Research Article

Mining Bug Report Repositories to Identify Significant Information for Software Bug Fixing

Bancha Luaphol*, Jantima Polpinij and Manasawee Kaenampornpan
Department of Computer Science, Faculty of Informatics, Mahasarakham University, Maha Sarakham, Thailand

* Corresponding author. E-mail: bancha.lu@ksu.ac.th
DOI: 10.14416/j.asep.2021.03.005
Received: 6 December 2020; Revised: 20 January 2021; Accepted: 5 February 2021; Published online: 17 March 2021

Abstract
Most studies relating to bug reports aim to automatically identify necessary information from bug reports for software bug fixing. Unfortunately, the study of bug reports focuses only on one issue, but more complete and comprehensive software bug fixing would be facilitated by assessing multiple issues concurrently. This becomes a challenge in this study, where it aims to present a method of identifying bug report at a severe level from a bug report repository, together with assembling their related bug reports to visualize the overall picture of a software problem domain. The proposed method is called "*mining bug report repositories*". Two techniques of text mining are applied as the main mechanisms in this method. First, classification is applied for identifying severe bug reports, called "*bug severity classification*", while "*threshold-based similarity analysis*" is then applied to assemble bug reports that are related to a bug report at a severe level. Our datasets are obtained from three opensource namely SeaMonkey, Firefox, and Core:Layout downloaded from the Bugzilla. Finally, the best model from the proposed method is selected and compared with two baseline methods. For identifying severe bug reports using classification technique, the results show that our method improved accuracy, F1, and AUC scores over the baseline by 11.39, 11.63, and 19% respectively. Meanwhile, for assembling related bug reports using threshold-based similarity technique, the results show that our method improved precision, and likelihood scores over the other baseline by 15.76, and 9.14% respectively. This demonstrate that our proposed method may help to increase the chance to fix bugs completely.

Keywords: Bug report, Severe bug report, Related bug report, Text mining, Classification, Similarity analysis

1 Introduction

Bug reports contain all significant information for helping a development team to find and fix problems occurred in software. To gather bug reports on a global scale, bug tracking systems (BTSs) have been developed and proposed, where BTSs are able to gather bug reports from people from around the world. After gathering bug reports from end-users worldwide, software experts, so called bug triagers, are required to analyze the bug reports, which includes classifying bug and non-bug reports, checking for duplicated bug reports, prioritizing bug reports, and assigning suitable developers. These tasks are time-consuming and costly [1]–[3]. As a result, there are subsequently a large number of bug report studies, and generally they can be divided into three main areas.

The first area of bug report studies concerns bug report optimization which aims to enhance report quality and reduce the amount of incorrect information in reports. Bug report optimization can be classified into three tasks: Content optimization [4], [5]; bug report misclassification [6]–[10]; and severity prediction [11]–[15]. Yet, the most important task in bug report optimization is the bug report misclassification. This is because analysis time in the following stage can increase if outlier bug reports are not removed from the bug reports. The second study in the bug report area is the report triage which concerns duplicated bug detection [1], [16]–[19], prioritization [20]–[23], and suitable developer assignment tasks [24], [25]. Duplicated bug reports are detected and removed from the bug report repository where further processing is not required [1], [17], [18]. Bug report prioritization serves to predict the priority of bug reports and assigning suitable developers for fixing the bug is the last task of the bug report triage. The third study in the bug report area is bug fixing, which can also be divided into three main tasks: bug localization [26]–[28]; recovering links between bug reports and change files [29], [30]; and bug fixing time prediction [31], [32]. Bug localization aims to identify the location of bugs in software code or in a program. Recovering links between bug reports serves to connect bug reports and change files, where the change files are the logs of the software correction history. In this case, when software is corrected according to a specific bug report, the report must be linked to its related log. However, some links may be missing, so this task also seeks to recover the links between the bug reports and the change files. Finally, predicting the bug fixing time seeks to identify how long it will take between identifying a bug and resolving it.

In fact, most studies mentioned above aims to automatically identify necessary information from bug reports for software bug fixing. Bug severity analysis is an important process that involves estimating the impact of the bug on software according to a ranking scale. Severity is a measure of the seriousness of a software issue and how it terribly affects functionality. Consequently, bug triagers often look for bug reports that contain the most severity, especially severity information at "*blocker*" and "*critical*" levels [12]–[15]. Software bugs involved in blockers can impact further testing in a specific environment, while bugs at a critical level result in software crashing, data loss, or other serious damage. Bug reports at blocker or critical levels are defined as "*severe bug reports*". Many studies have proposed automatic bug severity analysis, mostly driven by text classification, called "*bug severity classification*" [13]–[15].

However, bug severity analysis alone cannot help to obtain sufficient information for completely fixing software bugs because the development team may not see the overall picture of a software problem domain. A solution to see the overall picture of a software problem domain is to find all related bug reports that are called "*related bug reports*" or "*bug report dependency*". Related bug reports can be described by a situation in which an unfixed bug '*a*' affects bug '*b*'. That is, bug '*b*' continues to occur despite it being fixed if bug '*a*' is not yet completely fixed. Unfortunately, this issue has not yet been earnestly studied. It was just mentioned in [2], [32]. This may be because performance improvements are still required for bug report misclassification, severity and priority prediction, bug duplicated detection, bug localization, and bug fixing tasks [3]. Therefore, finding a solution to see the overall picture of a software problem domain is a challenge in this study.

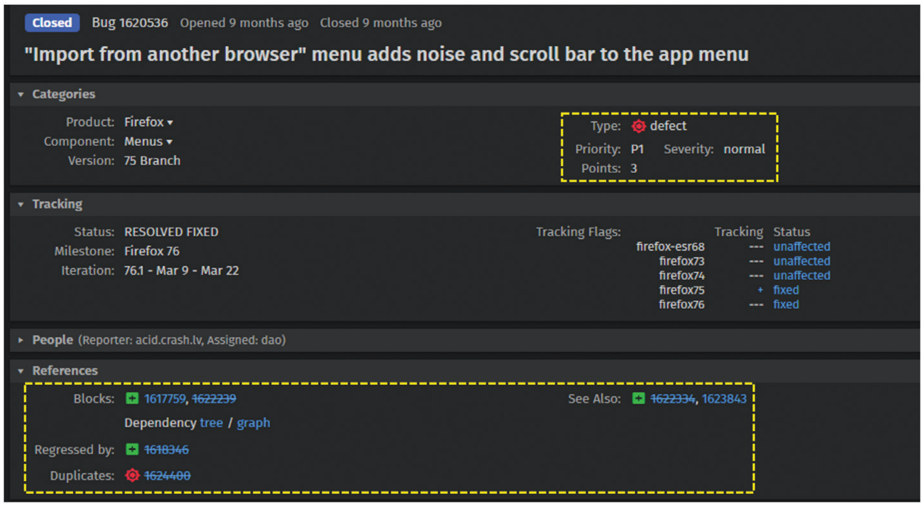
This study aims to present a method of identifying severe bug reports from a bug report repository, together with assembling their related bug reports to visualize the overall picture of a software problem domain. The proposed method is called "*mining bug report repositories*". Two techniques of text mining are applied as the main mechanisms. First, classification is applied for bug severity analysis, called "*bug severity classification*", while text similarity is then applied to assemble related bug reports, called "*threshold-based similarity analysis*". In classification tasks, machine learning and deep learning algorithms are compared to obtain the most appropriate models. We selected *multinomial naïve bayes* (*MNB*), *support vector machines* (*SVM*), *random forest* (*RF*), and *convolutional neural networks* (*CNN*). Furthermore, to increase the class distinguishing power, a *supervised term weighting* (*STW*) scheme, called *term frequency - inverse gravity moment* (*tf-igm*), is applied because this weighting scheme can determine the importance of a word in a document of a specific class.

Finally, the best model from our "*bug severity classification*" method is compared with the baseline method proposed by Ramay *et al.* [15], while the best model of the "*threshold-based similarity analysis*" method is compared with the baseline method proposed by Rocha *et al.* [33].

The paper is organized as follows. In section 2, we present the datasets used for this study. Meanwhile, the proposed method is presented in section 3 and the experimental results are presented in section 4. Finally, the conclusion is in section 5.

2 Dataset

The dataset used here was gathered from Bugzilla,



Closed Bug 1620536 Opened 9 months ago Closed 9 months ago

"Import from another browser" menu adds noise and scroll bar to the app menu

▾ Categories

Product:	Firefox ▾	Type:	defect
Component:	Menus ▾	Priority:	P1 Severity: normal
Version:	75 Branch	Points:	3

▾ Tracking

Status: RESOLVED FIXED
Milestone: Firefox 76
Iteration: 76.1 - Mar 9 - Mar 22

Tracking Flags:

	Tracking	Status
firefox-esr68	---	unaffected
firefox73	---	unaffected
firefox74	---	unaffected
firefox75	+	fixed
firefox76	---	fixed

▸ People (Reporter: acid.crash.lv, Assigned: dao)

▾ References

Blocks: 1617759, ~~1622239~~ See Also: ~~1622334~~, 1623843
Dependency tree / graph
Regressed by: ~~1618346~~
Duplicates: ~~1624400~~

Figure 1: An example of bug report and its severe level and related bug reports.

while bug reports relating to Mozilla were downloaded between 1 September 2019 and 30 October 2020. Our dataset is from three opensource namely SeaMonkey, Firefox, and Core:Layout. The dataset consists of 66,989 bug reports. Here, the bug report statuses used in this study are '*verified*' and '*closed*' because bug reports with these statuses were confirmed by bug triagers, software developers, and software testers that might be "*real*" bug reports [3], [21].

In general, a bug report contains three major parts, i.e. summary, description, and discussion. The summary is the title of the bug report, while the description contains details of each particular bug report. The discussion contains information concerning mentions or comments on that particular bug report submitted by other end users. However, many bug report studies deploy only the summary because this part contains less noise [17], [34], [35]. Therefore, here, we also investigated only the summary part.

Bug reports labeled as "*blocker*", "*critical*" and "*major*" are re-assigned as severe bug reports, while bug reports labeled as "*normal*", "*minor*" or "*trivial*" are re-assigned as non-severe bug reports. However, when we downloaded bug reports from Bugzilla, none were labeled as "*major*". Therefore, major bug reports were not utilized in our study.

Each bug report is also assigned information to indicate its associated bug reports, labeled as "*depends on*" (Figure 1). This dataset was used in both the proposed and compared methods, and these were set in the same environment. The summary of our dataset is presented Table 1.

Table 1: Summary of dataset

Dataset	Total of Bug Reports	Number of Bug Reports	
		Severe	Non-severe
Core:Layout	9,840	1,067	8,773
Firefox	36,324	1,762	34,562
SeaMonkey	20,825	2,643	18,182

It notices that 70% of the available data in the severe class is allocated for training. The remaining 30% of data are referred to test datasets. To prevent a problem of imbalance class, the number of non-severe bug reports at the test set should be equal to the number of severe bug reports at the training set.

3 The Proposed Method

An overview of the proposed method, called "*mining bug report repositories*", is shown as Figure 2. It consists of two main stages. The first stage is to select bug reports that are deemed to a severe level utilizing text classification technique, called "*identifying severe-bug report*" stage. The second stage is to assemblage bug reports related to those severe bug reports using threshold-based text similarity analysis, called "*assembling related bug reports*". Each stage can be detailed as follows.

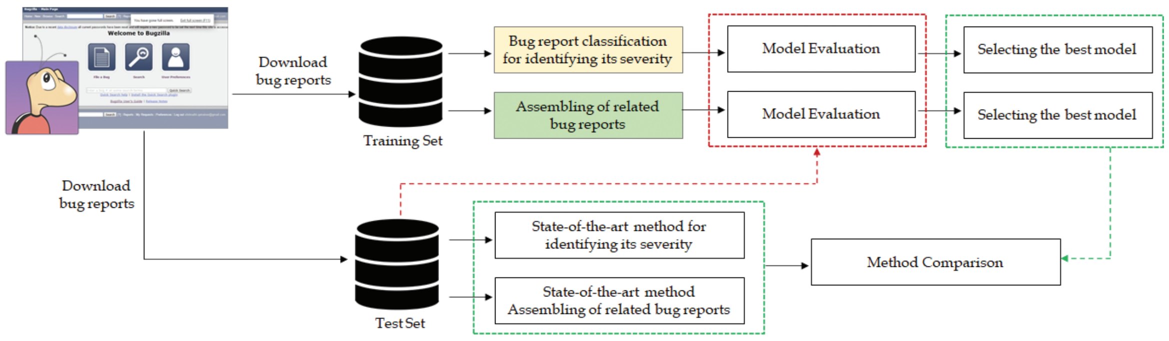

Download
bug reports
Training Set
Bug report classification for identifying its severity
Assembling of related bug reports
Model Evaluation
Model Evaluation
Selecting the best model
Selecting the best model
Download
bug reports
Test Set
State-of-the-art method for identifying its severity
State-of-the-art method Assembling of related bug reports
Method Comparison

Figure 2: Overview of the mining bug report repositories method.

3.1 *Identifying severe bug reports using severity classification*

This stage is to apply the text classification technique to identify bug reports that are deemed to the severe level. It consists of three main processing steps. They are pre-processing, bug report representation, and term weighting, and severe bug classifier modeling. An overview of the proposed methodology for identifying severe bug reports can be shown in Figure 3. Each processing step can be described as follows.

3.1.1 Pre-processing

The first stage of bug report pre-processing is text tokenization. This process separates text to tokens, called "*words*" in this study. Bug report features (or words) used in this study are unigram and CamelCase. Unigram means a single word, while CamelCase [6], [26]–[28], [36] (also referred to as Snakecase or Compound words) refers to words that combine many single words or abbreviations with no intervening spaces or punctuation. Some examples are "*browser_views*" and "*AutoComplete*". Unigram and CamelCase are popularly used in bug report studies because a unigram is simple to extract from a bug report, while CamelCase can indicate the specificity of the software. However, when using CamelCase words, these words are split into single words before use. This expands the bug report features. Therefore, examples of CamelCase words such as "*browser_views*" and "*AutoComplete*" can be split as "*browser*", "*views*", "*Auto*", and "*Complete*", respectively. It is noted that both original CamelCase words and words that are split from those CamelCase words are used in this study.

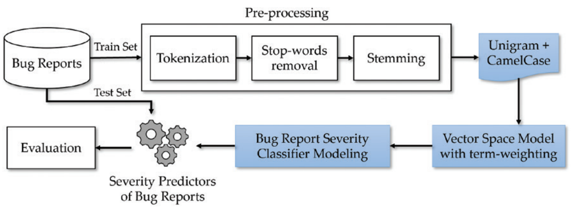

Pre-processing
Bug Reports
Train Set
Tokenization
Stop-words removal
Stemming
Unigram + CamelCase
Test Set
Evaluation
Severity Predictors of Bug Reports
Bug Report Severity Classifier Modeling
Vector Space Model with term-weighting

Figure 3: Overview of the proposed methodology for identifying severe bug reports.

After tokenizing text to words, the stop-words are removed. This is followed by the stemming process. In this case, the Snowball stemmer is utilized to reduce inflection in words to their common base, stem, or root form. It is noted that we performed bug report pre-processing using the Natural Language Tool Kit (NLTK) library in Python.

Suppose there is an example of the summary part of a bug report. It is "*AutoComplete for URLs isn't working (urlbar)*". An example of pre-processing bug report can be illustrated in Table 2.

Table 2: An example of pre-processing bug report

Processing Tasks	Result of each Step
Tokenized	AutoComplete/ for/ URLs/ is/ n't/ working/ ./ (/ urlbar/)
Stop-words removal	AutoComplete/ URLs/ working/ urlbar
CamelCase words splitting	AutoComplete/ Auto/ Complete/ URLs/ working/ urlbar
Stemming	autocomplet/ auto/ complet/ url/ work/ urlbar

3.1.2 Bug report representation and term weighting

After obtaining features of bug reports, they are represented as vector space model (VSM) format.

Later, *term frequency - inverse gravity moment* (*tf-igm)* which is one of supervised term weighting (STW) schemes is used to assign weight score for each bug report feature [37]. The specific character of STW is to consider term distribution in the classes of interest. Consequently, this may help to improve its discriminating power for text classification tasks [37], [38]. The *tf-igm* is to combine *term frequency* (*tf*) with the igm measure. The equation of *tf-igm* is represented in Equation (1).

$$tf-igm(t_i)=tf(t_i,d_i)\times(1+\lambda\times igm(t_i)) \tag{1}$$

The *igm* is indicated in Equation (2), where f_{ir} (r = 1,2,...,M) is the number of bug reports containing the word t_i in the ***r***-th class, which are sorted in descending order. Thus, f_{i1} represents the frequency of word t_i in the class in which it occurs most often. The equation of *igm* is represented in Equation (2).

$$igm(t_i)=\frac{f_{i1}}{\sum_{r=1}^{M} f_{ir}\times r} \tag{2}$$

In Equation (1), λ is an adjustable coefficient used to relatively balance between the global and local factors in the weight of a term. To the best of our knowledge, the λ coefficient should have a default value of 7.0. However, it and can be set as a value between 5.0 and 9.0 [37].

3.1.3 Severe bug classifier modeling

To build the severe bug report classifiers, suppose *BR* is bug reports allocated as training set and *br* is a bug report. This can be denoted as $BR = \{b_{r1},b_{r2},...,b_{ri}\}$. A fixed set of classes can be denoted as *C* = {*severe, non-severe*}. In this study, four classification algorithms are applied. These algorithms can be briefly described.

- *Multinomial Naïve Bayes (MNB)*

The Naïve Bayes (NB) is a classification algorithm which refers to conditional independence of each of the bug report features in the classifier model. However, this algorithm was improved by adding Laplacian or add-one smoothing to prevent the zero probability for an unseen word and this NB version is called multinomial naïve bayes (MNB). The MNB classifier is a specific instance of an NB classifier that uses a multinomial distribution for each of the features [39]. From the training set, we can calculate as follows [Equation (3)].

$$P(c_j)=\frac{c_j}{\sum_{i=1}^{n} c_n} \tag{3}$$

While the equation of $P(w_i \mid c)$ is:

$$P(w_i \mid c)=\frac{count(w_i,c)+1}{\sum_{w\in V}(count(w,c)+1)} \tag{4}$$

where count(w_i, c) is the total count of word i in all documents of class c of the training set, and $|V|$ represents the entire words found in the document. It can be seen that Equation (4) uses Laplacian or add-one smoothing to prevent the zero probability for an unseen word.

- *Support Vector Machine (SVM)*

The SVM is a popular algorithm for text classification. It determines a decision boundary together with a maximal margin to separate almost all the documents into two classes. The SVM has returned good results in many studies. In SVM learning, the classification problem involves finding a separating hyperplane that maximizes "*the margin*" [40]. This technique allows for errors in classification using "*slack-variables*", and also, operates as a "*dual problem*" that only depends on inner products between feature vectors which can be replaced with kernels [40]. In SVM learning, a kernel function uses an infinite number of features for pattern analysis [41], [42].

- *Random Forest (RF)*

RF is also a popular classification method. This comprises an ensemble of a set of trees as a learning classification method [43]. RF performs by building a lot of decision trees at training time. Each node in the decision tree conducts on a random subset of features to generate the output. Finally, the random forest aggregates the output of individual decision trees to a summary as the final output. One of the most popular forest construction procedures proposed by [43]. In this study, 100 trees were constructed.

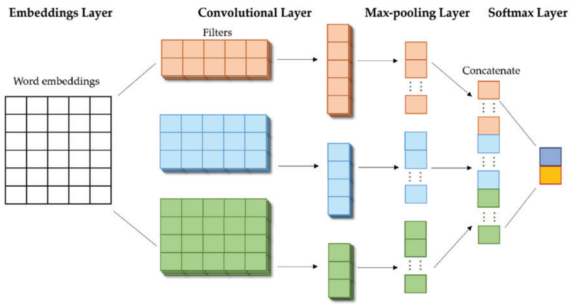


Figure 4: Overview of general CNN architecture for text classification.

- *Convolutional Neural Network (CNN)*

CNN is a class of deep neural networks (DNN) that is most commonly applied to image analytics. CNN has also been applied to text classification. It has proved useful for this task. The architecture of CNN for text classification consists of four connected layers as word embedding layer, convolutional layer, pooling layer, and softmax layer, as shown in Figure 4.

Each layer of CNN for text classification can be described as follows.

Word Embedding Layer: Word embedding is the first layer in CNN. This process is performed to map vocabulary word indices to low dimensional vectors by transforming natural language into a meaningful numerical form. Word embeddings are represented as vectors, and each vector depicts the features of a word. The closeness of two words embedding vectors in the vector space indicates the degree to which they are semantically related. A word embedding vector is learned for every word in all of the texts included in the text corpus.

Convolutional Layer: This layer converts the texts to sequences of word embeddings as input, and then creates feature vectors by analyzing the word embeddings for each text using a mechanism called "*convolution filters*". A convolution filter is a matrix filled with weights that analyzes multiple consecutive words in a text concurrently. This process continues throughout the whole text to create a feature map. The same operation is performed for every text to detect different relationships between the words using multiple convolution filters. These convolution filters also differ from each other in height, which indicates how many consecutive words a filter considers concurrently in each step. To obtain the feature vectors, the feature maps generated by the convolution operation are added with a bias term, and an activation function is also applied to add non-linearity.

Pooling Layer: This layer used the variable-length feature vectors obtained from the convolutional layer as input and produces fixed-length vectors. By doing this, the less relevant local information should be removed.

Softmax Layer: It is the final layer of CNN used to convert the fixed length feature vectors to be the input to a fully-connected layer as an efficient way of learning non-linear feature combinations. Outputs of this fully connected layer are numerical values for each class. To assign a straightforward interpretation to these numbers, the softmax function is applied to force the output of the CNN to represent predicted probabilities for each of the classes. Finally, the class achieving the highest predicted probability is the predicted class generated from the CNN.

During the training of the CNN, the weights in the embedding, convolutional, and softmax layers are updated in each epoch using the categorical cross-entropy loss function. This process of updating the weights is called "*back-propagation*", and is the essence of neural network training. Back-propagation is used to fine-tune the weights of a neural network based on the error rate resulting from the previous epoch. Minimizing the error rates through proper tuning can increase model reliability and generalization.

In this study, we used three layers of CNN with the following settings. The filter was defined as 128 to represent the number of neurons, with each neuron performing a different convolution to the input of the layer. The kernel size was defined as 1, representing the size of the filter, and the tanh activation function represented the final value of a neuron. Finally, the dense output layer fully connected the 128 neurons to every activation units of the next layer and contained 2 neurons.

3.2 *Assembling related bug reports using threshold-based similarity analysis*

After recognition, severe bug reports, identified by the process described in Section 3.1, are used as the center point to find their related bug reports. This solution helps the software development team can see an overall picture of the software issue. Consequently, this may increase the chances of to completely fixing the software.

All bug reports used in this study are pre-processed in the previous stage and formatted as VSM. Therefore, these bug reports do not require further pre-processing. However, when using BM25 as the main algorithm, each bug report feature should be given weight by *term frequency* (*tf*).

3.2.1 Cosine similarity (CS)

CS is applied to assemblage related bug reports because it has been widely used for bug report studies [26], [27], [33], [44]. The CS equation is [Equation (5)]:

$$sim(V_1,V_2) = \frac{v_1 \cdot v_2}{\|v_1\|\|v_2\|} \tag{5}$$

where V_1 and V_2 are the term vectors of a pairwise between a severe bug report and related bug reports in the dataset. The similarity result should be close to 1 if both bug reports are similar. Also, thresholds are provided to determine the similarity score. Thresholds used in this study are from 0 to 1 with step 0.1. When the similarity score of the severe bug report and a bug report is greater than or equal to the threshold, it means that those bug reports should be grouped into the same cluster because they may be relevant. Yet, when the similarity score of a severe bug report and a bug report is below the threshold, those bug reports may be irrelevant.

3.2.2 BM25

When using BM25, each term should be given its weight by *tf*. The BM25 is applied for assembling related bug reports because this algorithm has been proved that it could return satisfactory for bug reports analysis, especially in real-bug report identification and duplicate bug report analysis [18].

BM25 is a ranking function which was developed in the Okapi information retrieval system [45]. For BM25, it does not require giving term weights by *tf-igm*, where it requires only *document frequency* (*df*). The *df* is the number of documents where the term *t* appears. Instead of regarding the inter-relationship between the query terms with in a document (or bug report), the BM25 equation is [Equation (6)]:

$$BM25(SB, br) = \sum_{i=1}^{|SB|} idf(q_i) \times \left(\frac{tf(q_i, br) \times (k+1)}{tf(q_i, br) + k \times (1 - b + b \times \frac{|br|}{brl_{avg}})} \right) \tag{6}$$

Let *SB* be a severe bug report and br be a bug report that may be related to that *SB*. Therefore, $tf(q_i,br)$ is the term frequency, where it is defined as the number of occurrences that the terms *q*-th of *SB* appear in *br*. Indeed, $|br|$ is the length of br in words, while brl_{avg} is the average bug report length for all the bug reports in the corpus. For, *k* and *b*, they are free parameters. They are used to control the weighting between $tf(q_i,br)$ and the normalized bug report length. Generally, the values of *k* and *b* should be in the range of $1.2 < k < 2.0$ and $0.5 < b < 0.8$ [45], [46]. This study uses the values of *k* and *b* at 2.0 and 0.8 respectively. They are the same values used in [45].

For $idf(q_i)$, it is the *inverse document frequency* of the term *q*-th of *SB*. It can be calculated by Equation (7).

$$idf(q_i) = log\left(\frac{N - df(q_i) + 0.5}{df(q_i) + 0.5} \right) \tag{7}$$

where *N* is the entire number of bug reports in the corpus, while $df(q_i)$ is the number of bug reports containing the term *q*-th of *SB*.

In general, the similarity score should be between 0 and 1. However, when using the BM25 technique to estimate the similarity score, it is possible that this technique can return a score greater than 1.0. Similarity scores should be normalized to allow a comparison of different similarity values using a single scale. Normalizing similarity scores helps to remove the mean and scale to the similarity score variance. To normalize the BM25 similarity scores in the range [0,1], the function was shown as Equation (8) also applies in this case.

$$f(x) = x / (1 + x) \tag{8}$$

where *x* is the similarity score generated by BM25.

In this study, threshold-based text similarity analysis is also applied, where thresholds are also provided to determine the similarity score. The thresholds used in this study are from 0 to 1 with step 0.1. When the

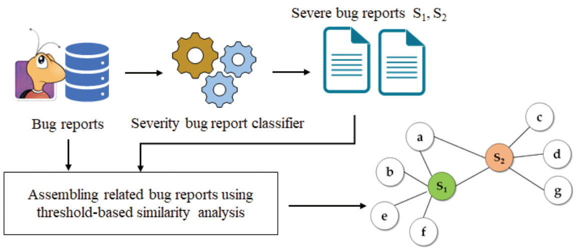

Severe bug reports S_1, S_2
Bug reports
Severity bug report classifier
Assembling related bug reports using threshold-based similarity analysis
a
b
c
d
e
f
g
S_1
S_2

Figure 5: Example of expected results of the proposed method.

similarity score of the severe bug report and its related bug reports is greater than, or equal to the threshold, it appears that those bug reports should be grouped into the same cluster because they may be relevant. Yet, when the similarity score of the severe bug report and its related bug reports is below the threshold, those bug reports may be irrelevant. This process is iteratively performed until that bug report is able to identify its suitable clusters. It is noted that a bug report can be in many clusters.

An example of the expected results of identifying severe bug reports using the severity classification, and assembling related bug reports in suitable clusters with center points as severe bug reports obtained from the previous task is shown in Figure 5.

After performing bug report severity classification, S_1 and S_2 were identified as severe bug reports (Figure 5). Then, using S_1 and S_2 were used to find other related bug reports by CS or BM25. Results determined that bug reports *a*, *b*, *e* and *f* were related to S_1, while bug reports *c*, *d*, and *g* were related to S_2 and bug report *a* was related to both S_1 and S_2.

Consequently, the best model of assembling related bug reports to a specific bug report with blocking severe based on our proposal is selected and compared with the baseline model proposed by [33].

4 The Experimental Results

4.1 *Evaluation measures*

True Positive Rate (TPR) is also called Sensitivity or Recall. It is used to measure the proportion of actual positives that are correctly identified. True Negative Rate (TNR) is also called Specificity. It is used to measure the proportion of actual negatives that are correctly identified. F1 is the harmonic mean of the recall and precision. This measure is used to determine the test accuracy. The best value for F1 is 1 and the worst value is 0. Accuracy is literally how good our model is at predicting the correct category (classes or labels) for the dataset used [47], [48].

The ROC (Receiver Operating Characteristic) curve is used to measure how well a related bug report can be detected from a dataset of bug reports. The ROC curve is plotted with TPR against the false positive rate (FPR or 1-TNR), with TPR on the y-axis and FPR on the x-axis. While AUC (Area Under the Curve) is used to presents the degree or measure of separability by considering the area under the ROC curve [47], [48]. The ROC curve and AUC are two of the most important evaluation metrix for checking the performance of dependent bug reports assembly. The ROC curve and AUC can be used to obtain the most appropriate threshold and models based on our proposed method.

In addition, this study used feedback, precision and likelihood measurements [33], [44], [49], which before presenting the formulas for these three measurements the following sets should be firstly defined. Let BR_q be the set of related bug reports retrieved by the proposed method, while $BR_q(k)$ is top-k bug reports in BR_q ordered by textual similarity (only defined for $|BR_q| \geq k$). R_q is the set of related bug reports with their answers. Meanwhile, Z is the total number of severe bug reports (queries) in total, and Z_k is a subset of Z that can retrieve the related bug reports at least k. These definitions help to define feedback, precision, and likelihood.

Feedback: It involves measuring the number of bug reports that are retrieved when using a given severe bug report as a query. Formally, the feedback of k, denoted as $FB(k)$, is the percentage of queries with at least k bug reports retrieved. The feedback equation can be defined as [Equation (9)]:

$$FB(k) = \frac{|Z_k|}{|Z|} \tag{9}$$

Precision: It is denoted as P(k) and used to measure the ratio of related bug reports that are retrieved. The formula for precision can be expressed as [Equation (10)]:

$$P_q(k) = \frac{|BR_q(k) \cap R_q|}{|BR_q(k)|} \tag{10}$$

Moreover, total precision in our dataset is determined as the average precision executed for each severe bug report (or query). The equation can be defined as [Equation (11)]:

$$P(k) = \frac{1}{|Z_k|} \sum_{q \in Z_k} P_q(k) \tag{11}$$

Likelihood: It is denoted as $L(k)$. The likelihood is a common measure used to estimate the usefulness of retrieving related bug reports. The likelihood of the top-k related bug reports can be defined as [Equation (12)]:

$$L_q(k) = \begin{cases} 1 & \text{if } BR_q(k) \cap R_q \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

where $L_q(k)$ is a binary measure. If at least one related bug report exists among the top-k bug reports that are retrieved, the answer is returned one; if not, the return is zero. The total likelihood in our dataset is defined as the average likelihood measured for each severe bug report (or query). The equation can be defined as [Equation (13)]:

$$L(k) = \frac{1}{|Z_k|} \sum_{q \in Z_k} L_q(k) \tag{13}$$

4.2 *The experimental results of identifying severe bug reports using classification technique*

After obtaining the bug reports represented their features with weights in the VSM format, the training set will be used to model "*bug report severity classifier*". This classifier is used to identify bug reports with blocking severe. This study compared two different techniques for modeling bug report severity classifier. They are SVM with RBF kernel function and a deep learning technique that is called CNN.

Results in Table 3 show that RF, MNB, and SVM with RBF returned better results than CNN when using our datasets. However, SVM with RBF returned the most satisfactory results. CNN returned the poorest results because this algorithm generally requires a lot of training data. Unfortunately, all datasets used in this study were small, and this was the main reason why CNN gave poor results. By contrast, SVM is the best suited for extreme case binary classification and outliers have less impact. Simply speaking, SVM performs well for smaller datasets. The RF and MNB classifier models returned poorer results than SVM. The RF classifier may overfit some datasets with outlier classification, and also consists of many decision trees, whose construction may impact irrelevant features. While the MNB classifier is a simple and easy algorithm for the text classification. Theoretically, naive Bayes classifiers have a minimum error rate compared with other classifiers. However, practically this is not always true because of the assumption of class conditional independence.

In addition, using *tf-igm* as a term weighting reinforces the class distinguishing power of each term. Therefore, this may help to increase the performance for severity classification, especially when using *tf-igm* along with machine learning algorithms.

Finally, the severity classifiers modelled by SVM with RBF are selected as the best models and they are compared with the baseline method proposed by [15].

4.3 *The experimental results of assembling related bug reports using threshold-based similarity analysis*

Table 4 shows that the BM25 algorithm outperformed CS on the datasets used in this study. One potential reason for the effectiveness of BM25 is that it can show the degree of importance of terms appearing in documents. This allows BM25 to derive the relevance of a document more accurately by extracting elaborate information of terms, documents, and document collection, rather than considering only term appearance following the CS similarity scheme. BM25 is better for document length normalization and satisfying the

Table 3: The experimental results of identifying severe bug reports using classification technique

Dataset	RF			MNB			SVM with RBF			CNN		
	ACC	F1	AUC	ACC	F1	AUC	ACC	F1	AUC	ACC	F1	AUC
Core:Layout	0.89	0.89	0.95	0.92	0.92	0.98	**0.94**	**0.94**	**0.99**	0.73	0.73	0.73
Firefox	0.77	0.77	0.83	0.78	0.78	0.86	**0.83**	**0.83**	**0.89**	0.72	0.72	0.72
SeaMonkey	0.77	0.77	0.84	0.77	0.77	0.85	**0.81**	**0.81**	**0.87**	0.72	0.72	0.72

concavity constraint of the term frequency. Also, BM25 performs well with short document collections [50], and each bug report used in this study was short. Based on these reasons, BM25 achieves better performance compared to CS.

However, it was not possible to specify the best threshold for BM25 since the thresholds which had the best performance for these techniques may be between 0.1 and 0.5. To specify the best threshold for BM25, the ROC curve and AUC were applied. Figure 6 indicates that the best BM25 threshold should be 0.5. Then, it returns the best AUC score of assembling related bug reports at 0.835. Finally, this model is selected as the best model for assembling related bug reports.

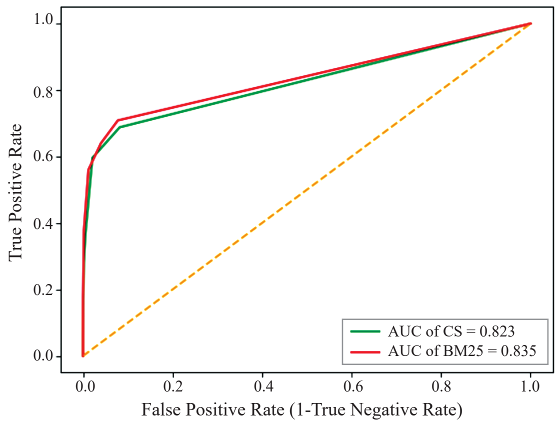


Figure 6: The AUC scores of CS and BM25 for assembling related bug reports.

4.4 *The results of comparing to the baseline methods*

4.4.1 Comparing the proposed method of identifying severe bug reports to the baseline method

The baseline method used to compare with our method is proposed by [15]. They proposed a deep neural network-based automatic approach to predict the severity of bug reports. Their method consisted of four steps. First, NLP techniques were applied for text pre-processing of bug reports. Second, an emotion score was computed and assigned for each bug report. Third, a vector for each pre-processed bug report was created and, finally, the constructed vector and emotion score of each bug report was passed to a deep neural network-based classifier for severity prediction. Three layers of CNN were used with the following settings: filter = 128, kernel size = 1, and activation = tanh. The filter represents the number of neurons, and each neuron performs a different convolution on the input to the layer (more precisely, the neurons' input weights form convolution kernels). Kernel size represents the size of the filter, while the activation function represents the final value of a neuron. The output of CNN was forwarded to a flattening layer that turned the given converted numerical vectors into a one-dimensional vector. (Table 5)

Results in Table 6 show that our proposed method improved both the accuracy and F1 over the baseline method. There are three reasons for this. First, we considered the used features. Ramay *et al.* used only unigram as features, while we used unigram together

Table 4: The experimental results of assembling related bug reports using threshold-based similarity analysis

Algorithm	Evaluation Metrics	Threshold Used									
		0.1	***0.2***	***0.3***	***0.4***	***0.5***	***0.6***	***0.7***	***0.8***	***0.9***	***1.0***
CS	TPR	**0.688**	0.596	0.36	0.28	0.179	0.058	0.029	0.009	0.001	0.001
	TNR	**0.918**	0.979	0.995	0.998	0.999	1.000	1.000	1.000	1.000	1.000
BM25	TPR	0.710	0.710	0.710	0.710	**0.709**	0.640	0.560	0.379	0.180	0.000
	TNR	0.918	0.918	0.918	0.918	**0.922**	0.960	0.988	0.998	1.000	1.000

Table 5: The results of comparing to the baseline method proposed by Ramay *et al.*

Dataset	Ramay *et al.* (2019)					Proposed Method				
	ACC	R	P	F1	AUC	ACC	R	P	F1	AUC
Core:Layout	0.79	0.79	0.79	0.78	0.79	0.94	0.94	0.94	0.94	0.99
Firefox	0.76	0.76	0.77	0.76	0.76	0.83	0.83	0.83	0.83	0.89
SeaMonkey	0.76	0.76	0.76	0.76	0.76	0.81	0.81	0.81	0.81	0.87

Table 6: The results of comparing to the baseline method proposed by Rocha *et al.*

Metrics	Core:Layout		Firefox		SeaMonkey	
	Rocha *et al.*	**Proposed Method**	**Rocha *et al.***	**Proposed Method**	**Rocha *et al.***	**Proposed Method**
Feedback	0.999	0.999	0.999	0.999	0.999	0.999
Precision	0.393	0.458	0.390	0.450	0.390	0.450
Likelihood	0.607	0.678	0.598	0.645	0.598	0.645

with CamelCase. Using unigram alone was not sufficient because unigram was unable to indicate the specificity of the software. Thus, using CamelCase together with unigram as features improved this problem.

Second, we considered the predefined class weight of each bug report before the classification task. Ramay *et al.* utilized Senti4SD to assign class weight to each bug report and used this to determine other attributes in the deep neural network for severity prediction of bug reports. Although this application is interesting, errors can occur during performance. Senti4SD was not developed using bug report datasets. Hence, many words in Senti4SD might not relate to bug reports and using Senti4SD might be appropriate for some datasets but lead to poor results in others.

By contrast, our proposed method did not use Senti4SD to predefine class weight. We used *tf-igm*, as an STW scheme to increase the class distinguishing power of each term found in our dataset. This helped to competently identify and distinguish between the characteristics of each class. For these reasons, our method generated better results than Ramay *et al.*

Word2Vec was also used by Ramay *et al.* to generate word vectors, however, this technique might be ineffective as it was not able to separate some opposite word pairs. For example, "*good*" and "*bad*" are sometimes located very close to each other in the vector space and this may limit the performance of word vectors when undertaking NLP tasks, such as severe bug report analyses.

In addition, the main mechanism used by Ramay *et al.* for classification was deep learning. In general, this technique requires a large dataset. Unfortunately, all our datasets were small. Therefore, this technique proved to be poor for all datasets used in this study. Moreover, deep learning techniques require extra computational resources than machine learning algorithms. Therefore, our method performs faster than the method proposed by Ramay *et al.*

4.4.2 Comparing the result of assembling related bug reports to the baseline method

Our proposed method was compared with the baseline method proposed by [33]. The tool developed by using the baseline method is called "NextBug", which is implemented as a plug-in for Bugzilla. Then, Rocha *et al.* used only the summary component of bug reports and they also used unigram features with tf-idf. The main mechanism for identifying similar bug reports was cosine similarity with a threshold set as 0.1. Interestingly, Rocha *et al.* retrieved only the first five recommended bug reports and they used feedback, precision, and likelihood for their evaluation. Table 6 shows a comparison of the results.

After testing with three tested sets of bug reports – namely Core:Layout, Firefox, and SeaMonkey, the average score of feedback, precision, and likelihood are presented in Table 6. Results in Table 6 show that our proposed method returned better results than the baseline method proposed by [33], with improved average scores of precision and likelihood at 15.76% and 9.14%, respectively. There are two points that can help to improve the performance of assembling dependent bug reports. First, the use of CamelCase as features can indicate the specificity of a problem domain in software, since different problem domains of a software may use different CamelCase terms. Meanwhile, BM25 is the appropriate similarity technique for this work. A potential reason for the effectiveness of BM25 is that it can show the degree of importance of terms appearing in documents, and thus to derive the relevance of a document to a given more accurately by taking more elaborate information of terms, documents, and document collection into consideration, rather than only term appearance in the traditional similarity scheme (e.g. cosine similarity). For example, the weighting model of BM25 incorporates document length, the average length of all documents in the collection, as well as the term frequency normalization effect.

This technique is subsequently able to return better performance than the CS technique.

5 Conclusions

A software bug (or defect) can cause a program to crash, or produce invalid, or unexpected results. In general, end users can help development teams find bugs in software. Feedback or reports related to bugs from end-users are called "*bug reports*". Bug reports are essential for developer teams to improve and maintain software quality. However, collecting bug reports from users around the world is difficult. A better collection method for large bug reports involving an increased numbers of users requires bug tracking systems (BTS). These systems allow end-users to report, describe, track, classify, and comment on bug reports, including feature requests. At present, systems like Bugzilla, Jira, Mantis, or Trac are widely used for bug reporting. In the early BTS usage, when a new bug report was submitted to a bug report repository, a special person called a bug triager screened and prioritized the report before assigning the suitable developers to address a particular bug. All processes in BTS are time-consuming because they are performed manually. Therefore, copious research has investigated methods to automatically identify the necessary information from bug reports to allow software bug fixing. Unfortunately, most studies of bug reports focused only on one issue, whereas more complete and comprehensive software bug fixing requires assessing multiple issues concurrently. This becomes a challenge in our study, which presents a method of identifying bug reports at a severe level from a bug report repository, together with finding their related bug reports to visualize the overall picture of a software problem domain. This method is called "*mining bug report repositories*" and consists of two main processing steps. The first step is the classification process, called "*bug severity classification*". Classification technique involves experimenting with various supervised machine learning algorithms to model bug severity classifiers. Classifier models are used to automatically identify severe bug reports. The second step applies unsupervised learning to automatically assemble bug reports related to server bug reports with respect to a similarity measure. This process is called "*threshold-based similarity analysis*". Then, these two processing steps are investigated using various algorithms to obtain the most suitable models of bug severity classification and threshold-based similarity analysis. Our study experimented on three open data sources as SeaMonkey, Firefox, and Core:Layout downloaded from Bugzilla. The most suitable models were compared with baseline methods. Result in this study showed that our method improved the performance of bug severity classification and assembly of related bug reports over the baseline methods and increased the chances of fixing bugs in the software.

Acknowledgments

This research was supported by Division of Research Facilitation and Dissemination, Mahasarakham University (Grant No. 6303001/2563).

References

[1] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 499–510.

[2] R. J. Sandusky, L. Gasser, and G. Ripoche, "Bug report networks: Varieties, strategies, and impacts in af/oss development community," in *Proceedings of 1st Int'l Workshop on Mining Software Repositories*, 2004, pp. 80–84.

[3] 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.

[4] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008, pp. 308–318.

[5] S. Davies and M. Roper, "What's in a bug report?," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2014, p. 26.

[6] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: A text-based approach to classify change requests," in *CASCON, 2008*, vol. 8,

pp. 304–318.
[7] K. Herzig, S. Just, and A. Zeller, “It's not a bug, it's a feature: How misclassification impacts bug prediction,” in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 392–401.
[8] N. Limsettho, H. Hata, A. Monden, and K. Matsumoto, “Automatic unsupervised bug report categorization,” in *2014 6th International Workshop on Empirical Software Engineering in Practice*, 2014, pp. 7–12.
[9] H. Qin and X. Sun, “Classifying bug reports into bugs and non-bugs using LSTM,” in *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, 2018, p. 20.
[10] P. Terdchanakul, H. Hata, P. Phannachitta, and K. Matsumoto, “Bug or not? bug report classification using n-gram idf,” in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 534–538.
[11] K. Kowsari, K. Jafari Meimandi, M. Heidarysafa, S. Mendu, L. Barnes, and D. Brown, “Text classification algorithms: A survey,” *Information*, vol. 10, no. 4, p. 150, 2019.
[12] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, “Predicting the severity of a reported bug,” in *7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 2010, pp. 1–10.
[13] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, “Comparing mining algorithms for predicting the severity of a reported bug,” in *15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 249–258.
[14] T. Menzies and A. Marcus, “Automated severity assessment of software defect reports,” in *2008 IEEE International Conference on Software Maintenance*, 2008, pp. 346–355.
[15] 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. 46846–46857, 2019.
[16] O. Chaparro, J. M. Florez, U. Singh, and A. Marcus, “Reformulating queries for duplicate bug report detection,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 218–229.
[17] N. Jalbert and W. Weimer, “Automated duplicate detection for bug tracking systems,” in *2008 IEEE International Conference on Dependable Systems and Networks with FTCS and DCC (DSN)*, 2008, pp. 52–61.
[18] C.-Y. Lee, D.-D. Hu, Z.-Y. Feng, and C.-Z. Yang, “Mining temporal information to improve duplication detection on bug reports,” in *2015 IIAI 4th International Congress on Advanced Applied Informatics*, 2015, pp. 551–555.
[19] Q. Xie, Z. Wen, J. Zhu, C. Gao, and Z. Zheng, “Detecting duplicate bug reports with convolutional neural networks,” in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, 2018, pp. 416–425.
[20] J. Kanwal and O. Maqbool, “Bug prioritization to facilitate bug report triage,” *Journal of Computer Science and Technology*, vol. 27, no. 2, pp. 397–412, 2012.
[21] J. Uddin, R. Ghazali, M. M. Deris, R. Naseem, and H. Shah, “A survey on bug prioritization,” *Artificial Intelligence Review*, vol. 47, no. 2, pp. 145–180, 2017.
[22] Q. Umer, H. Liu, and Y. Sultan, “Emotion based automated priority prediction for bug reports,” *IEEE Access*, vol. 6, pp. 35743–35752, 2018.
[23] Q. Umer, H. Liu, and I. Illahi, “CNN-based automatic prioritization of bug reports,” *IEEE Transactions on Reliability*, 2019.
[24] P. Bhattacharya and I. Neamtiu, “Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging,” in *2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.
[25] J. Lee, D. Kim, and W. Jung, “Cost-aware clustering of bug reports by using a genetic algorithm,” *Journal of Information Science and Engineering*, vol. 35, no. 1, pp. 175–200, 2019.
[26] R. Almhana, W. Mkaouer, M. Kessentini, and A. Ouni, “Recommending relevant classes for bug reports using multi-objective search,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 286–295.
[27] X. Ye, R. Bunescu, and C. Liu, “Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation,” *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 379–402, 2015.

[28] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 14–24.

[29] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," *ACM Sigsoft Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[30] Y. Sun, Q. Wang, and Y. Yang, "Frlink: Improving the recovery of missing issue-commit links by revisiting file relevance," *Information and Software Technology*, vol. 84, pp. 33–47, 2017.

[31] S. Akbarinasaji, B. Caglayan, and A. Bener, "Predicting bug-fixing time: A replication study using an open source software project," *Journal of Systems and Software*, vol. 136, pp. 173–186, 2018.

[32] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models: Can we do better?," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 207–210.

[33] H. Rocha, G. De Oliveira, H. Marques-Neto, and M. T. Valente, "NextBug: A Bugzilla extension for recommending similar bugs," *Journal of Software Engineering Research and Development*, vol. 3, no. 1, p. 3, 2015.

[34] N. Pandey, A. Hudait, D. K. Sanyal, and A. Sen, "Automated classification of issue reports from a software issue tracker," in *Progress in Intelligent Computing Techniques: Theory, Practice, and Applications*, 2018, pp. 423–430.

[35] Y. Zhou, Y. Tong, R. Gu, and H. Gall, "Combining text mining and data mining for bug report classification," *Journal of Software: Evolution and Process*, vol. 28, no. 3, pp. 150–176, 2016.

[36] B. Luaphol, B. Srikudkao, T. Kachai, N. Srikanjanapert, J. Polpinij, and P. Bheganan, "Feature comparison for automatic bug report classification," in *International Conference on Computing and Information Technology*, 2019, pp. 69–78.

[37] K. Chen, Z. Zhang, J. Long, and H. Zhang, "Turning from TF-IDF to TF-IGM for term weighting in text classification," *Expert Systems with Applications*, vol. 66, pp. 245–260, 2016.

[38] B. Luaphol, J. Polpinij, and M. Kaneampornpun, "Automatic bug report severity prediction by binary text classification techniques," in *The 25th International Symposium on Artificial Life and Robotics 2020*, 2020, pp. 206–211.

[39] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. Massachusetts: MIT press, 2012.

[40] K. Soman, R. Loganathan, and V. Ajay, *Machine Learning with SVM and Other Kernel Methods*. Delhi, India: PHI Learning Pvt. Ltd., 2009.

[41] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge, UK: Cambridge University Press, 2000.

[42] 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.

[43] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[44] H. Rocha, G. Oliveira, H. Maques-Neto, and M. Valente, "Nextbug: A tool for recommending similar bugs in open-source systems," in *V Brazilian Conference on Software: Theory and Practice–Tools Track (CBSoft Tools)*, 2014, vol. 2, pp. 53–60.

[45] C.-Z. Yang, H.-H. Du, S.-S. Wu, and X. Chen, "Duplication detection for software bug reports based on bm25 term weighting," in *2012 Conference on Technologies and Applications of Artificial Intelligence*, 2012, pp. 33–38.

[46] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. New York: ACM Press, 1999.

[47] G. Forman, "An extensive empirical study of feature selection metrics for text classification," *Journal of Machine Learning Research*, vol. 3, no. 3, pp. 1289–1305, 2003.

[48] N. Japkowicz and M. Shah, *Evaluating Learning Algorithms: A Classification Perspective*. Cambridge, UK: Cambridge University Press, 2011.

[49] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[50] S. Robertson and H. Zaragoza, *The Probabilistic Relevance Framework: BM25 and Beyond*. Massachusetts: Now Publishers Inc, 2009.