EKD-BSP: Bug Report Severity Prediction by Extracting Keywords from Description

Yanxin Jia†, Xiang Chen†‡*, Shuyuan Xu†, Guang Yang†, Jinxin Cao†
†*School of Information Science and Technology*, *Nantong University*, China
‡*Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics)*,
Ministry of Industry and Information Technology, China
Email: yanxin15@yeah.net, xchencs@ntu.edu.cn, 1113585141@qq.com
1930320014@stmail.ntu.edu.cn, alfred7c@ntu.edu.cn

***Abstract*—Bug severity is important for triagers. Recently, the text in the summary field (i.e., bug summary) of bug reports is usually used to extract features, and then bug report severity prediction models are constructed. In some bug reports, the bug summary may not contain enough useful information. While the text field in the description (i.e., bug description) of bug reports contains detailed information of the bug (e.g., steps to reproduce the bug, stack traces, and expected behavior). However, the bug description may contain irrelevant information. Motivated by the above findings, we propose a novel method EKD-BSP (Bug Report Severity Prediction by Extracting Keywords from Description), which uses the bug summary and the keywords extracted from the bug description to perform severity prediction. Our empirical study selects two large-scale open-source projects (i.e., Eclipse and Mozilla) as the empirical subjects. The empirical results show that EKD-BSP can improve the performance of $F\text{-}measure$ by up to 5.19% after compared with the baselines.**

***Index Terms*—Bug report severity prediction, Bug report description, Keyword extraction, Text mining**

I. Introduction

With the increasing scale and complexity of software projects, bugs in software are unavoidable during the process of software development and maintenance. The previous study [1] shows that about 90% of bugs have a serious negative impact on the developer's experience and even result in huge economic loss. Therefore it is particularly important to track and manage bugs in software projects.

Currently, most of the software projects employ bug tracking systems (such as Bugzilla[1], JIRA[2]) to assist in bug management. When detecting a bug, one will submit a bug report to the bug tracking system. A bug report generally consists of ID number, title, report time, severity, assigned developer, resolution state (e.g., new, assigned, resolved, closed), description, comments (e.g., discussion about the possible solutions), attachments (e.g., patches, test cases), and ID number of underlying reports. In the bug reports, severity is an important field and is often assigned manually by users who submitted these bug reports. It is critical for developers to set the reasonable priority of these bug reports. However, users often fail to assign the reasonable severity of bug reports due to the lack of experience in software development. To address this problem, researchers [2] [3] [4] [5] [6] aimed to construct high-quality bug report severity prediction models, which can predict appropriate severity after analyzing the contents in the bug reports.

Most of the previous studies [4] [7] [8] performed bug report severity prediction by using the text in the summary field (i.e., bug summary). However, Chen et al. [9] found that the quality of the bug summary has not received enough attention. For some bug reports, the bug summary may contain a few words or do not contain enough useful information. While the text in the description field (i.e., bug description) of bug reports contains detailed information of bug (e.g., steps to reproduce the bug, stack traces, and expected behavior). Therefore, utilizing the information in the bug description is a potential way to improve the performance of the bug report severity prediction. In previous studies, Tian et al. [10] and Otoom et al. [11] used the bug summary and the bug description for bug severity prediction. But neither of these two studies investigated the prediction model construction by considering the information from both the bug summary and the bug description.

We use an example to show the motivation of our study. Table I shows a bug report[3] from the Eclipse project. From Table I, we can find the bug summary is "Auth settings for Docker Hub repository", which mainly contains keywords (such as "Auth", "repository"). However, only considering this information is not enough to determine the bug report severity. After analyzing the bug description, the developer can understand the root cause of this bug. Specifically, this bug is caused by the user, who does not have enough write permissions. If this bug is not fixed in time, it will affect the subsequent project development. Therefore this bug is a blocker bug. Moreover, not all the information (such as source code [12], URL, console output information, irrelevant information) in the bug description is useful for bug report severity prediction. Therefore, in our study, we resort to the keyword extraction method to identify useful information in the bug description.

Based on the above motivation example, we propose a

* Xiang Chen is the corresponding author.

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

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

[3]https://bugs.Eclipse.org/bugs/show_bug.cgi?id=564155

22767-6684/21/$31.00 ©2021 IEEE
DOI 10.1109/DSA52907.2021.00014

TABLE I
A BUG REPORT FROM THE ECLIPSE PROJECT

Bug Summary	Auth settings for Docker Hub repository
Bug Description	Hi, the Honor project publishes Docker images to repositories under the Eclipse organization at hub.docker.com. We have created a new repository Eclipse/hono-service-device-registry-file which I would like to have all Hono committees write access to. For that purpose, the "Hono" team already exists in the org Can you please give write access for the repo to that team? Can you also please verify that this team has write access to all of the other Eclipse/hono-* repositories? Without write access we cannot perform our releases... Thanks, Kai

novel method EKD-BSP (**B**ug Report **S**everity **P**rediction by **E**xtracting **K**eywords from **D**escription), which uses the bug summary and the keywords extracted from the bug description to perform bug report severity prediction. EKD-BSP mainly consists of the model construction phase and model application phase. In particular, the model construction phase includes four steps: (1) Text Preprocessing: it uses the standard preprocessing step, the tag substitution step for text processing of the bug summary, and the bug description. (2) Keyword Extraction from the Bug Description: it uses the TextRank algorithm [13] to extract keywords from the bug description. (3) Word Embedding: it uses the FastText model [14] to get the word embedding. (4) Model Construction via Classifier: it uses the LR (Logistic Regression) classifier to construct the bug severity prediction model. In the model application phase, we can use the trained model to predict the severity of the new bug reports.

In our empirical study, we choose two large-scale open-source projects (i.e., Eclipse and Mozilla) as our experimental subjects. According to a recent survey on bug report severity prediction by Gomes et al. [15], they find 92% of the studies used the Eclipse project as the experimental subject, and 70% of the studies used the Mozilla project as the experimental subject. Therefore, using these two open-source projects can guarantee the generality of our empirical results. Then we first conducted empirical studies to evaluate the performance of our proposed method EKD-BSP by comparing state-of-the-art baselines in bug report severity prediction. We also design experiments to verify the effectiveness of component sets (such as extracting keywords from the bug description, using the LR as the classifier) in our proposed method EKD-BSP.

Our empirical results show that our proposed method EKD-BSP can achieve promising performance (i.e., 70.43% $F\text{-}measure$, 73.81% $Precision$, and 68.67% $Recall$ for Eclipse, and 80.09% $F\text{-}measure$, 82.01% $Precision$, and 78.76% $Recall$ for Mozilla). After compared with the state-of-the-art baselines, the method EKD-BSP can improve the performance by up to 5.19% in terms of $F\text{-}measure$. Moreover, we find keywords extracted from the bug report only need to keep 20% of words in the original bug description and can effectively improve the prediction performance when compared with the method only considering the bug summary. Finally, using the LR classifier can achieve the best performance in our proposed method.

To our best knowledge, the main contributions of our study can be summarized as follows:

- We propose a novel method EKD-BSP, which can predict the bug report severity by additionally using keywords extracted from the bug description. Specifically, we use the FastText model [14] to train word embedding for the bug summary and the bug description. Then we use the TextRank algorithm [13] to extract the keywords from the bug description.
- We choose two large-scale open-source projects as our empirical subjects. Based on empirical results, we find our proposed method EKD-BSP can achieve better performance than state-of-the-art baselines in the bug report severity prediction. Moreover, we also show the competitiveness of extracting keywords from the bug description and using the LR as the classifier in EKD-BSP.

The rest of this paper is organized as follows. Section II introduces the research background of bug report management and related work of bug report severity prediction. Section III shows the details of our proposed method EKD-BSP, including the model construction phase and the model application phase. Section IV introduces the experimental setup, including the experimental subjects, performance measures, and experimental setting. Section V summarizes the experimental results. Section VI analyzes potential threats to the validity of our empirical study. Section VII concludes this paper and discusses some potential future work.

II. BACKGROUND AND RELATED WORK

In this section, we mainly introduce the background of bug report management and related work of bug report severity prediction.

A. Bug Report Management

Currently, bug tracking systems are used to manage bug reports. Specifically, in Bugzilla, when a bug is first reported, the initial status of the bug report is set to "Unconfirmed". When a triager verifies that this bug is not duplicated and is a new bug, the status of the bug report is set to "New". Then the triager assigns this bug report to an appropriate developer for bug fixing, and the status is set to "Assigned". Later, the assigned developer reproduces the reported bug and then fixes

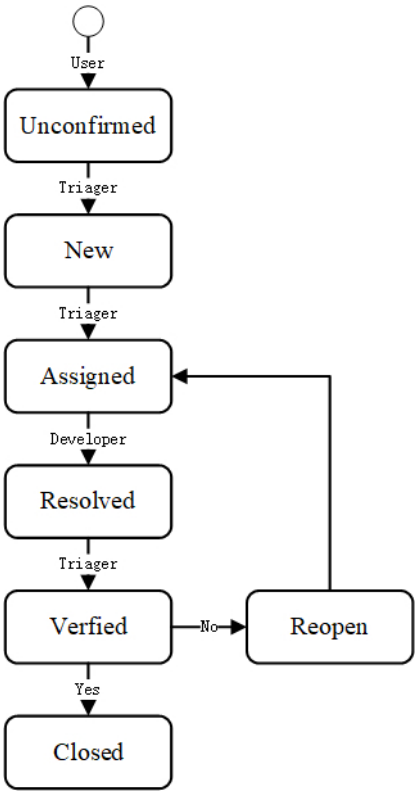

Fig. 1. The bug report life-cycle.

this bug. The status is changed to "Resolved". After that, if the triager verifies that the bug is fixed, the status is changed to "Verified", otherwise the status is changed to "Reopen" to implement a reassignment. The final status of the bug report is "Closed" when no occurrence of this bug is reported. The detailed life-cycle of the bug report is shown in Fig. 1.

In a survey on bug report analysis [16], Jie et al. classified the previous studies on bug report management from two perspectives. Specifically, from the reporters' perspective, most of the studies focused on improving the quality of bug reports. From the developers' perspective, most of the studies focused on automating bug report triage and fix. In our study, we mainly focus on bug report severity prediction, which can improve the quality of the bug reports.

B. Bug Report Severity Prediction

Based on the latest statistical results, the Mozilla project received 212 new bug reports on average for each week, while the Eclipse project received 224 new bug reports on average for each week [17]. To reduce the workload of manual labor and realize reasonable assignment of bugs, the bug report severity prediction problem has been studied. The bug report severity and its corresponding description can be found in Table II.

In this research topic, the first study was conducted by Menzies and Marcus [2]. They modeled the bug report severity prediction task as a classification problem and proposed an automatic bug severity prediction method by combining text

TABLE II
DETAILED DESCRIPTION OF BUG REPORT SEVERITY

Severity	Description
Blocker	Bugs that halt the development process and do not have any work around
Critical	Bugs that cause loss of data or severe memory leaks
Major	Bugs that are seriously obstacle to work with the software system
Normal	Bugs that are advised to be chosen when the user is not sure about the bug or if the bug is related to documentation
Minor	Bugs that are worth reporting but do not interfere with the functionality of program
Trivial	Bugs that are cosmetic bugs (such as typo in the java docs)
Enhancement	Bugs that are the gray areas (such as new features that are not considered as bug)

mining and machine learning. Then Lamkanfi et al. [3] used text mining algorithms to construct severity prediction models. After that, they [4] found MNB (Multinomial Naive Bayes) classifier with text mining outperforms other considered algorithms. Pushpalatha and Marlakunta [5] used supervised and unsupervised methods to predict the severity of bug reports.

For the sake of simplicity, the previous studies can be classified into three categories: extracting features from the bug report, using feature selection to improve the model performance, and considering different modeling methods.

The features extracted from the bug report have a high impact on the performance of the bug report severity prediction performance. Yang et al. [18] used multiple features (e.g., component, product, priority) to predict bug report severity. Sharma et al. [19] used priority, number of comments, number of dependents, number of duplicates, complexity, summary weight, and CC List to predict bug report severity in a cross-project scenario. Jin et al. [20] used the features from the bug summary and the bug description for model construction. In addition to the meta-features used by the above study, other features (such as report length, emotion words) were also used by the researchers. Yang et al. [21] integrated stack traces, steps to reproduce, attachments, report length, and several quality indicators. Then they explored their influence on severity prediction performance. Yang et al. [12] analyzed emotion words and used an EWD-multinomial classifier for bug report severity prediction. The empirical results showed the competitiveness of their proposed algorithm. Sabor et al. [22] combined a neural network model with stack traces to predict bug report severity. This method can achieve 73% $Accuracy$ in Eclipse product data and 85% $Accuracy$ in Eclipse component data. They [23] also used a linear combination of the stack trace and categorical feature similarity.

Some researchers used feature selection [24] [25] [26] [27] to improve the model performance. Yang et al. [7] used feature selection methods (i.e., information gain, chi-square, and correlation coefficient). Roy and Rossi [28] used text mining together with bi-grams and feature selection to improve the severity prediction performance of bug reports. Sharma et al. [29] used two feature selection methods (i.e., information gain and chi-square) to train the bug report severity prediction

model. Liu et al. [30] integrated different feature selection methods to form an ensemble feature selection method. Empirical results of these studies showed that using the ensemble feature selection method can improve the performance of bug report severity prediction.

Some researchers considered different modeling methods. For example, Gou et al. [31] proposed a new method DWKNN (Distance-Weighted K-Nearest Neighbor) rule. Compared to the state-of-the-art KNN-based methods, this method can achieve better performance under different K values. Tian et al. [10] proposed a new approach leveraging information retrieval to automatically predict the severity of bug reports. Zhang et al. [32] proposed a concept profile-based prediction technique and experimental results show that the proposed technique can effectively predict the severity of a new bug report. Zhang et al. [33] utilized a modified REP algorithm (i.e., REPtopic), K-Nearest Neighbor (KNN) classification to search the similar bugs and their features to construct the bug severity prediction models. Tian et al. [34] considered the problem of inconsistent severity in duplicated bug reports. Zhang et al. [35] found that most research work adopts a supervised and data-driven approach. This kind of approach may fail when the number of the training dataset is limited. Therefore, they proposed a method to label the severity of bug reports via active learning and used a sample enhancement method to train the models. Tan et al. [36] used the classical retrieval algorithm BM25 to search the data set gathered from Stack Overflow and selected 500 pairs of high similarity data for each project to realize the data set augmentation.

Different from the previous study, we conjecture that the bug description of some bug reports may contain useful information for bug severity prediction, which can not be found in the corresponding bug summary. Based on this conjecture, we propose a novel method EKD-BSP, which aims to augment the bug summary by the keywords extracted from the bug description. Then we want to use the bug reports from real-world open-source projects to verify the effectiveness of our proposed method.

III. Our Proposed Method

In this section, we propose a novel bug report severity prediction method EKD-BSP. Compared to previous studies [6] [12] [37], we combine the bug summary with keywords extracted from the bug description to predict the severity of bug reports. In our proposed method EKD-BSP, we model the severity prediction problem as a binary classification problem. Specifically, based on Table II, the severity Blocker, Critical or Major are classified as Severe type. While the severity Minor or Trivial are classified as Non-Severe type. Note we do not consider the severity Normal and Enhancement in our study. More detailed analysis can be found in Section IV-A.

Fig. 2 shows the framework of our proposed method EKD-BSP. According to this figure, our proposed method consists of two phases (i.e., the model construction phase and the model application phase). In the rest of this section, we will show the details of these two phases.

A. Model Construction Phase

The model construction phase mainly includes four steps: text preprocessing on the bug summary and the bug description, keyword extraction from the bug description, word embedding, model construction via classifier.

1) Text Preprocessing: To improve the performance of EKD-BSP, we take standard preprocessing step (i.e., tokenization, stop-word removal, and lemmatization) for the bug summary and the bug description. The bug description contains detailed information of the bug report, such as steps to reproduce the bug, stack traces, and expected behavior. However, not all of this information is useful for model construction. According to the previous analysis on the bug description, Chen et al. [9] found that URL requires extra efforts to analysis, but brings limited benefits in understanding the bug summary. Moreover, Yang et al. [12] also found analyzing the source code in the bug description cannot help to achieve better performance. Finally, in the bug description, the console output information, which is similar to the source code, also does not help to improve the performance. Therefore, we use the regular expression to match the URL, source code, and console output from the bug description and then use the corresponding tag to replace them.

For the bug summary, we only take standard preprocessing step (i.e., tokenization, stop-word removal, and lemmatization). The standard preprocessing step is common for text preprocessing in NLP (Natural Language Processing). Specifically, tokenization is a preliminary processing step of the original text data and is used to split the bug summary into a series of words (i.e., tokens). Moreover, this step can remove the special words (such as punctuation and comma). Stop-words (such as “the”, “is”, “at”, “which”, “on”) are words that have high frequency but do not have any real meaning. Removing these stop-words can improve the model performance and reduce the size of the extracted words. Lemmatization is used to restore various forms of words to their root forms. In the natural language, words often have different tenses, but the meanings of words are similar. Therefore, we take lemmatization for tokens to reduce the redundancy of the text information. For example, “make”, “makes”, and “making” are different tenses of “make”, however, they mean the same thing.

For the bug description, we take both the tag substitution step and the standard preprocessing step, which are the same as the standard preprocessing step used in the bug summary. The tag substitution step will use corresponding tags to replace the web URL, file URL, source code, and console output in the bug description. We show the used regular expressions for identifying the corresponding tags in Table III. The details of the tag substitution step are described as follows.

- If the web URL link or file URL link is matched, it will be replaced with the “url” tag.
- If the source code is matched, it will be replaced with the “code” tag.

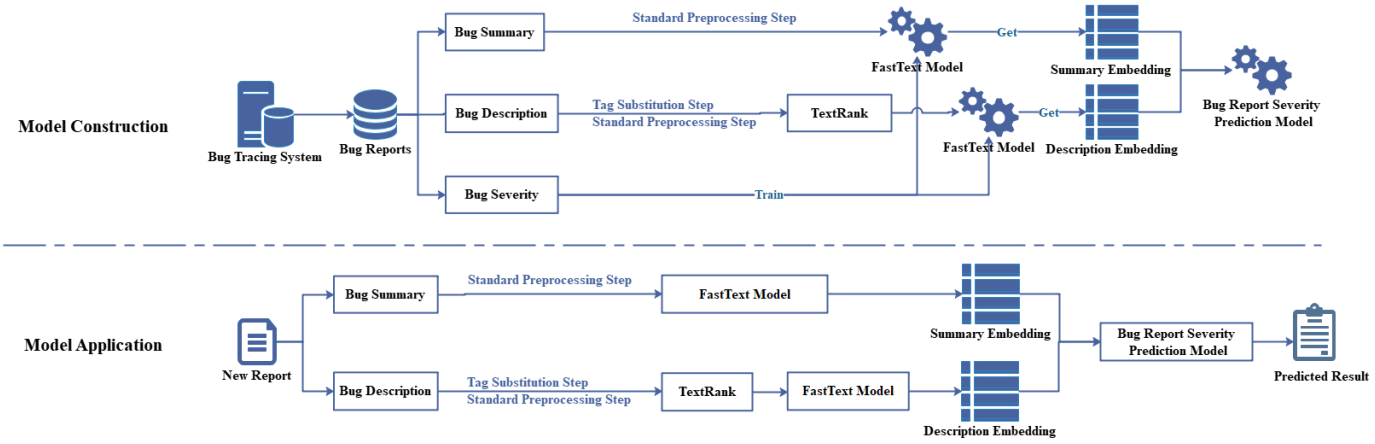

Fig. 2. The framework of our proposed method EKD-BSP

- If the console output information is matched, it will be replaced with the "console" tag.

TABLE III
THE REGULAR EXPRESSION FOR IDENTIFYING THE CORRESPONDING TAG

Tag	Regular Expression
Web URL	`[https|http]+:[^\s]*`
File URL	`[file|files]+://[^\s]*`
Source Code	`(?<=\{).*(?=\})`
Console Output	`(/[^\}]*/)`

2) Keyword Extraction from the Bug Description: To identify useful information in the bug description, we resort to keyword extraction. Because using keyword extraction can automatically obtain a series of words that can represent the information of the original document. EKD-BSP uses the TextRank algorithm [13] for keyword extraction after text preprocessing of the bug description.

The TextRank algorithm [13] is an unsupervised method. Specifically, TextRank is an algorithm that represents a given text and interconnects words with meaningful relations based on a graph. This algorithm starts with tokenization and part-of-speech tagging of the words and then passes syntactic filters (e.g., reserved nouns, verbs) as vertices of the graph. Next co-occur with a window of N-words to define the relation of edges in two vertices. After the graph is constructed, based on the idea of the PageRank algorithm [38], this algorithm will run for several iterations until it converges. Once a final score is obtained for each vertex in the graph, vertices are sorted in the reversed order based on their score and this algorithm can get the top vertex. This graph-based sorting algorithm is an essential way to determine the importance of vertices in a graph based on global information. Therefore, it is a good way to discover global key information in a document.

Since EKD-BSP involves the information of two text fields in the bug reports (i.e., the bug summary and the bug description), and our processing of them is slightly different from the classical text processing. Therefore, to show the processing details of the bug summary and the bug description by EKD-BSP, we choose an example to show the processing results, which can be found in Table IV. In this table, the bug summary only uses the standard preprocessing step to obtain the tokens. The bug description takes the tag substitution step to preprocess the URL, source code, console output information in the bug report and takes the standard preprocessing step to obtain the tokens. We use braces to match code snippets, therefore, the code snippet inside the brace is replaced with the "code" string in Table IV. Next, we extract keywords from the bug description. Finally, we show the obtained tokens in the bug summary and the bug description respectively.

3) Word Embedding: To learn word representation, we use the FastText model [14]. In the previous studies, most of the work [10] [28] [39] used TF-IDF for word representation, which was based on the word frequency, and the learned semantic information was limited. Recently, the FastText model can learn word representation while taking into account morphology [40] [41]. Empirical studies [14] showed that the FastText model can achieve state-of-the-art performance on word representation. Specifically, it is represented by considering subword units, which can represent words by a sum of its character n-grams. Moreover, the FastText model can be trained over a billion words in less than ten minutes using a standard multi-core CPU, which is several orders of magnitude faster than deep model training. In summary, it has many advantages (such as simple model construction, fast training speed, and high model performance) [40].

We construct and train the FastText models for the bug summary and the bug description respectively. When constructing the FastText model for the bug summary, the input of this model is the bug summary after text preprocessing and the severity of the bug report. For the bug description, the input of the FastText model is the bug description after keyword extraction and the severity of the bug report. Then we will train these two FastText models (i.e., the bug summary model $FastT_{sum}$ and the bug description model $FastT_{des}$). These models can generate word vectors for the words in the

TABLE IV
EXAMPLE FOR TEXT PROCESSING AND KEYWORD EXTRACTION IN THE BUG SUMMARY AND THE BUG DESCRIPTION

Step	Substep	Bug summary	Bug description
Origin Text		compiler bug: overwriting implicitely abstract method in anonymous inner class	The following code results in a compile time error (build 20020214): `public void Test() { AbstractTableModel tm = new AbstractTableModel(){ public int getColumnCount(){ return 0; } public int getRowCount(){ return 0; } public Object getValueAt(int rowIndex, int column Index){ return null; } }; tm.getColumnCount(); // <-- compile time error }` The method geColumnCount() is undefined for the type javax.swing.table.AbstractTableModel"
Text Preprocessing	Tag Substitution Step	No	The following code results in a compile time error (build 20020214): public void Test() { code }."The method geColumnCount() is undefined for the type .javax.swing.table.AbstractTableModel"
	Standard Preprocessing Step	compiler bug overwrite implicitely abstract method anonymous inner class	the following code result compile time error build 20020214 public void test code the method gecolumncount undefined type javax swing table abstract-tablemodel
Keyword Extraction		No	javax following code result compile time error build method gecolumncount undefined type public void test
Obtained Tokens		compiler bug overwrite implicitely abstract method anonymous inner class	javax following code result compile time error build method gecolumncount undefined type public void test

bug summary or the bug description. By summing the word vectors of words existing in the bug report [36], we can get a summary embedding $Embed_{sum}$ and a description embedding $Embed_{des}$. Then, we concatenate the embedding $Embed_{sum}$ and the embedding $Embed_{des}$ as the embedding of the bug report $Embed_{bug}$ for the subsequent model training.

4) Model Construction via Classifier: To construct the bug severity prediction model, we use the LR (Logistic Regression) classifier. LR is a classical statistics-based classification method, which can be used in the binary classification problem or the multi-class classification problem. Given a new bug report, its probability of severity can be computed as follows:

$$P\left(y=k|x\right)=\frac{\exp\left(w_k x+b_k\right)}{1+\sum_{k=1}^{K-1}\exp\left(w_k x+b_k\right)}, \quad k=1,2 \tag{1}$$

where x represents the input embedding (i.e., the bug report embedding $Embed_{bug}$), and y represents the label (i.e., Severe or Non-Severe). In addition, w_k and b_k are the two parameters in this method.

In our experiment, we classify the severity into two types: Severe and Non-Severe. Therefore, the Severe probability of a bug report can be computed by Equation (2) and the Non-Server probability of a bug report can be computed by Equation (3):

$$P\left(y=1|x\right)=\frac{\exp\left(w_k x+b_k\right)}{1+\exp\left(w_k x+b_k\right)} \tag{2}$$

$$P\left(y=0|x\right)=\frac{1}{1+\exp\left(w_k x+b_k\right)} \tag{3}$$

The reasons we choose LR as the classifier to construct the bug report severity prediction model can be summarized as follows. First, the computational cost of using LR is low [42] when compared with the deep learning method. Second, previous work [43] showed that LR is more suitable than NB (Naive Bayesian) on large-scale datasets. Thirdly, LR can achieve promising performance in this task and the detailed analysis can be found in Section V-D.

B. Model Application Phase

Once the prediction model is trained, we can apply it to the new bug reports for predicting the severity. Given a new bug report, for the bug summary, we take the text preprocessing step. For the bug description, we take the text preprocessing step and keyword extraction step. Then based on the above results, we perform the word embedding step using the trained

FastText model to convert and concatenate embeddings as the bug report embedding $Embed_{bug}$. Finally, we can use the trained bug report severity prediction model to predict the severity of this new bug report.

IV. Experimental Setup

In our empirical study, we aim to evaluate our proposed method EKD-BSP with two goals. The first goal is to evaluate the performance of EKD-BSP by comparing the state-of-the-art baselines in the previous bug report severity prediction studies. The second goal is to analyze the impact of different component settings in our proposed method EKD-BSP.

Based on these two goals, we design the following four research questions (RQs):

RQ1: Can our proposed method EKD-BSP outperform the state-of-the-art baselines in the bug report severity prediction?

RQ2: Can extracting keywords from the bug description help to improve the performance of our proposed method EKD-BSP?

RQ3: Whether our proposed method can extract useful information from the bug description?

RQ4: How the choice of the classifier influences the performance of our proposed method EKD-BSP?

To answer these four RQs, we first collected two commonly-used large-scale open-source projects as our experimental subjects (in Section IV-A). Then we evaluate the performance of our proposed method EKD-BSP and the baselines in terms of performance measures (in Section IV-B). We follow the experimental setting to achieve the experimental results (in Section IV-C).

A. Experimental Subjects

We collected bug reports from two large-scale open-source projects (i.e., Eclipse and Mozilla). Specifically, Eclipse[4] is an integrated development environment (IDE) used in computer programming. Mozilla[5] is an open-source software project that contains several popular products (e.g., Firefox, Thunderbird). These tools have been widely used by developers, therefore, the quality of these bug reports can be guaranteed [3]. Moreover, these projects also have been widely used in previous studies on bug report severity prediction. In a recent survey [15], Gomes et al. found that the Eclipse project has been used in 92% of previous studies, while the Mozilla project has been used in 70% of previous studies.

We downloaded the bug reports of these two projects from the Bugzilla system by visiting URL[6], which includes bug ID, the bug summary, status, and the bug severity. Then we only select the bug reports, whose status is "Closed" or "Fixed" following Zhang et al.'s suggestion [33]. However, the bug description can not be directly downloaded from this URL. Therefore, we used the crawler to collect the bug description according to the Bug ID.

The severity of the bug report obtained from Bugzilla includes Blocker, Critical, Major, Normal, Minor, Trivial to Enhancement. The description of these severities can be found in Table II. Note that we remove the bug reports whose severity is Normal or Enhancement. The reasons can be summarized as follows. (1) The Normal is the default value of the bug report severity. For some novices, they tend to use this severity value directly. Therefore, this value can not truly reflect the actual severity of the bug report. (2) The bug reports with Enhancement severity mean these bug reports do not contain real bugs [3].

For the rest of the five severity, we categorize them into two classes, which is consistent with previous studies [6] [35]. Specifically, the severity Blocker, Critical or Major is classified as Severe type. While the severity Minor or Trivial is classified as Non-Severe type.

Finally, the statistical information of our chosen experimental subjects can be found in Table V. In this table, we show the number of bug reports for different severity for these two projects respectively. In summary, the Eclipse project contains 29,397 bug reports with Severe type and 9,485 bug reports with Non-Severe type. While for the Mozilla project, it contains 22,094 bug reports with Severe type and 8,951 bug reports with Non-Severe type.

B. Performance Measures

To evaluate the performance of EKD-BSP and baselines, we use $F\text{-}measure$, $Precision$ and $Recall$ as the performance measures. These performance measures have also been commonly used in previous bug report severity prediction studies [6] [15].

In this problem, we treat the bug reports with Severe type as the positive class and the bug reports with Non-Severe type as the negative class. Therefore, the confusion matrix for bug report severity prediction can be found in Table VI. In this confusion matrix, TP denotes the number of bug reports with Severe type, which are predicted correctly; FN denotes the number of bug reports with Severe type, which are predicted incorrectly; FP denotes the number of bug reports with Non-Severe type, which are predicted incorrectly; TN denotes the number of bug reports with Non-Severe type, which are predicted correctly.

$Precision$ returns the ratio of the number of the bug reports with the Severe type that are correctly classified as Severe type to the number of the bug reports that are classified as Severe type. It can be defined as:

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

$Recall$ returns the ratio of the number of the bug reports with the Severe type that are correctly classified as Severe type to the total number of the bug reports with the Severe type. It can be defined as:

[4]https://www.Eclipse.org/

[5]https://www.Mozilla.org/en-US/

[6]https://www.bugzilla.org/installation-list, the data of the Eclipse project is accessed in December 31, 2020 and the data of the Mozilla project is accessed in January 4, 2021.

TABLE V
THE STATISTICAL INFORMATION OF THE EXPERIMENTAL SUBJECTS

Project	Severe				Non-Severe		
	Blocker	*Critical*	*Major*	*Total*	*Minor*	*Trivial*	*Total*
Eclipse	3,816	6,939	18,642	29,397	6,912	2,573	9,485
Mozilla	2,641	11,350	8,103	22,094	5,792	3,159	8,951

TABLE VI
CONFUSION MATRIX FOR BUG REPORT SEVERITY PREDICTION

		Predicted	
		Severe	**Non-Severe**
Actual	**Severe**	$TP: True\ Positives$	$FN: False\ Negatives$
	Non-Severe	$FP: False\ Positives$	$TN: True\ Negatives$

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

There exists a trade-off between $Precision$ and $Recall$ in practice. In most cases, a higher value of $Precision$ means a lower value of $Recall$ and vice versa. Here we use $F\text{-}measure$, which is the harmonic mean between $Precision$ and $Recall$, to evaluate the performance of the constructed models. It can be defined as:

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

C. Experimental Setting

For each project, we take the first 80% of bug reports as the training data and the remaining 20% of bug reports as the test data in chronological order [36]. For the chronological order, we order these bug reports according to the last change time of these bug reports. Using this data split method, we can use historical bug reports to predict the severity of new bug reports, which can reflect the real application scenarios.

Since different parameter settings in our proposed method EKD-BSP can affect the model performance, we set the value of the parameters as follows. For the FastText model of word embedding step, each word is represented as a bag of character n-gram. In our study, we set it to 3 based on our preliminary exploration. In addition, the dimension of word embedding in the FastText model is set as 10 according to the suggestions provided by Joulin et al. [14]. For the remaining parameters, we use the default value.

Our experimental study is performed on the platform with the CPU Intel(R) Core(TM) i5-6300HQ, 24 GB memory, and Windows 10 Operation System.

V. RESULTS ANALYSIS

A. Result Analysis for RQ1

RQ1: Can our proposed method EKD-BSP outperform the state-of-the-art baselines in the bug report severity prediction?

To answer this RQ, we compare our proposed method EKD-BSP with the baselines in previous bug report severity prediction studies. According to a recent survey [15], 74% of the studies only considered the bug summary, and the classifiers NB (Naive Bayesian) and KNN (k-Nearest Neighbor) are top-2 classifiers used in previous studies (i.e., in 44% of the studies). Therefore, we consider NB and KNN as the first two baselines. Since EKD-BSP uses LR as the classifier, we further consider LR as the third baseline. Finally, in addition to the traditional classifiers (e.g., NB, KNN, LR), the deep learning method LSTM (Long Short-Term Memory) has also been used in a recent study [36]. Therefore, we consider LSTM as the fourth baseline.

Specifically, NB is a classification method that uses knowledge of probability and statistics. It decides to which class an instance belongs based on the Bayesian algorithm of conditional probability. NB assumes that given the class, the value of the features is independent of other features [4] [44] [45]. KNN is an instance-based lazy learning method. It processes the available instances (or neighbors) based on its similarity measure to the k-nearest neighbors [23] [46]. In general, KNN uses euclidean distance to calculate the proximity between instances. LSTM is a deep learning method. It is a special kind of RNN (Recurrent Neural Network), which is capable of learning long-term dependencies. LSTM uses four neural network layers, interacting in a very special way to achieve the classification [36].

For the baselines NB, KNN, and LR, the FastText model is used to represent the word embedding to ensure a fair comparison with our proposed method EKD-BSP. While for the baseline LSTM, its default embedding is used to realize the word vector representation.

The comparison results between our proposed method with four baselines can be found in Table VII. In this table, we can find that our proposed method EKD-BSP can achieve better performance than all the baselines. For the Eclipse project, EKD-BSP can achieve the performance of 70.43%, 73.81%, and 68.67% in terms of $F\text{-}measure$, $Precision$ and $Recall$ respectively. Compared with the baselines, the proposed method EKD-BSP can improve the performance by up to

5.19%, 8.57%, and 4.50% in terms of $F\text{-}measure$, $Precision$ and $Recall$ respectively. For the Mozilla project, EKD-BSP can achieve the performance of 80.09%, 82.01%, and 78.76% in terms of $F\text{-}measure$, $Precision$ and $Recall$ respectively. Compared with the baselines, the proposed method EKD-BSP can improve the performance by up to 4.54% and 7.18% in terms of $F\text{-}measure$ and $Precision$. Though EKD-BSP cannot achieve the best performance in terms of $Recall$, the performance of EKD-BSP is still in second place, and the gap with the best baseline NB is very small.

TABLE VII
COMPARISON RESULTS BETWEEN OUR PROPOSED METHOD EKD-BSP WITH FOUR BASELINE METHODS IN TERMS OF THREE PERFORMANCE MEASURES

Project	Method	F-Measure(%)	Precision(%)	Recall(%)
Eclipse	NB	66.02	65.24	68.05
	KNN	65.61	67.36	64.62
	LR	65.73	69.83	64.17
	LSTM	65.24	65.88	64.76
	EKD-BSP	**70.43**	**73.81**	**68.67**
Mozilla	NB	75.55	74.83	**79.42**
	KNN	75.90	77.09	75.02
	LR	76.58	78.66	75.24
	LSTM	75.77	76.10	75.48
	EKD-BSP	**80.09**	**82.01**	78.76

Summary for RQ1: EKD-BSP can outperform four state-of-the-art bug report severity prediction baselines.

B. Result Analysis for RQ2

RQ2: Can extracting keywords from the bug description help to improve the performance of our proposed method EKD-BSP?

Based on a recent survey, in the previous studies of the bug report severity prediction, 74% of the related studies only extract features from the bug summary [15]. In EKD-BSP, we use keywords extracted from the bug description to enhance the bug summary. Therefore, we design this RQ to investigate whether extracting keywords from the bug description can improve the performance of bug report severity prediction. In this RQ, we use $summ$ to denote the EKD-BSP method, which only considers the bug summary. Then we use $sumk$ to denote the EKD-BSP method, which considers both the bug summary and the keywords extracted from the bug description.

The comparison results between $summ$ and $sumk$ can be found in Table VIII. We can find in terms of three performance measures, the method $sumk$ can improve the performance for both two projects. Specifically, for the Eclipse project, the performance can be improved by 3.56%, 2.95% and 3.44% in terms of $F\text{-}measure$, $Precision$, $Recall$ respectively. For the Mozilla project, the performance can be improved by 3.45%, 3.27% and 3.46% respectively in terms of $F\text{-}measure$, $Precision$, $Recall$ respectively.

TABLE VIII
THE COMPARISON RESULTS BETWEEN $summ$ AND $sumk$

Project	Method	F-Measure(%)	Precision(%)	Recall(%)
Eclipse	$summ$	66.87	70.86	65.23
	$sumk$	**70.43**	**73.81**	**68.67**
Mozilla	$summ$	76.64	78.74	75.30
	$sumk$	**80.09**	**82.01**	**78.76**

Summary for RQ2: Additionally using the keywords extracted from the bug description can improve the performance of EKD-BSP.

C. Result Analysis for RQ3

RQ3: Whether our proposed method can extract useful information from the bug description?

The bug description is the supplement of the bug summary and it may contain useful information, which cannot be found in the bug summary. Based on the analysis result in RQ2, we can find using the keywords extracted from the bug description can help to improve the performance of our proposed method EKD-BSP. However, keywords are selected from the bug description, which may lose the information of the original bug description. Therefore, we want to investigate whether our proposed method can extract useful information from the bug description. In this RQ, we use $sumd$ to denote the EKD-BSP method, which considers both the bug summary and the original bug description. Then we use $sumk$ to denote the EKD-BSP method, which considers both the bug summary and the keywords extracted from the bug description.

We show the comparison results between $sumd$ and $sumk$ in Table IX. In this table, we can find in terms of three performance measures, the method $sumk$ can improve the performance for both two projects. Specifically, for the Eclipse project, the performance can be improved by 1.59%, 1.12% and 1.61% in terms of $F\text{-}measure$, $Precision$, $Recall$ respectively. For the Mozilla project, the performance can be improved by 1.64%, 1.45% and 1.70% respectively in terms of $F\text{-}measure$, $Precision$, $Recall$ respectively.

TABLE IX
THE COMPARISON RESULTS BETWEEN $sumd$ AND $sumk$

Project	Method	F-Measure(%)	Precision(%)	Recall(%)
Eclipse	$sumd$	68.84	72.69	67.06
	$sumk$	**70.43**	**73.81**	**68.67**
Mozilla	$sumd$	78.45	80.56	77.06
	$sumk$	**80.09**	**82.01**	**78.76**

Except for performance comparison, we also analyze the ratio of selected words after extracting keywords from the bug description. Supposing the original bug description contains num_d words and extracted keywords contain num_k words, the ratio is num_k/num_d. Fig. 3 shows the distribution of the ratio of the selected words for these two projects. In this figure,

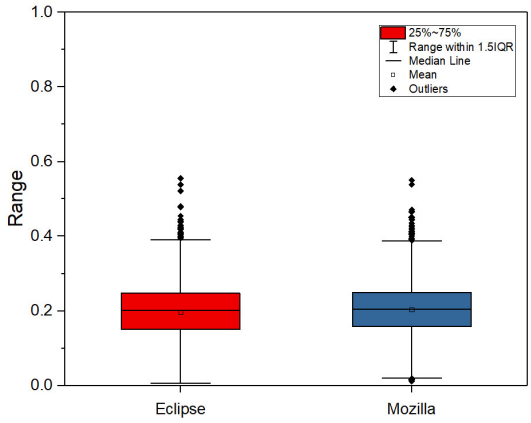

Fig. 3. The distribution of the ratio of the selected words for two projects

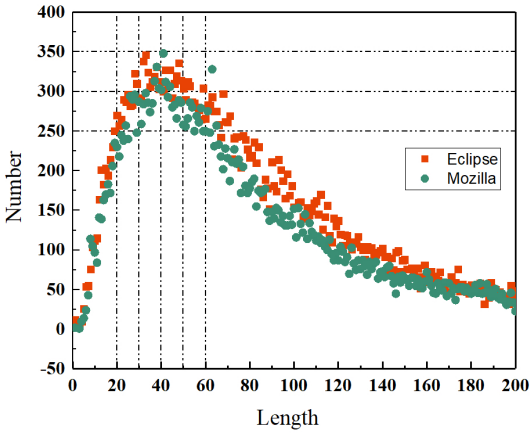

Fig. 4. The word number distribution of the bug description for two projects

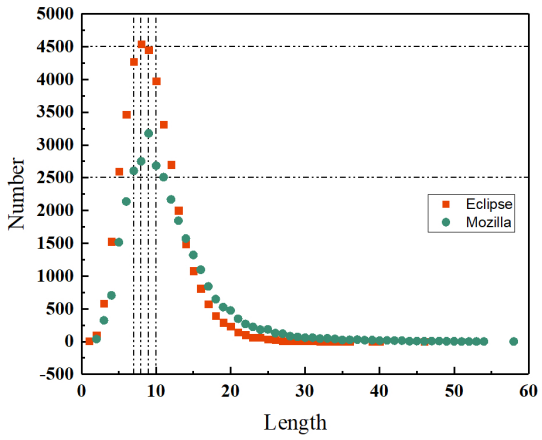

Fig. 5. The word number distribution of the bug summary for two projects

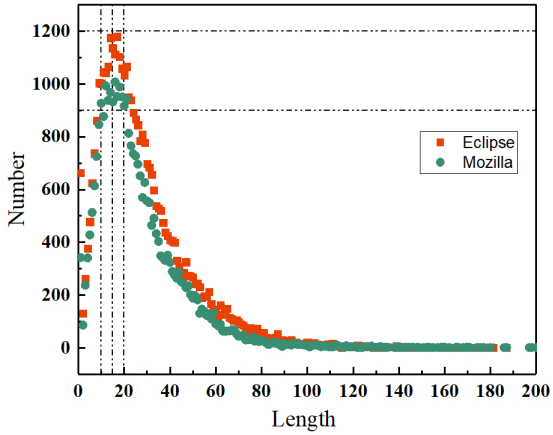

Fig. 6. The word number distribution of the keywords extracted from the bug description for two projects

we can find that the ratio of the selected words after extracting keywords is about 20%.

Then we use the scatter chart to analyze the word number distribution of the bug summary, the bug description, and the keywords extracted from the bug description. The final results can be found in Fig. 4, Fig. 5, and Fig. 6. In these figures, the horizontal axis shows the word number, and the vertical axis shows the number of corresponding bug reports.

Fig. 4 shows the word number distribution of the bug description. In this figure, we limit the maximum value of the word number to 200, since the word number of some bug descriptions can even up to 8000. It can be found that in the bug description, most of the bug reports range from 20 to 60. Fig. 5 shows the word number distribution of the bug summary. In this figure, we can find most of the bug reports ranges from 7 to 10. Fig. 6 shows the word number distribution of the keywords extracted from the bug description. In this figure, we can find most of the bug reports contains about 15 words. For the Eclipse project, the number of bug reports with 15 words is about 1200. For the Mozilla project, the number of bug reports with 15 words is about 1000. Moreover, when the length of keywords is more than 20, there is a linear downward trend, which is similar to the distribution of the bug summary. Therefore, after keyword extraction, the word number distribution of the bug description is similar to the word number distribution of the bug summary.

Summary for RQ3: Using keyword extraction (i.e., only keep 20% of words in the original bug description) can help to extract useful information and then improve the performance of EKD-BSP.

D. Result Analysis for RQ4

RQ4: How the choice of the classifier influences the performance of our proposed method EKD-BSP?

In our proposed method EKD-BSP, the classifier has a non-ignorable impact on the performance of the constructed bug report severity prediction model. In RQ4, we consider three different classifiers (i.e., NB, KNN, and LR) for EKD-BSP. To guarantee a fair comparison, we use the same FastText

model to represent the word embedding and use the same experimental setup.

The comparison results of using different classifiers can be found in Table X. From Table X, we can find that the LR classifier can achieve the best performance in terms of $F\text{-}measure$ and $Precision$. Specifically, in terms of $F\text{-}measure$, the classifier LR can improve the performance at most 2.80% and 2.21% for the Eclipse project and the Mozilla project respectively. In terms of $Precision$, the classifier LR can improve the performance at most 6.51% and 4.94% for the Eclipse project and the Mozilla project respectively. Though in terms of $Recall$, the classifier LR cannot achieve the best performance. However, LR can achieve the best performance in $Precision$ and then result in the best performance in $F\text{-}measure$.

TABLE X
THE COMPARISON RESULTS OF USING DIFFERENT CLASSIFIERS FOR EKD-BSP

Project	Classifier	F-Measure(%)	Precision(%)	Recall(%)
Eclipse	NB	67.63	67.30	**72.45**
	KNN	69.58	73.00	67.86
	LR	**70.43**	**73.81**	68.67
Mozilla	NB	77.88	77.07	**82.14**
	KNN	78.98	80.89	77.66
	LR	**80.09**	**82.01**	78.76

Summary for RQ4: By using LR as the classifier, EKD-BSP can achieve the best performance.

VI. THREATS TO VALIDITY

In this section, we identify potential threats, which may influence the validity of our empirical results. These threats include internal threats, construct threats, and external threats.

Internal Validity. The internal validity is the potentials defects in the implementation of our proposed method. To alleviate this threat, we performed code inspection and software testing. Moreover, we also used mature libraries (such as NLTK[7] and skit-learn[8].

Construct Validity. The construct validity is the performance measures for evaluating the performance of our proposed method. To alleviate this threat, we choose three performance measures (i.e., $F\text{-}Measure$, $Precision$, and $Recall$), which have been commonly used in previous bug report severity prediction studies.

External threats. The first external validity is the choice of experimental subjects. To alleviate this threat, we use the two most widely used real-world open-source subjects (i.e., Eclipse and Mozilla). The reasons can be summarized as follow. First, they were used in previous bug report severity prediction studies [3] [10] [15] [32]. Second, these two projects are popular and have a large number of high-quality bug reports. In the future, we will use other projects (such as OpenOffice, Netbeans) to verify the effectiveness of our proposed method. The second external validity is the setting of parameters in our proposed method. To alleviate this threat, we set the parameter value based on our preliminary exploration or based on the suggestions by previous studies [14].

[7]http://www.nltk.org/

[8]https://scikit-learn.org/stable/

VII. CONCLUSION AND FUTURE WORK

Through manual observation, we find that the bug summary of some bug reports may contain a few words or may not contain enough useful information, which is not helpful to construct high-quality bug report severity prediction models. While the bug description contains detailed information. Therefore, We propose a novel method EKD-BSP, which uses the bug summary and the keywords extracted from the bug description to perform bug report severity prediction. We use two large-scale open-source subjects (i.e., Eclipse and Mozilla) to evaluate the performance of our proposed method. The results show that EKD-BSP can effectively predict the severity of bug reports. Specifically, it can achieve 70.43% $F\text{-}measure$ for Eclipse and 80.09% $F\text{-}measure$ for Mozilla by only keeping 20% of words in the original bug description. Moreover, we also show the competitiveness of extracting keywords from the bug description and using the LR as the classifier in EKD-BSP.

In the future, we first want to evaluate the performance of our proposed method by considering more large-scale commercial and open-source projects. Second, we want to further improve the performance of our proposed method by using more advanced word representation methods. Third, we want to further model the bug report severity prediction as a multi-classification problem, which is a more challenging research problem. Its practicability may be slightly higher than modeling this issue as a binary classification problem since bug reports often have multiple severities.

ACKNOWLEDGMENT

This work is supported in part by the National Natural Science Foundation of China (Grant No. 61872263) and The Open Project of Key Laboratory of Safety-Critical Software for Nanjing University of Aeronautics and Astronautics, Ministry of Industry and Information Technology (Grant No. NJ2020022)

REFERENCES

[1] R. K. Saha, S. Khurshid, and D. E. Perry, "Understanding the triaging and fixing processes of long lived bugs," *Information and software technology*, vol. 65, pp. 114–128, 2015.

[2] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 346–355.

[3] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 1–10.

[4] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, 2011, pp. 249–258.

[5] M. Pushpalatha and M. Mrunalini, "Predicting the severity of open source bug reports using unsupervised and supervised techniques," *International Journal of Open Source Software and Processes (IJOSSP)*, vol. 10, no. 1, pp. 1–15, 2019.

[6] W. Y. Ramay, Q. Umer, X. C. Yin, C. Zhu, and I. Illahi, "Deep neural network-based severity prediction of bug reports," *IEEE Access*, vol. 7, pp. 46 846–46 857, 2019.

[7] 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 *2012 19th Asia-Pacific Software Engineering Conference*, vol. 1. IEEE, 2012, pp. 240–249.

[8] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry, "Are these bugs really" normal"?" in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 258–268.

[9] S. Chen, X. Xie, B. Yin, Y. Ji, L. Chen, and B. Xu, "Stay professional and efficient: Automatically generate titles for your bug reports," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 385–397.

[10] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 215–224.

[11] A. F. Otoom, D. Al-Shdaifat, M. Hammad, and E. E. Abdallah, "Severity prediction of software bugs," in *2016 7th International Conference on Information and Communication Systems (ICICS)*. IEEE, 2016, pp. 92–95.

[12] G. Yang, S. Baek, J.-W. Lee, and B. Lee, "Analyzing emotion words to predict severity of software bugs: A case study of open source projects," in *Proceedings of the Symposium on Applied Computing*, 2017, pp. 1280–1287.

[13] R. Mihalcea and P. Tarau, "Textrank: Bringing order into text," in *Proceedings of the 2004 conference on empirical methods in natural language processing*, 2004, pp. 404–411.

[14] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," *arXiv preprint arXiv:1607.01759*, 2016.

[15] L. A. F. Gomes, R. da Silva Torres, and M. L. Côrtes, "Bug report severity level prediction in open source software: A survey and research opportunities," *Information and software technology*, vol. 115, pp. 58–78, 2019.

[16] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei, "A survey on bug-report analysis," *Science China Information Sciences*, vol. 58, no. 2, pp. 1–24, 2015.

[17] R. Almhana and M. Kessentini, "Considering dependencies between bug reports to improve bugs triage," *Automated Software Engineering*, vol. 28, no. 1, pp. 1–26, 2021.

[18] G. Yang, T. Zhang, and B. Lee, "Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports," in *2014 IEEE 38th Annual Computer Software and Applications Conference*. IEEE, 2014, pp. 97–106.

[19] M. Sharma, M. Kumari, R. Singh, and V. Singh, "Multiattribute based machine learning models for severity prediction in cross project context," in *International Conference on Computational Science and Its Applications*. Springer, 2014, pp. 227–241.

[20] K. Jin, E. C. Lee, A. Dashbalbar, J. Lee, and B. Lee, "Utilizing feature based classification and textual information of bug reports for severity prediction," *International Information Institute (Tokyo). Information*, vol. 19, no. 2, p. 651, 2016.

[21] C.-Z. Yang, K.-Y. Chen, W.-C. Kao, and C.-C. Yang, "Improving severity prediction on software bug reports using quality indicators," in *2014 IEEE 5th International Conference on Software Engineering and Service Science*. IEEE, 2014, pp. 216–219.

[22] K. K. Sabor, M. Nayrolles, A. Trabelsi, and A. Hamou-Lhadj, "An approach for predicting bug report fields using a neural network learning model," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2018, pp. 232–236.

[23] K. K. Sabor, M. Hamdaqa, and A. Hamou-Lhadj, "Automatic prediction of the severity of bugs using stack traces and categorical features," *Information and Software Technology*, vol. 123, p. 106205, 2020.

[24] X. Chen, Z. Yuan, Z. Cui, D. Zhang, and X. Ju, "Empirical studies on the impact of filter-based ranking feature selection on security vulnerability prediction," *IET Software*, vol. 15, no. 1, pp. 75–89, 2021.

[25] C. Ni, X. Chen, F. Wu, Y. Shen, and Q. Gu, "An empirical study on pareto based multi-objective feature selection for software defect prediction," *Journal of Systems and Software*, vol. 152, pp. 215–238, 2019.

[26] C. Ni, W.-S. Liu, X. Chen, Q. Gu, D.-X. Chen, and Q.-G. Huang, "A cluster based feature selection method for cross-project software defect prediction," *Journal of Computer Science and Technology*, vol. 32, no. 6, pp. 1090–1107, 2017.

[27] W. Liu, S. Liu, Q. Gu, J. Chen, X. Chen, and D. Chen, "Empirical studies of a two-stage data preprocessing approach for software fault prediction," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 38–53, 2015.

[28] N. K.-S. Roy and B. Rossi, "Towards an improvement of bug severity classification," in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2014, pp. 269–276.

[29] G. Sharma, S. Sharma, and S. Gujral, "A novel way of assessing software bug severity using dictionary of critical terms," *Procedia Computer Science*, vol. 70, pp. 632–639, 2015.

[30] W. Liu, S. Wang, X. Chen, and H. Jiang, "Predicting the severity of bug reports based on feature selection," *International Journal of Software Engineering and Knowledge Engineering*, vol. 28, no. 04, pp. 537–558, 2018.

[31] J. Gou, L. Du, Y. Zhang, T. Xiong *et al.*, "A new distance-weighted k-nearest neighbor classifier," *J. Inf. Comput. Sci*, vol. 9, no. 6, pp. 1429–1436, 2012.

[32] T. Zhang, G. Yang, B. Lee, and A. T. Chan, "Predicting severity of bug report by mining bug repository with concept profile," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 2015, pp. 1553–1558.

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

[34] Y. Tian, N. Ali, D. Lo, and A. E. Hassan, "On the unreliability of bug severity data," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2298–2323, 2016.

[35] T.-L. Zhang, R. Chen, X. Yang, and H.-Y. Zhu, "An uncertainty based incremental learning for identifying the severity of bug report," *International Journal of Machine Learning and Cybernetics*, vol. 11, no. 1, pp. 123–136, 2020.

[36] Y. Tan, S. Xu, Z. Wang, T. Zhang, Z. Xu, and X. Luo, "Bug severity prediction using question-and-answer pairs from stack overflow," *Journal of Systems and Software*, vol. 165, p. 110567, 2020.

[37] M. Kumari, M. Sharma, and V. Singh, "Severity assessment of a reported bug by considering its uncertainty and irregular state," *International Journal of Open Source Software and Processes (IJOSSP)*, vol. 9, no. 4, pp. 20–46, 2018.

[38] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[39] S. Gujral, G. Sharma, S. Sharma *et al.*, "Classifying bug severity using dictionary based approach," in *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*. IEEE, 2015, pp. 599–602.

[40] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[41] X. Chen, C. Chen, D. Zhang, and Z. Xing, "Sethesaurus: Wordnet in software engineering," *IEEE Transactions on Software Engineering*, 2019.

[42] S. Vora and T. Kurzweg, "Modified logistic regression algorithm for accurate determination of heart beats from noisy passive rfid tag data," in *2016 IEEE-EMBS International Conference on Biomedical and Health Informatics (BHI)*. IEEE, 2016, pp. 29–32.

[43] A. Y. Ng and M. I. Jordan, "On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes," in *Advances in neural information processing systems*, 2002, pp. 841–848.

[44] S. S. Alia, M. N. Haque, S. Sharmin, S. M. Khaled, and M. Shoyaib, "Bug severity classification based on class-membership information," in *2018 Joint 7th International Conference on Informatics, Electronics & Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*. IEEE, 2018, pp. 520–525.

[45] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[46] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is "nearest neighbor" meaningful?" in *International conference on database theory*. Springer, 1999, pp. 217–235.