Text Mining Approaches for Dependent Bug Report Assembly and Severity Prediction

Bancha Luaphol
Department of Digital Technology,
Kalasin University, Thailand
bancha.lu@ksu.ac.th

Jantima Polpinij
Department of Computer Science,
Mahasarakham University, Thailand
jantima.p@msu.ac.th

Manasawee Kaenampornpan
Department of Computer Science,
Mahasarakham University, Thailand
manasawee.k@msu.ac.th

Abstract: *In general, most existing bug report studies focus only on solving a single specific issue. Considering of multiple issues at one is required for a more complete and comprehensive process of bug fixing. We took up this challenge and proposed a method to analyze two issues of bug reports based on text mining techniques. Firstly, dependent bug reports are assembled into an individual cluster and then the bug reports in each cluster are analyzed for their severity. The method of dependent bug report assembly is experimented with threshold-based similarity analysis. Cosine similarity and BM25 are compared with term frequency (tf) weighting to obtain the most appropriate method. Meanwhile, four classification algorithms namely Random Forest (RF), Support Vector Machines (SVM) with the RBF kernel function, Multinomial Naïve Bayes (MNB), and k-Nearest Neighbor (k-NN) are utilized to model the bug severity predictor with four term weighting schemes, i.e., tf, term frequency-inverse document frequency (tf-idf), term frequency-inverse class frequency (tf-icf), and term frequency-inverse gravity moment (tf-igm). After the experimentation process, BM25 was found to be the most appropriate for dependent bug report assemblage, while for severity prediction using tf-icf weighting on the RF method yielded the best performance value.*

Keywords: *Bug report, dependent bug report assembly, bug severity prediction, threshold-based similarity analysis, cosine similarity, BM25, term weighting, classification algorithm*.

Received April 28, 2020; accepted February 13, 2022
https://doi.org/10.34028/iajit/19/6/9

1. Introduction

Detecting and identifying bugs or defects in large software systems such as open source is never straightforward and easy. A solution that can help to fix software bugs is to gather bug report data from end-users worldwide because bug reports can detail the occurrence of defects or problems with specific formats. For more convenient gathering of bug reports, various Bug Tracking Systems (BTS) such as Jira, ClickUp, Mantis, Bugzilla, and eTraxis have been proposed [1, 8, 45]. The number of bug reports has been steadily increasing. It also enormously increases the size of bug report repositories, necessitating automatic identification of significant necessitating automatic identification of significant information hidden in bug reports to be utilized for software bug fixing.

In general, there are three primary studies for bug report [45]. Firstly, bug report optimization enhances report quality and reduces inaccurate information. Three important tasks are optimization of bug report content [5, 6], misclassification [14, 28, 32] and severity prediction [31, 40]. Secondly, bug report triage aims to screen and prioritize bug reports to ensure that all issues can be appropriately managed. This involves [13, 16, 24, 40], prioritization [18], and suitable developer assignment tasks [7, 25]. Finally, bug fixing concerns finding a solution to quickly fix the software bugs. For bug fixing study, there are three major tasks as localization of bug [2], recovery of links between bug reports because of files changing [39], and predicting time of bug fixing [8, 30].

Although many tasks related to bug reports have been studied and proposed, misclassification and severity prediction issues have attracted the most interest because the performance of software bug fixing relies on the quality and accuracy of the bug reports. Misclassification between bug and non-bug reports provides low quality or incorrect information to developer teams that cannot be utilized for software bug fixing, while severity prediction aims to predict the severity level of a bug report to determine which bug should be fixed first.

However, other bug report studies have addressed how the "bug dependence issue" impacts software bug fixing [29]. If bug "*x*" and bug "*y*" are related, bug "*y*" will continue to exist despite being addressed because bug "*x*" has not yet been entirely fixed. This issue has been mentioned in many studies but it has not yet been seriously addressed. This is because misclassification and severity prediction continue to be studied by many researchers, where performance improvements of bug report misclassification and severity prediction still remain unresolved.

In general, most existing bug report studies focus only on a specific single issue. However, to obtain a more complete bug fixing process, it might be better if considering multiple simultaneously. For example, if dependent bug reports can be recognized and analyzed together with bug severity level, this would import significant information for the appropriate assignation of software issues to developers and may help to fix bugs completely. This challenge was addressed here by presenting a method to study the two separate issues of bug report dependency and bug severity level concurrently. Bug report dependency analysis assists in acknowledging the relationships between bugs [8], while bug severity level analysis identifies and recognizes bugs with higher severity [22, 23]. Bugs with the highest severity level, called severe level, should be fixed first.

This paper is organized as follows. In section 2, it reviews the related works and the datasets is detailed in section 3, while the research method is described in section 4. The experimental results are shown in section 5. Finally, section 6 is to conclude this study.

2. Related Work

Software containing defects or bugs can produce erroneous or unexpected results [11, 41]. Identifying bugs in small software systems is not difficult. By contrast, bugs in larger open source software systems may be hidden and require more complex remediation methods [20]. If the software development team cannot identify all the defects or bugs in an open source, global end-users can assist by providing important bug-related information, called bug reports.

Many systems for tracking bug reports (e.g., Mantis, Jira, Bugzilla, Trace, etc.,) have been developed and proposed to facilitate gathering bug reports [1, 8, 45]. Consequently, many bug reports are generated and submitted daily to those systems by software end-users [37]. With a continual increase in the size of bug report repositories, manual analysis becomes a time-consuming and expensive process, with the essential for automatic data management and analysis.

There are three significant areas for bug report studies. Firstly, bug report optimization enhances report quality and reduces inaccurate information. This study can be divided into three major tasks as misclassification [14, 28, 32], content optimization [5, 6] and severity prediction [31, 40]. Misclassification is the most important task in this area. This is because if outlier or inappropriate bug reports are not removed, the next stage of bug report analysis can be a time-consuming process.

The second study area is called bug report triage. It consists of three main tasks namely duplicated bug detection [13, 16, 24, 41], prioritization [18] and assigning bug reports to suitable software developers [25]. Duplicated bug detection automatically identifies and removes duplications from the bug report to reduce the analysis time of further processing. Bug report prioritization identifies bug reports that should be fixed first, while the last study area of bug report triage assigns the bug reports to the most appropriate software developers.

For bug fixing area, it consists of three main tasks as localization of bug [2, 46], recovering links between bug reports when files are changed [39] and predicting time of software bug fixing [8, 30]. Localization of bug is to automatically find the position of the bugs in a software code, while recovering links between bug reports when files are changed aims to utilize logs of software correction history for analyzing a change in files. Finally, predicting time of software bug fixing is to estimate time of resolving the issue.

Besides these aforementioned tasks, another bug report topic has an impact on the bug-fixing process. This is called the “bug dependency issue” [29]. Many researchers have been mentioned this topic but not studied in detail. The bug dependency issue should be considered simultaneously with the other mentioned issues for accomplishing the process of bug fixing.

Previous bug reports studies only focused on only single issue. Considering multiple bug report issues at the same time may help to facilitate a method of bug fixing to be more complete and comprehensive. For example, recognizing and analyzing dependent bug reports together with bug severity level would render significant information for the appropriate assignation of software issues to developers and help to completely eradicate the bugs. Therefore, this issue is taken up as a challenge in this study.

3. Datasets

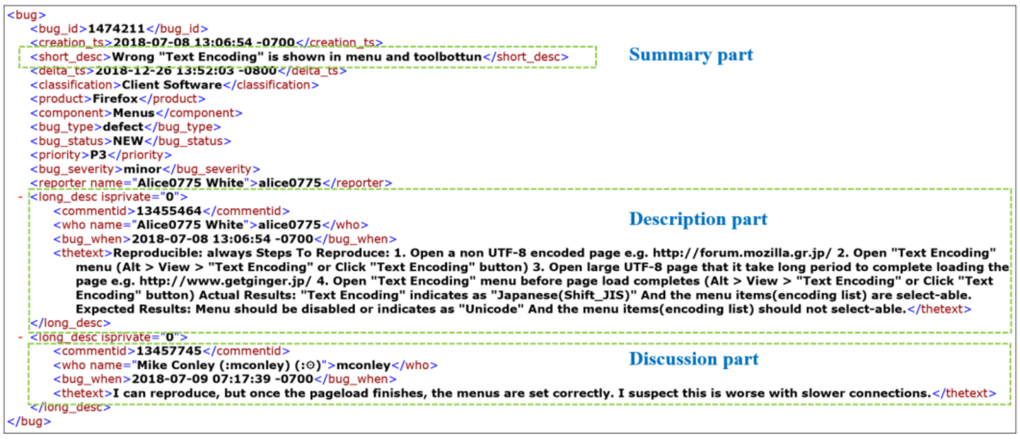
A bug report generally comprises of three main components. They are the summary or title, the description as the bug report detail and the discussion as comments on that bug report. The summary part of bug reports is widely used on the most previous works studies because this part contains less outlier or noise [46]. A bug report example is shown as Figure 1.

This study used a dataset relating to Mozilla Firefox were gathered from Bugzilla and they were downloaded between 1 October and 31 December 2019. The bug reports with ‘verified’ and ‘closed’ statuses are used for this study. This is because these bug reports have been confirmed by bug triagers and development teams that they are ‘real-bug’ reports.

In general, there are six levels of bug severity. They are ‘blocker’, ‘critical’ and ‘major’, ‘normal’, ‘minor’, and ‘trivial’. In this study, real-bug reports labeled as the first three severity levels were assigned to the severe class, while real-bug reports labeled as the remaining three severity levels were assigned to the non-severe class.

In this dataset, 512 bug reports were considered as meta-bugs by bug triagers, and 21,488 bug reports were considered as bug dependency in those meta-bug reports. Mata-bug report is the first or main bug report of a specific software problem domain. However, it ispossible that a bug report can be both a dependent bug report and a meta-bug report at the same time.

To reduce the impact of imbalanced data resulting from the severely skewed class distribution, we used 5,000 bug reports per class for the first stage, called modeling for software bug report severity predictor. It is performed with a practice based on 10-fold cross validation.



```
<bug>
    <bug_id>1474211</bug_id>
    <creation_ts>2018-07-08 13:06:54 -0700</creation_ts>
    <short_desc>Wrong "Text Encoding" is shown in menu and toolbottun</short_desc>
    <delta_ts>2018-12-26 13:52:03 -0800</delta_ts>
    <classification>Client Software</classification>
    <product>Firefox</product>
    <component>Menus</component>
    <bug_type>defect</bug_type>
    <bug_status>NEW</bug_status>
    <priority>P3</priority>
    <bug_severity>minor</bug_severity>
    <reporter name="Alice0775 White">alice0775</reporter>
  - <long_desc isprivate="0">
        <commentid>13455464</commentid>
        <who name="Alice0775 White">alice0775</who>
        <bug_when>2018-07-08 13:06:54 -0700</bug_when>
        <thetext>Reproducible: always Steps To Reproduce: 1. Open a non UTF-8 encoded page e.g. http://forum.mozilla.gr.jp/ 2. Open "Text Encoding"
            menu (Alt > View > "Text Encoding" or Click "Text Encoding" button) 3. Open large UTF-8 page that it take long period to complete loading the
            page e.g. http://www.getginger.jp/ 4. Open "Text Encoding" menu before page load completes (Alt > View > "Text Encoding" or Click "Text
            Encoding" button) Actual Results: "Text Encoding" indicates as "Japanese(Shift_JIS)" And the menu items(encoding list) are select-able.
            Expected Results: Menu should be disabled or indicates as "Unicode" And the menu items(encoding list) should not select-able.</thetext>
    </long_desc>
  - <long_desc isprivate="0">
        <commentid>13457745</commentid>
        <who name="Mike Conley (:mconley) (:☺)">mconley</who>
        <bug_when>2018-07-09 07:17:39 -0700</bug_when>
        <thetext>I can reproduce, but once the pageload finishes, the menus are set correctly. I suspect this is worse with slower connections.</thetext>
    </long_desc>
</bug>
```

Figure 1. A bug report example.

The remaining bug reports were used in an experiment for assembling of dependent bug reports together and identifying of the severity of each case. Two hundred meta-bug reports were also utilized as the center point (or centroid) for assembling the dependent bug reports.

4. The Proposed Method

The methodology for the study is described in this section. The first stage is a preliminary for bug severity predictor modeling described in section 4.1, while the second stage is the proposed method of assembling dependent bug reports followed by identifying bug severity level described in sections 4.2.

4.1. Preliminary: Modeling of Bug Severity Predictor

This section explains the process of generating the severity predictor used to predict severity levels for bug reports after they have been automatically assigned into appropriate clusters.

4.1.1. Pre-Processing of Bug Reports

This stage commences with the step of preparing bug report data by splitting the text into tokens [42], which are referred to as "words" in this study. In this study, the two features of bug reports namely Unigram and CamelCase are utilized here. Unigram is a single word, while CamelCase [2, 28, 46] is the practice of writing many words together without punctuation or intervening spaces. Some examples of CamelCase are 'ToolBar' and 'userInterface'.

CamelCase and unigram are widely employed in many previous bug report studies. This is because CamelCase is capable of revealing software specificity, while a unigram (or single word) is easier to leverage from a bug report. However, before using the CamelCase words, they are generally separated into single words. As this, it may help to expand the bug report features. Therefore, 'ToolBar' and 'userInterface' can be broken as 'Tool', 'Bar' and 'user', 'Interface', respectively.

After bug reports are tokenized into words, the 178 stop words listed by the Natural Language Toolkit (NLTK) library are removed. For the stemming process, the Snowball stemmer is performed to reduce the inflectional forms from each word to a common base or root form, called 'stem' [43].

4.1.2. Representation of Bug Reports

After pre-processing of bug reports is done, those are formatted as the Vector Space Model (VSM) [36]. This format represents each bug report as an *N*-dimensional vector, where *N* is the number of distinct terms (or words) found in the bug reports. The *i*-th term of a vector contains the weight score, called the term weight. In this study, four term weighting schemes (i.e. *tf*, *tf-idf*, *tf-igm* and *tf-icf*) are compared as described below.

Term Frequency (*tf*) is the simplest term weighting method. It is often used in bug report studies and returns satisfactory results [27, 31]. The formula of $f_{t,d}$ with logarithmically scaled frequency can be written as Equation (1).

$$tf_{t,d} = log(1 + f_{t,d}) \qquad (1)$$

Where $f_{t,d}$ is frequency of term (or word) t appearing in the document d.

Term frequency-inverse document frequency (*tf-idf*) [4] can be defined as Equation (2).

$$tf-idf_t = tf_{t,d} \times \log\left(\frac{N}{df_t}\right) \quad (2)$$

Where $tf_{t,d}$ is the frequency of term t appearing in a document d. The idf_t, is calculated by the logarithmically scaled number of the total number of documents in a collection (N) that is divided by the total number of documents containing the term t (df_t).

Term frequency-inverse gravity moment (*tf-igm*) is a Supervised Term Weighting (STW) scheme [10]. It can provide a term's class distinguishing power using the *igm* measure. The *tf-igm* formula can be written as Equation (3).

$$tf-igm_{t,d} = f_{t.d} \times (1+\lambda \times igm(t_k)) \quad (3)$$

Where $f_{t,d}$ is defined as the frequency of term t that appears in a document d. Meanwhile, λ is an adjustable coefficient parameter which is utilized to maintain the relative balance between two weight scores (global and local weights). The value of λ should be between 5.0 to 9.0. The *igm* is employed to measure a term's inter-class distribution concentration, and it can be defined as Equation (4).

$$igm(t_k) = \frac{f_{k1}}{\sum_{r=1}^{m} f_{kr} \cdot r} \quad (4)$$

Where f_{kr} is the frequency of term t_k that occurs in different classes, and r=1, 2, ..., m. The classes are listed in descending order, with the rank denoted by r. Meanwhile, f_{kr} is the frequency of class-specific document (df_{kr}), where df_{kr} is the number of documents in the r-th class containing the term t_k.

Lastly, the term frequency-inverse class frequency (*tf-icf*) is also a STW scheme proposed by Lertnattee and Theeramunkong [26]. *tf* is the number of term t that occurs in document d, while *icf* is the global term weight based on counting the total number of classes in the collection (N) that is divided by the total number of classes containing the term t (n_t). The formula of *tf-icf* can be written as Equation (5).

$$tf-icf_{t,d} = log(1+f_{t,d}) \times log_2\left(\frac{N}{n_t}\right) \quad (5)$$

4.1.3. Modeling of Bug Severity Predictor

To model the bug report severity predictors, suppose D is a training set of bug reports and d is a bug report, denoted as D=d_1, d_2, ..., d_i. A set of classes (C) can be denoted as C={*non-severe*, *severe*}. Four machine learning algorithms were also compared in this study, with each briefly described as follows.

K-Nearest Neighbor (*K*-NN) [17] is a nonparametric-based technique that has been employed in numerous academic disciplines for text classification applications. This algorithm determines the k nearest neighbors among training documents and weights the class candidates based on the classes of the closest k neighbors. The similarity scores are utilized to assign the candidate to the class with the highest score from the unknown label class of document x. The k-NN rule can be defined as Equation (6).

$$f(x) = \arg\max S(x, C_j) = \sum_{d_i \in KNN} sim(x, d_i) y(d_i, C_j) \quad (6)$$

Where S denotes the score with respect to $S(x, C_j)$ as the score of candidate i to the class of j, and the $f(x)$ output is a label for the document being analyzed.

Multinomial Naïve Bayes (MNB) [21] is a very well-known probabilistic algorithm for text or document classification tasks. The MNB algorithm can be defined as Equation (7).

$$P(c \mid d) = \frac{P(c)\prod_{w \in d} P(d \mid c)^{n_{wd}}}{P(d)} \quad (7)$$

Where n_{wd} is the number of word w appearing in the document and $P(w|c)$ is the probability of word w appearing in a given class c. $P(w|c)$ can be calculated as Equation (8).

$$P(w \mid c) = \frac{1+\sum_{d \in D_c} n_{wd}}{k + \sum_{w'} \sum_{d \in D_c} n_{w'd}} \quad (8)$$

Support Vector Machines (SVM) [38, 40] is a popular algorithm widely used as an automated process of text classification into predefined classes. Let x_1, x_2, ..., x_l be training examples belonging to one class C, where C is a compact subset of R^N. The SVM classifier can be built using Equation (9).

$$\min \frac{1}{2}\|w\|^2 + \frac{1}{vl}\sum_{i=1}^{l} \xi_i - \rho \quad (9)$$

Subject to Equation (10):

$$(w \cdot \phi(x_i)) \geq \rho - \xi_i = 1, 2, ..., l \;\; \xi \geq 0 \quad (10)$$

By using w and ρ, the SVM algorithm can develop the decision function by Equation (11).

$$f(x) = sign((w \cdot \phi(x_i)) - \rho) \quad (11)$$

In this investigation, the Radial Basis Function (RBF) kernel function is employed with SVM parameters as cost and gamma 100.0 and 0.001, respectively.

Random Forest (RF) introduced by Ho [15] and Singh and Verma [38] in 1995, is a machine learning based on ensemble method for text classification. It was modified in 1999 by Breiman [9]. The basic concept of RF is to create multiple decision trees. This algorithm employs bagging and feature randomness when building each individual tree to build an uncorrelated forest of trees. Using multiple trees for class prediction is more accurate than that of any single tree. This study generated 100 decision trees for our forest.

After evaluating all bug severity predictors via the Area Under The Curve (AUC), the results showed that the RF classifier with *tf-icf* weighting returned the most satisfactory results. Therefore, this predictor was selected for bug report severity analysis to identify the severity of each case after performing the process of assembling dependent bug reports.

4.2. Assembling Dependent Bug Reports followed by Predicting Severity of Bug Reports

This section commences with assembling dependent bug reports, followed by identifying their severity. The overview picture of this method can be seen as Figure 2. Details of each process in the method can be explained as follows.

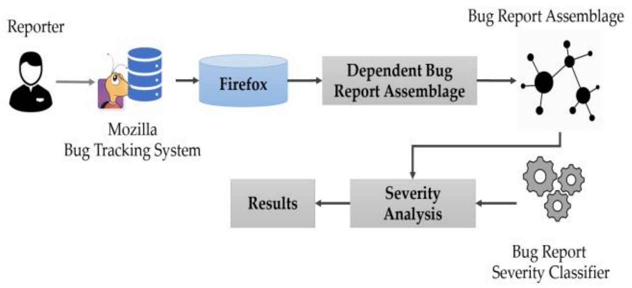

Reporter
Mozilla Bug Tracking System
Firefox
Dependent Bug Report Assemblage
Bug Report Assemblage
Results
Severity Analysis
Bug Report Severity Classifier

Figure 2. Overview of the method assembling dependent bug reports followed by predicting severity of bug reports.

4.2.1. Assemblage of Dependent Bug Reports

Here, threshold-based similarity analysis is proposed as a main method for assembling dependent bug reports into an appropriate cluster. The center point (or centroid) of a specific cluster is a meta-bug report utilized as main factor to assemble a set of relevant bug reports in the same cluster (or group). First, we transformed all bug reports into a suitable format driven on the processing steps discussed in sections 4.1.1 and 4.1.2, respectively. The pre-processed bug reports were then automatically assembled as dependent bug reports into a cluster. The Algorithm (1) depicts the pseudocode for assemblage of dependent bug reports.

In this study, the thresholds were determined as similarity score criteria. Our thresholds are from 0 to 1 with a step of 0.1. This concept was similar to the method used by [13, 33, 34]. If the similarity score between a bug report and a meta-bug report exceeds or equals the threshold, that bug report should be assigned to the cluster with that meta-bug report as the centroid because they may be related. To get the most suitable similarity analysis technique for assemblage of dependent bug reports, two similarity techniques (i.e., cosine similarity and BM25) were compared. The formulas of each similarity analysis technique are explained below.

Algorithm 1: Assemblage of dependent bug reports

```
Input: A collection of meta-bugs, denoted as A
Input: A collection of bug reports, denoted as D
Input: A set of thresholds, {0.1, 0.2, ..., 1.0}
Output: Cluster that has a meta-bug as a centroid and its
dependent bug reports
Parameter: X                 //A ∪ D
Parameter: $mb_i$                //current meta-bug
Parameter: $br_i$                //current bug report
Parameter: similar           //similarity function
Parameter: $cluster_{mi}$         //cluster of $mb_i$
1   Let X be A ∪ D
2   while not the end of A do
3       $mb_i \leftarrow A$            //read the next $mb_i$
4       while not the end of X do
5           $br_i \leftarrow X$          //read the next $br_i$
6           if $r_i \neq mb_i$ then
7               similarity_score ← sim($mb_i$, $br_i$);
8               if similarity_score ≥ threshold then
9                   assign $br_i$ into $cluster_{mi}$ ;
10              end-if
11          end-if
12      end-while
13  end-while
```

Cosine Similarity (CS) is a simple similarity technique. Its formula is defined as Equation (12).

$$sim_{\cos(\theta)}(X_1, X_2) = \frac{x_1 \cdot x_2}{\|x_1\|\|x_2\|} \quad (12)$$

Where X_1 and X_2 are the vectors of words found in bug reports in the collection and the particular meta-bug. If both reports are relevant and similar, the similarity score should be close to 1.

Best Match 25 (BM25) [44] is a similarity ranking function that calculates and ranks a set of documents utilizing the query terms (or words) found in each document. The BM25 formula can be written as Equation (13).

$$BM25(Q,D) = \sum_{q=1}^{|Q|} idf(q_i) \times \left(\frac{tf(q_i,d) \times (k_1+1)}{tf(q_i,d) + k_1 \times (1 - b + b \times \frac{|D|}{DL_{avg}})} \right) \quad (13)$$

Where $tf(q_i,d)$ is the times of occurrences that the q-th term of query (Q) occur in bug report d, while DL_{avg} is the mean length of all documents in the collection. $|D|$ is the word count of bug report D. k_1 and b are free parameters used to balance the weight scores between the term frequency and the normalized bug report length. The value of k_1 should generally be in the range [1.2, 2.0] and the value of b should generally be in the range [0.5, 0.8] [44]. However, in our experiment, the values of k_1 and b parameters were set as 2.0 and 0.8, respectively because these values are the common values used in many previous studies [44].

The $idf(q_i)$ in BM25 is defined as the inverse document frequency and can be computed by Equation (14).

$$idf(q_i) = log\left(\frac{N - df(q_i) + 0.5}{df(q_i) + 0.5}\right) \quad (14)$$

Where *N* is denoted as the total number of bug reports in the collection, and $df(q_i)$ is denoted as the total number of bug reports containing the *q*-th term of *Q*.

Generally, the similarity score should be in the range [0, 1] but the similarity score generated by the BM25 can be greater than 1. Therefore, the data normalization technique is required, with Equation (15) applied to normalize the similarity score of BM25 in the range [0, 1] [3].

$$f(sim\_score) = \frac{sim\_score}{1 + sim\_score} \quad (15)$$

Where *sim_score* is the score of similarity generated by the BM25.

Many previous studies confirmed that BM25 provided the most satisfactory results for analyzing bug reports based on similarity measurements. Therefore, BM25 was also utilized in our study.

4.2.2. Severity Prediction of Bug Reports

After assembling of dependent bug reports, the bug severity predictor described in section 4.1 is performed to identify the severity level of each bug. The overview picture of this process can be presented as Figure 3.

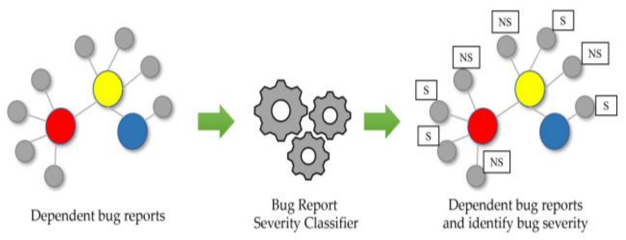

NS
S
NS
NS
S
S
S
NS
Dependent bug reports
Bug Report Severity Classifier
Dependent bug reports and identify bug severity

Figure 3. Process of identify bug severity.

5. The Experimental Results

In this study, F1, True Positive Rate (TPR), True Negative Rate (TNR), False Positive Rate (FPR or 1-TNR), Receiver Operating Characteristic (ROC) curve, and the AUC are metrics to evaluate the performance of assembling dependent bug reports and predicting bug severity [12].

TPR (or recall) calculates the proportion of true positives that are correctly identified, and TNR calculates the proportion of true negatives that are correctly identified. F1 score is the harmonic mean of precision and recall. Then, precision is calculated by dividing the number of true positives by the total number of true positive and false positive. The ROC chooses the most appropriate cut-off value for a test. The optimal cut-off has the highest TPR together with the lowest FPR, while the AUC is used to distinguish the quality of a prediction. In general, the AUC value lies between 0.5 and 1, where 0.5 denotes a bad prediction and 1 denotes an excellent prediction.

5.1. Evaluation for Severity Predictor Models

In this section, bug report severity predictors with *k*-NN, MNB, SVM, and Random Forest (RF) algorithms are evaluated in order to obtain the most appropriate predictors, the measure used in this evaluation is AUC, because if there are multiple predictors, the AUC is widely used to estimate the predictive accuracy of distributional predictors.

The bug report severity predictors with *k*-NN, MNB, SVM, and RF were evaluated and their experimental results can be shown in Table 1 and Figure 4.

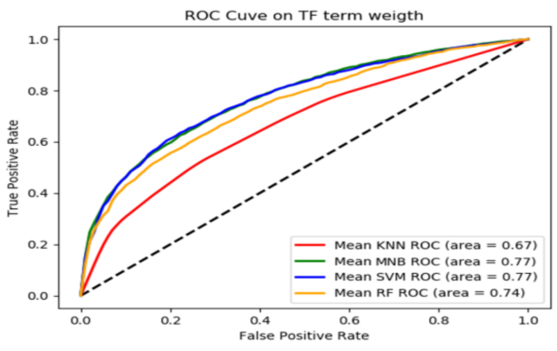
Table 1. The evaluation of severity predictor models by AUC.

Algorithm	**Term weighting schemes**			
	tf	***tf-idf***	***tf-igm***	***tf-icf***
k-NN	0.67	0.63	0.70	0.59
MNB	0.77	0.75	0.72	0.68
RF	0.74	0.74	0.74	**0.97**
SVM	0.77	0.76	0.83	0.72

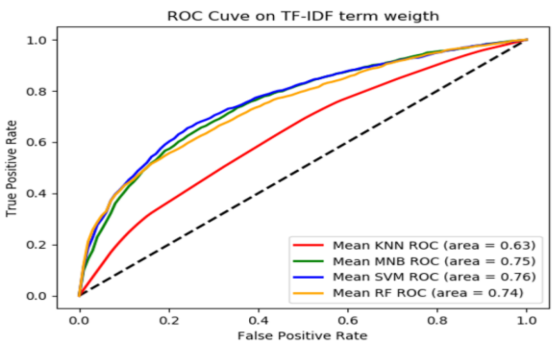
Table 1 showed that the RF predictors with a *tf-icf* weighting scheme returned the most promising results. This is because RF predictors are based on a bagging algorithm and use ensemble learning techniques. RF creates many trees on the data subset and combines the output of all the trees. This ameliorates the overfitting problem in decision trees and also reduces the variance, there by improving the accuracy.

For other predictors, the *k*-NN may have two problems. Firstly, it is difficult to define the value of *k*. If the value of *k* is inappropriate, this may lead to misclassification. If the value of *k* is small, then noise interference will have a higher influence on the result, while if the value of *k* is large the cost of computation increases. Here, we selected the value of *k* as 3. Our results were reasonably good but less than the results of the RF predictors. Secondly, the main concept of *k*-NN is to find the nearest neighbor that delegates equal weight to each feature. This may cause confusion when there are many irrelevant features in the document and lead to poor predictive accuracy. The performance of the *k*-NN predictor using *tf-igm* weighting yielded the best value with its AUC score at 0.70.

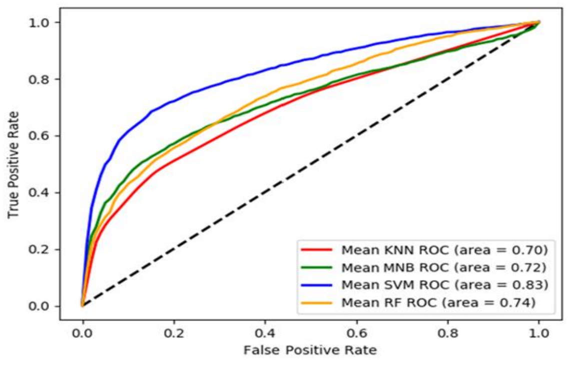
The concept behind MNB is class-conditional independence and this holds if the features of the category members are statistically independent given the true class. Therefore, if the features are co-related, MNB may return unsatisfactory results. MNB also requires a lot of features to accurately learn the predictors. Therefore, if the features in the BOW model are insufficient, the predicted results may also be poor.



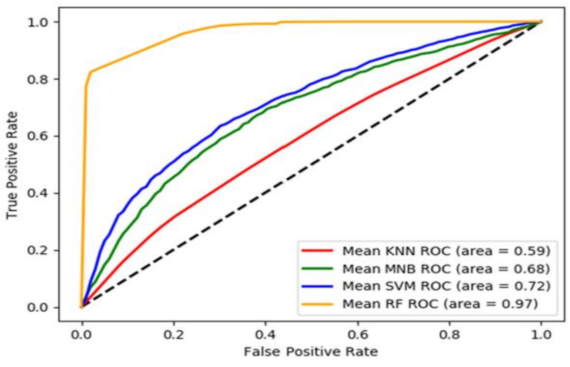
a) ROC curve of the model using *tf* as term weighting schem.



b) ROC curve of the model using *tf-idf* as term weighting schem.



c) ROC curve of the model using *tf-igm* as term weighting schem.



d) ROC curve of the model using *tf-icf* as term weighting schem.

Figure 4. Comparison of each supervised learning algorithm used for modeling bug severity predictors with particular weighting schemes.

In Figure 4-a), the performance of the MNB predictor using *tf* weighting yielded the best value with its AUC score at 0.77. This result was better than the *k*-NN predictors. However, in Figure 4-b), the performance of the SVM predictor using *tf-idf* weighting yielded the best value with its AUC score at 0.76. Meanwhile, consider Figure 4-c). It can be seen that the SVM predictors also returned poorer performance than the RF predictors, possibly because the data used had more noise. Many features of bug reports may have overlapping classes and then the SVM predictors will return inappropriate results. Here, the SVM predictor with *tf-igm* weighting returns the AUC score at 0.83 that is better than the results of *k*-NN and MNB predictors. Lastly, in Figure 4-d), using the RF algorithm with *tf-icf* weighting scheme to model bug severity predictors returned the most satisfactory results with the AUC score at 0.97.

Both the *tf-igm* and *tf-icf* weighting schemes can return good prediction results because a '*word* (or *feature*)' can be discoverable in many classes. However, the same '*word*' found in different classes may indicate disparate importance. Therefore, when using *tf-igm* or *tf-icf* as the weighting scheme, it may be possible to present the particular importance of that '*word*' in an explicit way.

5.2. Evaluation for Assembling Dependent Bug Reports

In Table 2, it can be seen that the most appropriate technique should be BM25, Especially, when using the threshold as 0.5, it can return the most appropriate scores of TPR, TNR, and F1 at 0.78, 0.94, and 0.86 respectively. Then, Figure 5 presents the ROC curves that are used to show the connection between TPR and FPR (1-TNR) for every possible cut-off for a test. As the results, the technique used for assembling dependent bug reports should be BM25 with the threshold as 0.5.

5.3. Evaluation for Bug Report Severity Prediction after Assembling Dependent Bug Reports

After the process of assembling dependent bug report was performed, it returned the TPR score at 0.78. This score indicated the ability of assembling an individual correctly and meant that 3,578 from 4,581 bug reports were correctly identified. The computational time of this process was about 9 minutes.

Table 2. The evaluation of the threshold-based similarity analysis methods.

Threshold	Cosine similarity			BM25		
	TPR	TNR	F1	TPR	TNR	F1
0.1	**0.73**	**0.92**	**0.81**	0.80	0.93	0.86
0.2	0.53	0.98	0.69	0.80	0.93	0.86
0.3	0.36	0.99	0.53	0.80	0.93	0.86
0.4	0.24	1.00	0.39	0.80	0.93	0.86
0.5	0.16	1.00	0.28	**0.78**	**0.94**	**0.86**
0.6	0.06	1.00	0.11	0.71	0.98	0.83
0.7	0.03	1.00	0.06	0.56	0.99	0.72
0.8	0.01	1.00	0.02	0.35	1.00	0.52
0.9	0.00	1.00	0.00	0.04	1.00	0.08
1.0	0.00	1.00	0.00	0.00	1.00	0.00

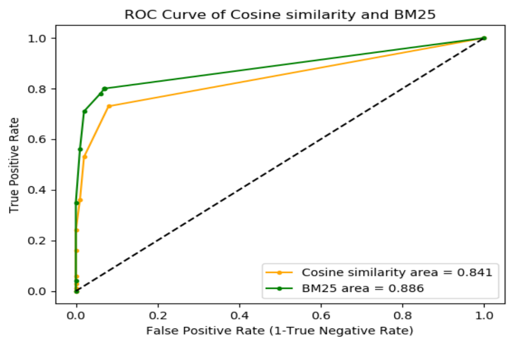


Figure 5. Comparison ROC curve of cosine similarity and BM25.

The 3,578 bug reports that were assembled into individual clusters were then assessed for severity analysis. Score of AUC is 0.89 with computational time of testing process at about 1 minute. Compared to the previous studies such as [35], these results would be acceptable because the results for using the same dataset (Mozilla) of [35] are between 0.798 and 0.924.

If comparing to hand-crafted analysis, previous researchers estimated that out of 350 bug reports submitted daily to Mozilla BTS [37] and bug triagers were only able to manually task and prioritize 300. Bug triagers spent at least 30 minutes to analyze and recognize all the issues in each bug report. Therefore, our proposal may help to reduce analysis time since five bug reports could be analyzed in less than one second.

6. Conclusions

Most previous bug report studies have addressed only one issue. Bug report fixing studies should focus concurrently on multiple issues for a complete and comprehensive analysis. Therefore, this study presents a method of bug report analysis that addresses two points of the problem concurrently. The proposed method consists of two study stages. Firstly, the dependent bug reports are grouped into clusters and then the bug reports in each cluster are analyzed for their severity. Our experimental results determined that the processes of assembling dependent bug reports and identifying their severity were satisfactory. Furthermore, we also compared our proposed method of bug severity analysis with a baseline [19] using the same our dataset. Our method was slightly better than the baseline. This is because, by using CamelCase words together with single words and supervised term weighting (i.e., *tf-igm* and *tf-icf*), this may help to increase class distinguishing power and indicate the specificity of problem domains.

However, bug report assembly analysis cannot be compared because no prior works on this issue are available. Also, when considering computational time, we found that time of analysis for each bug report based on our proposal was less than analysis time by bug triagers. Using our system for direct corrective maintenance activity may help to reduce the cost of maintenance activities that account for over two-thirds of software life cycle costs.

Acknowledgement

This research project was financially supported by Mahasarakham University.

References

[1] Aggarwal K., Timbers F., Rutgers T., Hindle A., Stroulia E., and Greiner R., “Detecting Duplicate Bug Reports with Software Engineering Domain Knowledge,” *Journal of Software: Evolution and Process*, vol. 29, no. 3, pp. e1821, 2017.

[2] Almhana R., Mkaouer W., Kessentini M., and Ouni A., “Recommending Relevant Classes for Bug Reports using Multi-Objective Search,” *in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, Singapore, pp. 286-295, 2016.

[3] Amati G. and Van Rijsbergen C., “Probabilistic Models of Information Retrieval Based on Measuring the Divergence from Randomness,” *ACM Transactions on Information Systems*, vol. 20, no. 4, pp. 357-389, 2002.

[4] Baeza-Yates R. and Ribeiro-Neto B., *Modern Information Retrieval*, Addison Wesley, 1999.

[5] Bettenburg N., Just S., Schröter A., Weiß C., Premraj R., and Zimmermann T., “Quality of bug reports in Eclipse,” *in Proceedings of the OOPSLA Workshop on Eclipse Technology Ex-Change*, Montreal, pp. 21-25, 2007.

[6] Bettenburg N., Just S., Schröter A., Weiss C., Premraj R., and Zimmermann T., “What Makes A Good Bug Report?,” *in Proceedings of the 16th ACM SIGSOFT International Symposium on Found-Ations of Software Engineering*, Atlanta, pp. 308-318, 2008.

[7] Bhattacharya P. and Neamtiu I., “Fine-Grained Incremental Learning and Multi-Feature Tossing Graphs to Improve Bug Triaging,” *in Proceedings of IEEE Inter-National Conference on Software Main-Tenance*, Timisoara, pp. 1-10, 2010.

[8] Bhattacharya P. and Neamtiu I., “Bug-Fix Time Prediction Models: Can We Do Better?,” *in Proceedings of the 8th Working Conference on Mining Software Repositories*, New York, pp. 207-210, 2011.

[9] Breiman L., “Random Forests,” *Machine Learning*, vol. 45, no. 1, pp. 5-32, 2001.

[10] Chen K., Zhang Z., Long J., and Zhang H., “Turning From TF-IDF to TF-IGM For Term Weighting in Text Classification,” *Expert Systems with Applications*, vol. 66, no. 30 pp. 245-260, 2016.

[11] Ferreira I., Cirilo E., Vieira V., and Mourao F.,

"Bug Report Summarization: An Evaluation of Ranking Techniques," *in Proceedings of X Brazilian Symposium on Software Components, Architectures and Reuse*, Maringá, pp. 101-110, 2016.

[12] Gomes L., Torres R., and Côrtes M., "Bug Report Severity Level Prediction in Open Source Software: A Survey and Research Oppor-Tunities," *Information and Software Tech-nology*, vol. 115, pp. 58-78, 2019.

[13] Gopalan R. and Krishna A., "Duplicate Bug Report Detection Using Clustering," *in Proceedings of 23rd Australian Software Engineering Conference*, Milsons Point, pp. 104-109, 2014.

[14] Herzig K., Just S., and Zeller A., "It's Not A Bug, It's A Feature: How Misclassification Impacts Bug Prediction," *in Proceedings of 35th International Conference on Software Engineering*, San Francisco, pp. 392-401, 2013.

[15] Ho T., "Random Decision Forests," *in Proceedings of 3rd International Conference on Document Analysis and Recognition*, Montreal pp. 278-282, 1995.

[16] Jalbert N. and Weimer W., "Automated Duplicate Detection for Bug Tracking Systems," *in Proceedings of IEEE International Conference on Dependable Systems and Networks with FTCS and DCC*, Anchorage, pp. 52-61, 2008.

[17] Jiang S., Pang G., Wu M., and Kuang L., "An Improved K-Nearest-Neighbor Algorithm for Text Categorization," *Expert Systems with Applications*, vol. 39, no. 1, pp. 1503-1509, 2012.

[18] Kanwal J. and Maqbool O., "Bug Prioritization to Facilitate Bug Report Triage," *Journal of Computer Science and Technology*, vol. 27, no. 2, pp. 397-412, 2012.

[19] Kaur S. and Dutta M., "Improved Framework for Bug Severity Classification using N-gram Features with Convolution Neural Network," *International Journal of Recent Technology and Engineering*, vol. 8, no. 3, pp. 1190-1196, 2019.

[20] Kim M., Kim Y., and Kim H., "A Comparative Study of Software Model Checkers as Unit Testing Tools: an Industrial Case Study," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 146-160, 2010.

[21] Kowsari K., Jafari Meimandi K., Heidarysafa M., Mendu S., Barnes L., and Brown D., "Text Classification Algorithms: A Survey," *Information*, vol. 10, no. 4, pp. 150, 2019.

[22] Lamkanfi A., Demeyer S., Giger E., and Goethals B., "Predicting The Severity of A Reported Bug," *in Proceedings of 7th IEEE Working Conference on Mining Software Repositories*, Cape Town, pp. 1-10, 2010.

[23] Lamkanfi A., Demeyer S., Soetens Q., and Verdonck T., "Comparing Mining Algorithms for Predicting the Severity of A Reported Bug," *in Proceedings of 15th European Conference on Software Main-tenance and Reengineering*, Oldenburg, pp. 249-258, 2011.

[24] Lee C., Hu D., Feng Z., and Yang C., "Mining Temporal Information to Improve Duplication Detection on Bug Reports," *in Proceedings of IIAI 4th International Congress on Advanced Applied Informatics*, Okayama, pp. 551-555, 2015.

[25] Lee J., Kim D., and Jung W., "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] Lertnattee V. and Theeramunkong T., "Analysis of Inverse Class Frequency in Centroid-Based Text Classification," *in Proceedings of IEEE International Symposium on Communications and Information Technology*, Sapporo, pp. 1171-1176, 2004.

[27] Limsettho N., Hata H., Monden A., and Matsumoto K., "Automatic Unsupervised Bug Report Cate-Gorization," *in Proceedings of 6th International Workshop on Empirical Software Engineering in Practice*, Osaka, pp. 7-12, 2014.

[28] Luaphol B., Srikudkao B., Kachai T., Srikanjanapert N., Polpinij J., and Bheganan P., "Feature Comparison for Automatic Bug Report Classification," *in Proceedings of International Conference on Com-puting and Information Technology*, Bangkok, pp. 69-78, 2019.

[29] Luaphol B., Polpinij J., and Kaenampornpan M., "Automatic Dependent Bug Reports Assembly for Bug Tracking Systems by Threshold-Based Similarity," *Indonesian Journal of Electrical Engi-neering and Computer Science*, vol. 23, no. pp. 1620-1633, 2021.

[30] Ohira M., Hassan A., Osawa N., and Matsumoto K., "The Impact of Bug Management Patterns on Bug Fixing: A Case Study of Eclipse Projects," *in Proceedings of 28th IEEE International Conference on Software Maintenance*, Trento, pp. 264-273, 2012.

[31] Otoom A., Al-Shdaifat D., Hammad M., and Abdallah E., "Severity Prediction of Software Bugs," *in Proceedings of 7th International Conference on Information and Communication Systems*, Irbid, pp. 92-95, 2016.

[32] Pandey N., Hudait A., Sanyal D., and Sen A., Automated Classification of Issue Reports from A Software Issue Tracker," *in Proceedings of Progress in Intelligent Computing Techniques: Theory, Practice, and Applications*, pp. 423-430, 2018.

[33] Rocha H., Oliveira G., Maques-Neto H., and Valente M., "Nextbug: A Tool for Recommending Similar Bugs in Open-Source Systems," *in Proceedings of V Brazilian*

Conference on Software: Theory and Practice-Tools Track (CBSoft Tools) SBC, Maceio, pp. 53-60, 2014.

[34] Rocha H., De Oliveira G., Marques-Neto H., and Valente M., "Nextbug: A Bugzilla Extension For Recommending Similar Bugs," *Journal of Software Engineering Research and Development*, vol. 3, no. 1, pp. 1-14, 2015.

[35] Roy N. and Rossi B., "Towards An Improvement Of Bug Severity Classification," *in Proceedings of 40th EUROMICRO Conference on Software Engi-Neering and Advanced Applications*, Verona, pp. 269-276, 2014.

[36] Salton G., Wong A., and Yang C., "A Vector Space Model for Automatic Indexing," *Commu-Nications of the ACM*, vol. 18, no. 11, pp. 613-620, 1975.

[37] Shokripour R., Anvik J., Kasirun Z., and Zamani S., "Why So Complicated? Simple Term Filtering and Weighting for Location-Based Bug Report Assignment Recommendation," *in Proceedings of 10th Working Conference on Mining Software Repo-Sitories*, San Francisco, pp. 2-11, 2013.

[38] Singh P. and Verma S., "Multi-Classifier Model for Software Fault Prediction," *The International Arab Journal of Information Technology*, vol. 15, no. 5, pp. 912-919, 2018.

[39] Śliwerski J., Zimmermann T., and Zeller A., "When do Changes Induce Fixes?," *ACM Sigsoft Software Engineering Notes*, vol. 30, no. 4, pp. 1-5, 2005.

[40] Tian Y., Lo D., and Sun C., "Information Retrieval Based Nearest Neighbor Classification for Fine-Grained Bug Severity Prediction," *in Proceedings of 19th Working Conference on Reverse Engineering*, Kingston, pp. 215-224, 2012.

[41] Tian Y., Sun C., and Lo D., "Improved Duplicate Bug Report Identification," *in Proceedings of 16th European Conference on Software Maintenance and Re-Engineering*, Szeged, pp. 385-390, 2012.

[42] Verma T., Renu R., and Gaur D., "Tokenization and Filtering Process in Rapidminer," *International Journal of Applied Information Systems*, vol. 7, no. 2, pp. 16-18, 2014.

[43] Willett P., "The Porter Stemming Algorithm: Then and Now," *Program*, vol. 40, no, 3, pp. 219-223, 2006.

[44] Yang C., Du H., Wu S., and Chen X., "Duplication Detection for Software Bug Reports Based on Bm25 Term Weighting," *in Proceedings of Conference on Technologies and Applications of Artificial Intelligence*, Tainan, pp. 33-38, 2012.

[45] Zhang J., Wang X., Hao D., Xie B., Zhang L., and Mei H., "A Survey on Bug-Report Analysis," *Science China Information Sciences*, vol. 58, no. 2, pp. 1-24, 2015.

[46] Zhou Y., Tong Y., Gu R., and Gall H., "Combining Text Mining and Data Mining for Bug Report Classification," *Journal of Software: Evolution and Process*, vol. 28, no. 3, pp. 150-176, 2016.

Bancha Luaphol received Ph.D. degree in Computer Science from Mahasarakham University. He currently works for Department of Digital Technology, Faculty of Administrative Science, Kalasin University, Thailand. He is currently engaged in the study of applications of natural language processing, and machine learning and deep learning approach.

Jantima Polpinij received Ph.D. degree in Computer Science from University of Wollongong, Australia. She is an associate professor of computer science at Mahasarakham University, Thailand. Her research interest includes data science, natural language processing, text mining, and machine learning and deep learning approach.

Manasawee Kaenampornpan received Ph.D. degree in Computer Science from University of Bath, UK. She is an assistant professor of computer science at Mahasarakham University, Thailand. Her research interests are user experience design, context awareness, mobile and ubiquitous computing.