction of a co-occurrence network through a simple example with four bug reports A, B, C, and D. The source code files associated with these reports are listed on the right. Clearly, for A, B, and C, their associated source code files form three fully connected communities respectively, while the associated files of D connect them, thus forming a large co-occurrence network (COON). In COON, two files may co-occur frequently, so in order to distinguish the strength of the relationship between files, we use the co-occurrence times as a weight. Note that the co-occurrence relationship is undirected in our context.

After constructing COON, the relational features are learned by using a GNN model, which iteratively updates the representation of each node by aggregating the features of its neighboring nodes. This process is mainly divided into two steps. First, aggregate the features of neighboring information to obtain $a_v^{(k)}$, and then combine the neighboring features $a_v^{(k)}$ with the node features of the previous layer $h_v^{(k-1)}$, in order to obtain the updated features.

$$a_v^{(k)} = AGGREGATE^{(k)}(\{h_u^{(k-1)}: u \in N(v)\}) \quad (1)$$

$$h_v^{(k)} = COMBINE^{(k)}(h_v^{(k-1)}, a_v^{(k)}) \quad (2)$$

where $h_v^{(k)}$ is the vector of node v at the k-th layer. $h_v^{(0)} = X_v$. $N(v)$ is the set of neighbors of node v. $AGGREGATE(\cdot)$ and $COMBINE(\cdot)$ are the aggregation function and combination function respectively.

Given that the source code files are defective, and when there is only one class of instance labels, it is not possible to use a supervised model. However, adding additional instances would change the structure of the co-occurrence network. Therefore, we choose an unsupervised model DGI [14] to learn COON. Generating this type of feature set mainly consists of three steps: (1) constructing the network; (2) initializing the node attributes; and (3) using DGI to capture the topological structure information and generate relational features. DGI can learn both network relational and node attribute features. Since the original COON has no node attributes, the node attributes in the COON should be provided before training the DGI. In addition, rich node attributes allow DGI to be better trained. Consequently, the token vectors extracted from the source code are used as the initial node attributes. Finally, the network topology and node attributes are fed into DGI, and *FSet-3* is output.

IV. Experiments and Analysis

A. Datasets

In this paper, we use five public datasets commonly used in Bug Localization[2]. We queried the bug priority and severity attributes from https://bugs.eclipse.org/bugs/ and https://bz.apache.org/bugzilla/ to label these datasets. It is well known that data imbalance is one of the problems in multi-class prediction. To accurately predict the labels for each class, a large amount of data is needed for training. The majority labels of bug reports are P3 or *normal*, which cannot be learned adequately for minor classes. Therefore, priority and severity levels are coarse-grained into three categories. For priority, P1 and P2 are classified as *high*, P3 as *medium*, P4 and P5 as *low*. For severity, *blocker*, *critical* and *major* are classified as *severe*, *normal* as *normal*, *minor*, *enhancement* and *trivial* as *non-severe*.

Table I gives the statistics of the datasets. It can be seen that *medium* and *normal* are the majority categories in most projects. The data imbalance problem of priority prediction is particularly serious. For better empirical validation, we dropped the *medium* category from the priority and the *normal* category from the severity. That is, we perform binary classification aimed to help developers prioritize fixing high-level bugs.

TABLE I. Summary of the Datasets

Products	Bug priority prediction			Bug severity prediction		
	high	***medium***	***low***	***severe***	***normal***	***non-severe***
AspectJ	107	477	9	114	378	101
Eclipse	1072	5301	122	1235	4473	787
JDT	1020	5163	91	690	4410	1174
SWT	230	3901	20	787	3090	274
Tomcat	981	53	22	114	636	306

B. Settings

A five-fold cross-validation is used. For each run, the dataset is divided into five copies, of which 80% is used for training and 20% for testing, with each division ensuring that the ratio of positive to negative instances is approximated. For each project, the experiment was repeated 25 times, and the final results were averaged to reduce the bias introduced by dividing the data randomly.

The output dimension of three feature sets is set to 16. For the CNN used in *FSet-1* and *FSet-2*, a four-layer architecture is employed, including an embedding layer, a convolutional layer, a max-pooling layer and a fully connected layer. The batch size is set to 32 and the epoch is 100, using Adam optimizer with a learning rate of 0.001. For the DGI used in *FSet-3*, a two-layer convolution is employed. The dimension of the hidden layer is set to 64. The DGI is a full batch training with epochs of 200, also using the Adam optimizer with a learning rate of 0.001. The classifier is MLP. The F-measure and Accuracy are used as evaluation metrics in this paper.

C. Analysis

1) RQ1: Which type of feature set performs better?

FSet-1 is the textual features for bug reports. For better illustration, the preprocessed text is treated as the original feature without CNN learning, namely *origin*. Tables II and III show the prediction results. It can be seen that *FSet-2* and *FSet-3* perform poorly, even worse than the *origin*. For priority prediction, due to the extreme imbalance of the datasets, both *FSet-2* and *FSet-3* predict into the majority class labels, which could not construct a reasonable prediction model. For severity prediction, the results are similar to the priority prediction.

[2] https://github.com/yanxiao6/BugLocalization-dataset

Moreover, the performance of *FSet-2* is much worse than that of *FSet-3*.

2) **RQ2: Is it helpful to consider the importance of source code files?**

Based on our assumption, if the source code file is a core file with a high importance level in the project, then the bug reports associated with it are likely to be of high priority or severity level. Since the projects in the dataset used in this paper are not from the same version, and many of them are test files, which cannot be found in the official released version. Therefore, we can only calculate the importance of the source code files from the available data. Specifically, if the corresponding bug report has a *high* or *severe* label, the importance value of all source code files associated with it will be added by 10, while the *medium* and *normal* will be added by 3, and the *low* and *non-severe* will be added by 1. As shown in Figure 2, the importance value of the source code file a.java associated with bug reports A, C and D is 3+1+3=7. Similarly, after calculating the importance values of all source code files, the importance of the bug reports can be obtained indirectly. For example, the importance of bug report A is the sum of the importance values of a.java, b.java, c.java, and d.java. Tables IV and V show the results of priority and severity prediction. △ represents the improvement considering the importance of source code files.

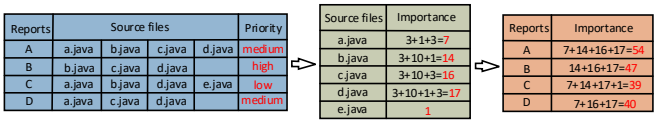

Figure 2. An example of importance calculation.

TABLE II. THE PERFORMANCE OF FEATURE SETS ON PRIORITY

Feature sets	metrics	AspectJ	Eclipse	JDT	SWT	Tomcat	avg
origin	F1	0.518	0.506	0.518	**0.515**	0.496	0.511
	ACC	0.874	0.826	0.853	0.870	0.965	0.878
FSet-1	F1	**0.804**	**0.812**	**0.802**	0.479	**0.770**	**0.734**
	ACC	**0.966**	**0.940**	**0.949**	**0.920**	**0.987**	**0.952**
FSet-2	F1	0.480	0.473	0.479	0.479	0.494	0.481
	ACC	0.922	0.898	0.918	**0.920**	0.978	0.927
FSet-3	F1	0.480	0.473	0.479	0.479	0.494	0.481
	ACC	0.922	0.898	0.918	**0.920**	0.978	0.927

TABLE III. THE PERFORMANCE OF FEATURE SETS ON SEVERITY

Feature sets	metrics	AspectJ	Eclipse	JDT	SWT	Tomcat	avg
origin	F1	0.606	0.560	0.592	0.537	0.555	0.570
	ACC	0.612	0.583	0.617	0.638	0.658	0.622
FSet-1	F1	**0.893**	**0.848**	**0.859**	**0.883**	**0.840**	**0.865**
	ACC	**0.894**	**0.856**	**0.872**	**0.913**	**0.877**	**0.882**
FSet-2	F1	0.398	0.390	0.428	0.427	0.421	0.413
	ACC	0.514	0.610	0.629	0.742	0.729	0.645
FSet-3	F1	0.630	0.532	0.521	0.485	0.481	0.530
	ACC	0.644	0.630	0.648	0.751	0.722	0.679

In priority prediction, using the importance feature for *FSet-1* reduced the overall prediction performance. In particular, the negative impact was significant for *AspectJ* and *Tomcat*, and *SWT* predicted all the labels into the majority class. Due to the relatively sufficient data, the impact on *Eclipse* and *JDT* was slight. For *FSet-2, AspectJ*, *SWT* and *Tomcat,* with insufficient data, still failed to build a reasonable model. For *Eclipse* and *JDT*, on the other hand, achieved some improvements. The impact of importance feature for *FSet-3* was not positive. In severity prediction, using the importance feature for *FSet-1* had a significant negative impact on *AspectJ*, indicated by -32.4% F1 value, but the impact on other projects was not significant. The importance feature showed a great improvement for *FSet-2*, while it had little impact for *FSet-3*.

In short, the results show that the introduction of the importance feature of the source code files does not benefit *FSet-1* and *FSet-3*, and even reduces the prediction performance, while it is a great improvement for *FSet-2*. In addition, based on the use of the source file importance feature, the performance of *FSet-2* is better than that of *FSet-3*. Moreover, the problems of data imbalance and data insufficiency have a great impact on prediction. Data imbalance leads to difficulties in constructing reasonable model, while data insufficiency leads to large fluctuations in prediction results.

TABLE IV. THE PERFORMANCE OF USING IMPORTANCE ON PRIORITY

Feature sets	metrics	AspectJ	Eclipse	JDT	SWT	Tomcat	avg
FSet-1	F1	0.480	**0.829**	0.780	0.479	0.572	0.628
	△	-32.4%	**1.8%**	-2.2%	0.0%	-19.8%	-10.6%
	ACC	0.922	**0.947**	0.947	0.920	0.980	0.943
	△	-4.3%	**0.7%**	-0.3%	0.0%	-0.7%	-0.9%
FSet-2	F1	0.480	**0.579**	**0.514**	0.479	0.494	**0.509**
	△	0.0%	**10.6%**	**3.5%**	0.0%	0.0%	**2.8%**
	ACC	0.922	**0.908**	0.921	0.920	0.978	**0.930**
	△	0.0%	**1.0%**	**0.3%**	0.0%	0.0%	**0.3%**
FSet-3	F1	0.480	0.472	0.476	0.479	0.494	0.480
	△	0.0%	-0.1%	-0.2%	0.0%	0.0%	-0.1%
	ACC	0.922	0.860	0.862	0.920	0.977	0.908
	△	0.0%	-3.8%	-5.6%	0.0%	-0.1%	-1.9%

TABLE V. THE PERFORMANCE OF USING IMPORTANCE ON SEVERITY

Feature sets	metrics	AspectJ	Eclipse	JDT	SWT	Tomcat	avg
FSet-1	F1	0.719	0.845	**0.863**	0.879	**0.860**	0.833
	△	-17.4%	-0.2%	**0.4%**	-0.4%	**2.0%**	-3.2%
	ACC	0.751	0.854	**0.875**	0.910	**0.891**	0.856
	△	-14.3%	-0.1%	**0.3%**	-0.4%	**1.4%**	-2.6%
FSet-2	F1	**0.656**	**0.616**	**0.542**	**0.606**	**0.576**	**0.599**
	△	**25.9%**	**22.6%**	**11.4%**	**17.9%**	**15.5%**	**18.7%**
	ACC	**0.660**	**0.686**	**0.658**	**0.774**	**0.731**	**0.702**
	△	**14.5%**	**7.6%**	**2.9%**	**3.2%**	**0.2%**	**5.7%**
FSet-3	F1	0.526	**0.553**	0.508	**0.525**	**0.588**	**0.540**
	△	-10.3%	**2.1%**	-1.3%	**4.0%**	**10.7%**	**1.0%**
	ACC	0.564	0.625	0.586	0.616	**0.736**	0.625
	△	-8.0%	-0.5%	-6.2%	-13.5%	**1.4%**	-5.4%

3) **RQ3: Do the combined feature sets achieve better results?**

According to the results obtained above, *FSet-2* and *FSet-3* are much less effective than *FSet-1*. Inspired by this, can hybrid feature sets further improve prediction performance? In this RQ, we explore three combinations: *FSet-1+2*, *FSet-1+3* and *FSet-1+2+3*. For example, given a bug report A, the source code files associated with it are $a_1.java$, $a_2.java$, $a_3.java$ and $a_4.java$, where the importance of the source code files are imp_1, imp_2 imp_3, and imp_4. The dimension of *FSet-1* is d_1, while *FSet-2* is d_2 and *FSet-3* is d_3. Then the combined features of A are respectively expressed as follows:

$$\boldsymbol{x_r} = [x_1, x_2, \dots, x_{d_1}] \quad (6)$$

$$\boldsymbol{x_s} = [x_1, x_2, \dots, x_{d_2}] \quad (7)$$

$$\boldsymbol{x_n} = [x_1, x_2, \dots, x_{d_3}] \quad (8)$$

$$\boldsymbol{x}_{\boldsymbol{c_{1+2}}} = \boldsymbol{x_r} \oplus mean(imp_1 \cdot \boldsymbol{x_{s_1}}, imp_2 \cdot \boldsymbol{x_{s_2}}, imp_3 \cdot \boldsymbol{x_{s_3}}, imp_4 \cdot \boldsymbol{x_{s_4}}) \quad (9)$$

$$\boldsymbol{x}_{\boldsymbol{c_{1+3}}} = \boldsymbol{x_r} \oplus mean(imp_1 \cdot \boldsymbol{x_{n_1}}, imp_2 \cdot \boldsymbol{x_{n_2}}, imp_3 \cdot \boldsymbol{x_{n_3}}, imp_4 \cdot \boldsymbol{x_{n_4}}) \quad (10)$$

$$\boldsymbol{x}_{\boldsymbol{c_{1+2+3}}} = \boldsymbol{x_r} \oplus mean\big(imp_1 \cdot \boldsymbol{x_{s_1}}, imp_2 \cdot \boldsymbol{x_{s_2}}, imp_3 \cdot \boldsymbol{x_{s_3}}, imp_4 \cdot \boldsymbol{x_{s_4}}\big) \oplus mean(imp_1 \cdot \boldsymbol{x_{n_1}}, imp_2 \cdot \boldsymbol{x_{n_2}}, imp_3 \cdot \boldsymbol{x_{n_3}}, imp_4 \cdot \boldsymbol{x_{n_4}}) \quad (11)$$

where $\boldsymbol{x_r} \in \mathbb{R}^{d_1}$ is *FSet-1*, $\boldsymbol{x_s} \in \mathbb{R}^{d_2}$ is *FSet-2*, $\boldsymbol{x_n} \in \mathbb{R}^{d_3}$ is *FSet-3*, $\boldsymbol{x}_{\boldsymbol{c_{1+2}}} \in \mathbb{R}^{d_1+d_2}$ is *FSet-1+2*, $\boldsymbol{x}_{\boldsymbol{c_{1+3}}} \in \mathbb{R}^{d_1+d_3}$ is *FSet-1+3*, $\boldsymbol{x}_{\boldsymbol{c_{1+2+3}}} \in \mathbb{R}^{d_1+d_2+d_3}$ is *FSet-1+2+3*. $mean(\cdot)$ is the mean function, $\oplus$ is feature splicing symbol.

Tables VI and VII show the results of priority prediction and severity prediction. △ shows an improvement compared to *FSet-1*. The results show that the combined features are not as effective as expected. *FSet-1+3* and *FSet-1+2+3* were decreased greatly, indicating that *FSet-3* was highly destructive to *FSet-1*. Although *FSet-1+2* decreased slightly and can improve prediction performance in some cases, the enhancement was not significant. Hence, blindly combining feature set for bug prediction can easily lead to negative effects.

TABLE VI. THE PERFORMANCE OF COMBINED FEATURE SETS ON PRIORITY

Feature sets	metrics	AspectJ	Eclipse	JDT	SWT	Tomcat	avg
FSet-1+2	F1	0.679	**0.815**	0.781	0.479	0.581	0.667
	△	-12.5%	**0.4%**	-2.2%	0.0%	-18.9%	-6.7%
	ACC	0.945	0.939	0.945	0.920	0.978	0.945
	△	-2.1%	-0.1%	-0.4%	0.0%	-0.9%	-0.7%
FSet-1+3	F1	0.480	0.558	0.518	**0.485**	0.489	0.506
	△	-32.4%	-25.4%	-28.4%	**0.5%**	-28.1%	-22.8%
	ACC	0.922	0.869	0.904	0.886	0.959	0.908
	△	-4.3%	-7.1%	-4.5%	-3.4%	-2.8%	-4.4%
FSet-1+2+3	F1	0.480	0.595	0.532	**0.496**	0.499	0.520
	△	-32.4%	-21.6%	-27.1%	**1.6%**	-27.1%	-21.3%
	ACC	0.922	0.895	0.888	0.850	0.950	0.901
	△	-4.3%	-4.5%	-6.1%	-7.0%	-3.7%	-5.1%

TABLE VII. THE PERFORMANCE OF COMBINED FEATURE SETS ON SEVERITY

Feature sets	metrics	AspectJ	Eclipse	JDT	SWT	Tomcat	avg
FSet-1+2	F1	**0.907**	**0.848**	0.857	0.865	0.831	0.862
	△	**1.4%**	**0.04%**	-0.2%	-1.8%	-0.9%	-0.3%
	ACC	**0.908**	**0.856**	0.869	0.899	0.869	0.880
	△	**1.4%**	**0.1%**	-0.2%	-1.4%	-0.9%	-0.2%
FSet-1+3	F1	0.872	0.706	0.788	0.748	0.806	0.784
	△	-2.2%	-14.2%	-7.1%	-13.6%	-3.4%	-8.1%
	ACC	0.873	0.722	0.810	0.822	0.852	0.816
	△	-2.1%	-13.4%	-6.1%	-9.1%	-2.5%	-6.7%
FSet-1+2+3	F1	0.851	0.738	0.775	0.716	0.819	0.780
	△	-4.3%	-11.0%	-8.4%	-16.8%	-2.1%	-8.5%
	ACC	0.860	0.758	0.794	0.777	0.863	0.810
	△	-3.4%	-9.8%	-7.7%	-13.6%	-1.4%	-7.2%

4) **RQ4: Is the impact of sampling methods obvious?**

From Table I, it is clear that the data used for prediction are obviously unbalanced and insufficient. For example, for *SWT,* the ratio of majority class to minority class was as high as 15:1. Extreme data imbalance could easily lead to the prediction results completely biased towards the majority class, or the results may be very unstable. Therefore, we further investigated several data sampling methods to explore whether they could alleviate the data imbalance problem: (1) SMOTE [15] (2) RUS [16] (3) SMOTEENN [17] (4) AdaBoost [18] (5) GAN [19]. SMOTE is an over-sampling method, RUS is an under-sampling method, SMOTEENN is a comprehensive method combining over-sampling and under-sampling, and AdaBoost is an integrated learning method. GAN is a neural network method that constructs two networks, a generator and a discriminator. These two models compete with each other so that the generator generates instances closer to the ground truth, while the discriminator is getting stronger to identify the fake instances and ground truth.

TABLE VIII. THE PERFORMANCE OF DIFFERENT SAMPLING METHODS ON PREDICTION

Feature sets	metrics	AspectJ	Eclipse	JDT	SWT	Tomcat	avg
non-sample	F1	0.804	**0.812**	**0.802**	0.479	0.770	0.734
	ACC	**0.966**	**0.940**	**0.949**	0.920	0.987	0.952
SMOTE	F1	**0.878**	0.735	0.773	0.668	0.650	0.741
	ACC	0.964	0.866	0.912	0.854	0.940	0.907
SMOTE-ENN	F1	0.790	0.686	0.733	0.626	0.619	0.691
	ACC	0.927	0.815	0.877	0.802	0.918	0.868
Adaboost	F1	0.814	0.801	0.789	**0.732**	**0.827**	**0.793**
	ACC	**0.966**	0.938	0.948	**0.943**	**0.988**	**0.957**
GAN	F1	0.478	0.730	0.733	0.479	0.495	0.583
	ACC	0.917	0.925	0.937	0.920	0.980	0.936

TABLE IX. THE PERFORMANCE OF DIFFERENT SAMPLING METHODS ON SEVERITY

Feature sets	metrics	Eclipse	JDT	SWT	Tomcat	avg
non-sample	F1	0.848	0.859	0.883	0.840	0.858
	ACC	0.856	**0.872**	0.913	0.877	0.879
SMOTE	F1	0.841	0.848	0.872	0.819	0.845
	ACC	0.847	0.858	0.899	0.851	0.864
RUS	F1	0.864	**0.865**	0.893	0.842	0.866
	ACC	0.864	0.865	0.893	0.843	0.866
SMOTEENN	F1	0.838	0.842	0.842	0.797	0.830
	ACC	0.842	0.850	0.868	0.826	0.847
AdaBoost	F1	0.831	0.851	0.876	0.840	0.849
	ACC	0.840	0.864	0.909	0.877	0.872
GAN	F1	**0.876**	0.860	**0.896**	**0.894**	**0.882**
	ACC	**0.884**	0.869	**0.920**	**0.917**	**0.897**

Tables VIII and IX show the results. In priority prediction, AdaBoost outperforms the other sampling methods, while GAN is the worst. The advantages of SMOTE and SMOTEENN are not outstanding, as indicated by both enhancement and reduction. In severity prediction, GAN performs best, followed by RUS, which is slightly less accurate than AdaBoost. AdaBoost, SMOTE, and SMOTEENN showed an overall decrease in performance. The results suggest that over-sampling and comprehensive sampling may not be able to synthesize high-quality instances without sufficient data. Instead, RUS can avoid noise caused by synthesized instances when sufficient data is available. Note that GAN performs the worst in priority prediction but the best in severity prediction due to the fact that GAN requires a large amount of data for training. Therefore, we recommend using the GAN model for bug prediction when training data is sufficient.

V. DISCUSSION

In this section, we discuss the shortcomings of the research questions in our study.

Firstly, for *FSet-2*, we use CNN to extract the semantic features of source code files. As we know, CNN is a supervised model, the source code files associated with bug reports are considered as buggy. Therefore, additional clean files need to be fed to CNN for training along with the buggy files. We randomly select the remaining source files in the project as clean instances, which is consistent with the number of buggy instances. However, this may have potential noise, which leads to poor performance of *FSet-2*. First, not all the remaining source code files in the project are clean files. Second, since the mapping between bug reports and source code files is many-to-many, we can only determine the priority and severity of bug reports, not the source code files. Therefore, our treatment in this paper may lead to the learned features deviating from the actual predictions.

Secondly, for *FSet-3*, building a software network [5] based on the actual dependencies between code files may better represent their relationships. Since the projects in the datasets may across multiple versions, it is not possible to construct the corresponding software network accurately, so we use a co-occurrence network (COON) instead. Note that COON is actually a virtual network, in which the relationship between files is not completely real. Moreover, the stability of the network may change greatly as the dataset expands. This could also explain the unsatisfactory performance of the relational features.

Finally, the results show that the importance feature of the source code files is useful for *FSet-2*. In fact, our method involves only some of the source files, but the importance of these files is calculated from the perspective of the project, such as Key Class Identification [20]. Hence, there is still some deviation from the real situation.

VI. CONCLUSION

This study attempts to explore the impact of source code files feature sets on bug prioritization and severity prediction. Leveraging CNN and GNN to learn the semantic features of source code files and the relational features between these files. The experimental results show that the impact of learning feature sets is not as expected. In addition, five typical sampling methods are also introduced to analyze the impact on data imbalance. The results show that the GAN model synthesizes the highest quality instances with an adequate dataset, significantly improving the results on F-measure and Accuracy. Next is under-sampling, while over-sampling and comprehensive sampling do not perform well. In future work, in order to further validate the work of this paper accurately, we need to construct a dataset suitable for the method of this paper. The effects caused by other factors such as data quality should be excluded as much as possible, so that the source code file feature sets can be fully utilized.

ACKNOWLEDGMENT

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

REFERENCES

[1] M. Mihaylov, M. Roper. Predicting the Resolution Time and Priority of Bug Reports: A Deep Learning Approach, Ph.D. dissertation, Department of Computer and Information Sciences, University of Strathclyde, 2019.

[2] Tian Y, Lo D, Sun C. Drone: Predicting priority of reported bugs by multi-factor analysis[C]//2013 IEEE International Conference on Software Maintenance. IEEE, 2013: 200-209.

[3] Sharma M, Kumari M, Singh R K, et al. Multiattribute based machine learning models for severity prediction in cross project context[C]//International Conference on Computational Science and Its Applications. Springer, Cham, 2014: 227-241.

[4] Xiao Y, Keung J, Mi Q, et al. Improving bug localization with an enhanced convolutional neural network[C]//2017 24th Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2017: 338-347.

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

[6] Sharma M, Kumari M, Singh R K, et al. Multiattribute based machine learning models for severity prediction in cross project context[C]//International Conference on Computational Science and Its Applications. Springer, Cham, 2014: 227-241.

[7] Umer Q, Liu H, Sultan Y. Emotion based automated priority prediction for bug reports[J]. IEEE Access, 2018, 6: 35743-35752.

[8] Ramay W Y, Umer Q, Yin X C, et al. Deep neural network-based severity prediction of bug reports[J]. IEEE Access, 2019, 7: 46846-46857.

[9] Bani-Salameh H, Sallam M. A Deep-Learning-Based Bug Priority Prediction Using RNN-LSTM Neural Networks[J]. E-Informatica Software Engineering Journal, 2021, 15(1) DOI:10.37190/e-Inf210102.

[10] McIntosh S, Adams B, Nagappan M, et al. Mining co-change information to understand when build changes are necessary[C]//2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, 2014: 241-250.

[11] Zhang J, Wang X, Zhang H, et al. A novel neural source code representation based on abstract syntax tree[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 783-794.

[12] Wu Z, Pan S, Chen F, et al. A comprehensive survey on graph neural networks[J]. IEEE Transactions on Neural Networks and Learning Systems, 2020.

[13] Wang S, Liu T, Nam J, et al. Deep semantic feature learning for software defect prediction[J]. IEEE Transactions on Software Engineering, 2018.

[14] Velickovic P, Fedus W, Hamilton W L, et al. Deep Graph Infomax[J]. ICLR (Poster), 2019, 2(3): 4.

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

[16] Weiss G M. Foundations of imbalanced learning[J]. Imbalanced Learning: Foundations, Algorithms, and Applications, 2013: 13-41.

[17] Lamari M, Azizi N, Hammami N E, et al. SMOTE–ENN-Based Data Sampling and Improved Dynamic Ensemble Selection for Imbalanced Medical Data Classification[M]//Advances on Smart and Soft Computing. Springer, Singapore, 2021: 37-49.

[18] Ying C, Qi-Guang M, Jia-Chen L, et al. Advance and prospects of AdaBoost algorithm[J]. Acta Automatica Sinica, 2013, 39(6): 745-758.

[19] Sun Y, Jing X Y, Wu F, et al. Adversarial learning for cross-project semi-supervised defect prediction[J]. IEEE Access, 2020, 8: 32674-32687.

[20] Pan W, Song B, Li K, et al. Identifying key classes in object-oriented software using generalized k-core decomposition[J]. Future Generation Computer Systems, 2018, 81: 188-202.