**Indonesian Journal of Electrical Engineering and Computer Science**
Vol. 23, No. 3, September 2021, pp. 1620~1633
ISSN: 2502-4752, DOI: 10.11591/ijeecs.v23.i3.pp1620-1633

# Automatic dependent bug reports assembly for bug tracking systems by threshold-based similarity

**B. Luaphol, J. Polpinij, M. Kaenampornpan**
Department of Computer Science, Faculty of Informatics, Mahasarakham University, Khamriang, Thailand

**Article Info**

***Article history:***

Received Oct 14, 2020
Revised Jul 8, 2021
Accepted Aug 3, 2021

***Keywords:***

BM25
Bug dependency issue
Bug report
Cosine similarity
MATF
Threshold-based similarity analysis

**ABSTRACT**

Bug reports contain essential information for fixing problems that occur in software. Many studies have proposed methods for automatic analysis of bug reports. One such task could affect the completion of software bug fixing, known as “bug dependency”. Although this problem was mentioned by many researches, most of them discussed about the related bugs but not really dealt with dependency issue in bug reports. One possible solution used for addressing this issue is to assemble all relevant/dependent bug reports together before analysis of the next processing stages. This study presents a method of assembling dependent bug reports. The main mechanism is called “threshold-based similarity analysis”, and the three similarity techniques of cosine similarity (CS) multi aspect TF (MATF), and BM25 are compared with feedback, precision and likelihood value. As the BM25 with the threshold as 0.5 gives the best results, it was used to compare with the state of the art method. The results show that our method increases precision and likelihood values by 12% and 12.4% respectively. Therefore, our results can be used to encourage developers to recognize all dependent bugs in the same problem domain.

***Corresponding Author:***

Bancha Luaphol
Department of Computer Science
Faculty of Informatics, Mahasarakham University
Khamriang Sub-District, Kantarawichai District
Maha Sarakham 44150 Thailand
Email: bancha.lu@ksu.ac.th

## 1. INTRODUCTION

Locating and tracking bugs in large open-source software systems are never an easy process, that involves gathering bug reports from global end-users. To facilitate report gathering, many bug tracking systems (BTS) such as bugzilla, trace, and jira have been developed [1]-[7]. Large amounts of data are submitted daily to these BTS as bug reports.

The incidence of bug reports has continuously increase, with the size of bug report repositories mushrooming, resulting in the necessity for automatic data handling and analysis. Therefore, many studies on bug report have been proposed. In literature review, bug report studies can be divided into three major areas as bug report optimization, bug report triage and bug fixing [7]. The first area, called bug report optimization, aims to improve quality of bug reports and reduce the amount of erroneous information. It can be classified into three tasks as content optimization [8], [9], misclassification [10]-[14] and severity prediction [15]-[23]. The second area is bug report triage. This study area concerns duplicated bug detection [3], [5], [6], [24]-[27], prioritization [28], [29], and suitable developer assignment tasks [30]-[32]. Lastly, bug fixing can also be classified into three main tasks as bug localization [33]-[35], recovering links between bug reports and changes in files or source code [36]-[38], and prediction of bug fixing time [2], [39], [40].

Besides the aforementioned tasks, other interesting bug report topics remain. One such task could affect the completion of software bug fixing, recognized as the "bug dependency issue" [2], [4]. This issue can be described by a situation in which an unfixed bug "a" affects bug "b". That is, bug "b" continues to occur despite it is being fixed if bug "a" is not yet completely fixed. Despite this issue has been described by various authors [2]-[4], [6], it has not yet been earnestly studied. This may be because the performance improvements are still required for bug report misclassification, severity and priority prediction, bug duplicated detection, bug localization, and bug fixing tasks [7]. Nonetheless, the bug dependency issue should be resolved simultaneously along with the aforementioned tasks in order to complete the bug fixing process.

From our perspective, a potential solution for handling the bug dependency issue is to assemble dependent bug reports into the same group. This is because this solution may help developer teams to identify the overall picture the problems from that group. As a result, this could provide opportunities to further fix those software bugs completely.

Unfortunately, assigning related dependent bug reports with identical problems into the same group is currently handcrafted analysis done by bug triagers [4] who are software experts [41]. Bug triagers begin this process by identifying the main bug report referred to as the "meta-bug report" [4]. The meta-bug report is used as the center point and utilized to find bug reports that regard the same problem domain found in the content theme of the center point [4]. Those relevant bug reports are then grouped into the same cluster, referred to as "dependencies" [42]. When this process is manually performed, it becomes a time-consuming process.

After literature review, it was found that the closest study related to automatic bug dependency analysis was proposed by Rocha *et al*. [43], [44]. They presented a method for recommending similar bug reports to the report under consideration. Their proposed method may help the developer team to recognize bug reports that should also be considered. This method was used as a main mechanism in a system, called NextBug, that was used to find similar bug reports for recommendation. However, this system recommended only a certain number of retrieved bug reports for each analysis because they used the maximum number of retrieved bug reports as 1 to 5. As a result, the overall picture of a problem point in the software might be overlooked since all dependent or similar bug reports were not retrieved simultaneously. This might result in an incomplete fix of all bugs found in the considered problem point and some bugs may still occur.

Consequently, handling the problem of bug dependency is a major challenge in this study. If the overall picture of the software problem could be captured simultaneously, then this could lead to an improvement in bug fixing. Therefore, this work proposes a method for automatically assembling dependent bug reports to assist the developer team when considering and analyzing all bug reports concurrently. A method of assembling dependent bug reports is proposed, called "threshold-based similarity analysis". Two feature types namely unigram and unigram+CamelCase, and three similarity techniques namely cosine similarity (CS), multi aspect TF (MATF), and BM25 were compared. Finally, the best model from our proposed method is also selected and compared with the state-of-the-art method proposed by Rocha *et al*. [43], [44]. The article is organized as follows. In Section 2, it is the proposed method. The experimental results and discussion are presented in Section 3. Finally, the conclusion is in Section 4.

## 2. RESEARCH METHOD

An overview of the proposed method is shown as Figure 1. Firstly, the Firefox bug reports were collected from the Mozilla bug tracking system. Later, those bug reports were transformed into the pre-processing stage to identify their features. In the next stage, they were represented in the form of a vector space model (VSM), and their features were weighted using a term weighting scheme. Next, the bug reports formatted with the VSM were assembled as a set of dependent bug reports that is related to each meta-bug report. The proposed method was called "threshold-based similarity analysis", where the threshold value is used as a criterion to make decisions of similarity between bug reports and meta-bug reports. After analyzing the results in the evaluation stage, the most appropriate model was selected and compared with the state-of-the-art method proposed by Rocha *et al.* [43], [44].

### 2.1. Dataset collection

The dataset used here was gathered from Bugzilla, while bug reports relating to Mozilla Firefox were downloaded between 1 September and 30 November 2019. The dataset consisted of 22,000 bug reports. However, the dataset used in this study contained 11,059 reports, consisting of 478 meta-bug reports, and 10,581 bug dependency with meta-bug reports. It is noted that only 478 meta-bug reports were used because the other meta-bug reports had dependent bug reports with fewer than two bug reports and were therefore ignored. Here, all bug report statuses [7], [45] except unconfirmed status are used for our experiment because these were confirmed by bug triagers, software developers, and software testers as “real” bug reports [7], [40], [45], [46]. A bug report example can be presented as Figure 2.

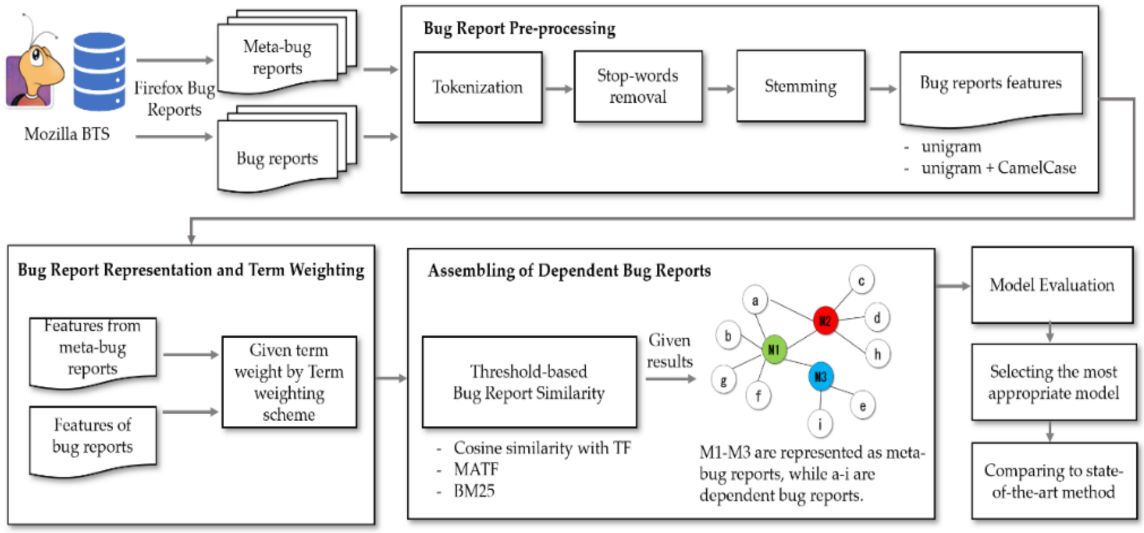
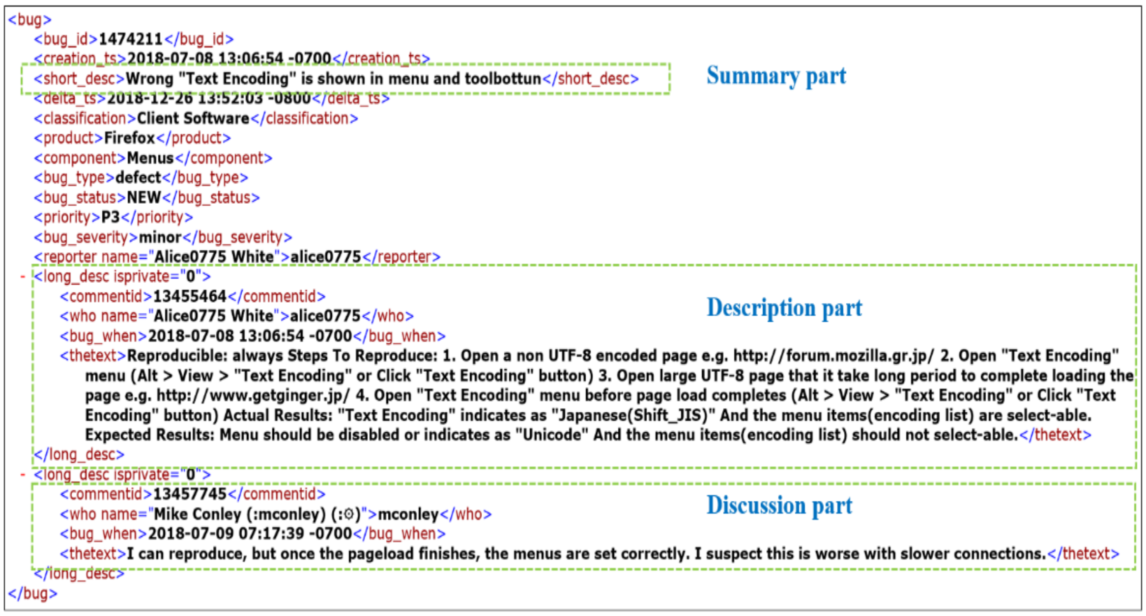

Mozilla BTS
Firefox Bug Reports
Meta-bug reports
Bug reports
Bug Report Pre-processing
Tokenization
Stop-words removal
Stemming
Bug reports features
- unigram
- unigram + CamelCase
Bug Report Representation and Term Weighting
Features from meta-bug reports
Features of bug reports
Given term weight by Term weighting scheme
Assembling of Dependent Bug Reports
Threshold-based Bug Report Similarity
- Cosine similarity with TF
- MATF
- BM25
Given results
M1
M2
M3
a
b
c
d
e
f
g
h
i
M1-M3 are represented as meta-bug reports, while a-i are dependent bug reports.
Model Evaluation
Selecting the most appropriate model
Comparing to state-of-the-art method

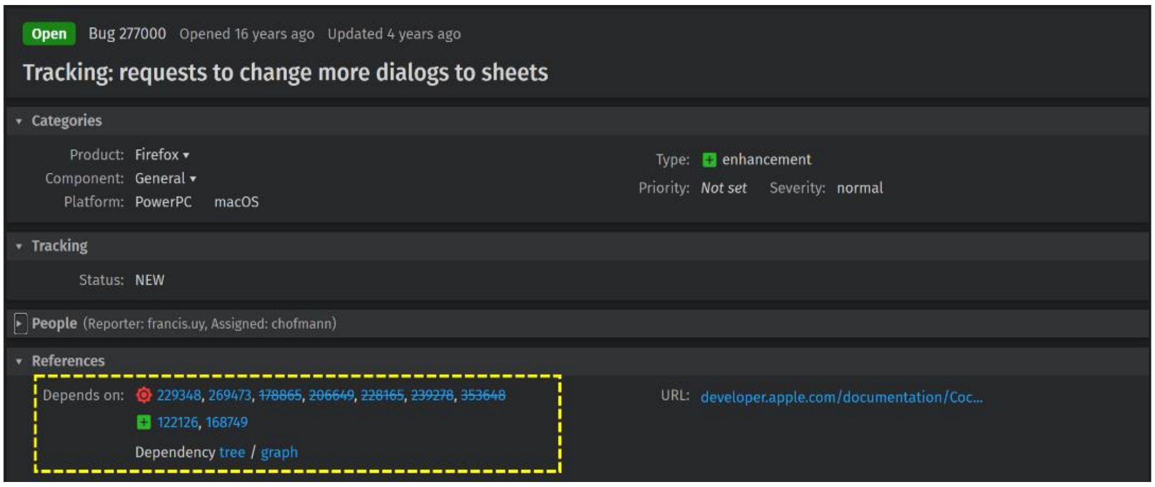
Figure 1. Overview of the proposed method



```
<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 2. An example of bug reports in the XML format

Generally, a bug report consists of three elementary parts namely 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, numerous studies related to bug reports deploy only the summary because this part contains less noise [3], [14], [47]. Therefore, here, we also investigated only the summary part. It is also noted that each bug report contains its labels representing its meta-bug reports. Then, one bug report can be in many meta-bug reports. In addition, a meta-bug report can be a dependent bug report in others meta-bug reports. Figure 3 shows an example in our dataset. Finally, this dataset was used in both the proposed and compared methods, and these were set in the same environment.



Open Bug 277000 Opened 16 years ago Updated 4 years ago

Tracking: requests to change more dialogs to sheets

Categories

Product: Firefox

Component: General

Platform: PowerPC macOS

Type: enhancement

Priority: Not set Severity: normal

Tracking

Status: NEW

People (Reporter: francis.uy, Assigned: chofmann)

References

Depends on: 229348, 269473, ~~178865~~, ~~206649~~, ~~228165~~, ~~239278~~, ~~353648~~

122126, 168749

Dependency tree / graph

URL: developer.apple.com/documentation/Coc...

Figure 3. A meta-bug report example and its dependent bug reports

## 2.2. Bug report pre-processing

The first stage of bug report pre-processing is text tokenization. This process separates text as tokens [48] that in this study are called "words". Bug report features (or words) used here are unigram and CamelCase. Unigram means a single word, while CamelCase [10], [13], [33]-[35] (also referred to as Snakecase or Compound words) is to write a word by using two words or abbreviations together to yield a new meaning with no punctuation and intervening spaces. Some CamelCase words often begin with a capital letter or use the capital letter in the middle of words. Some examples of CamelCase are "browser_views" and "UrlBar". Before using CamelCase words as features, these words are split into single words [10], [13], [33]-[35]. For examples, CamelCase words such as "browser_views" and "UrlBar" can be split as "browser", "views", "Url", and "Bar", respectively. Finally, we use both the original CamelCase words and their single words as features. By doing this, it seems to expand the bug report features.

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 [10], [12], [13], [15]-[17], [19], [33]-[35], [49], [50]. Two different types of features are used to obtain the most satisfactory results, namely unigram and unigram+CamelCase. 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 inflected words to their base or root form, called 'word stem' [51]. An example of bug report features after pre-processing can be shown as Table 1.

Table 1. Example of bug report features after pre-processing stage

| Input | Bug Report Summary Part | Accessibility label for Search Engine AutoComplete item is wrong |
|---|---|---|
| Output | Unigram | accessibl/label/search/engine/autocomplet/item/wrong |
| | Unigram+Camelcase | accessibl/label/search/engine/autocomplet/item/wrong/autocomplet/auto/complete |

## 2.3. Bug report representation and term weighting

After the meta-bug reports and the bug reports are pre-processed, they are represented with the format of VSM. Simply speaking, this VSM becomes word vectors of the meta-bug reports and the bug reports. In general, each term that occurs in the VSM should include its weight. The term weighting scheme used in this study is term frequency ($tf$), where the local weight designates the significance of a term within the overall bug report. Antoniol *et al*. [10], and Jalbert and Weimer [3] mention that this weighting scheme is sufficiently satisfactory for the bug report study area. The $tf$ formula can be:

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

where $f_{t,d}$ is the number of times that term $t$ appears in bug report, denoted as $d$.

It is noted that the VSM with $tf$ weight is used in the case of assembling dependent bug reports by the CS only. However, if assembling dependent bug reports by BM25 or MATF, these techniques do not require the VSM with $tf$ weight, but they instead require the VSM with the raw frequency of each term.

### 2.4. Assembling of dependent bug reports

The similarity analysis is a technique that is often used in the area of bug report studies. By using this technique alone, it can cause high false negative rate because it tends to return the irrelevant bug reports [24]. Previous studies in this area [24], [43], [44] have shown that a threshold-based approach for similarity analysis can improve the similarity analysis performance. However, those works manually defined a fixed number as a threshold value for bug report analysis. By doing this, it might lead to have an improper threshold. Therefore, this study proposes a method, called "threshold-based similarity analysis", to assemble the bug reports to the related meta-bug report. In general, most studies in this area often utilize CS as the similarity analysis technique. However, the CS does not work efficiently with nominal data [52]. Then, the summary part of the bug report that is used in this study is quite small. Consequently, the CS might return unsatisfactory results for our study due to its small dataset.

As mentioned above, this study performed the threshold-based similarity analysis with three similarity techniques, namely CS, MATF, and BM25. This is because it was necessary to have the most appropriate mechanism to estimate the similarity between the meta-bug reports and all the bug reports. The detail of each similarity technique is described. Then the paper describes the detail of our proposed method of dependent bug report assemblage.

#### 2.4.1. Cosine similarity approach for text similarity

The CS has been widely used for bug localization and bug duplication detection [24], [33]-[35]. Therefore, we also applied this similarity technique to assemble the dependent bug reports. The CS formula is:

$$sim_{cos(\theta)}(V_1, V_2) = \frac{v_1 \cdot v_2}{\|v_1\|\|v_2\|} \tag{2}$$

where $V_1$ and $V_2$ are the term vectors of a pairwise between the particular meta-bug report and bug reports in the dataset. The similarity result should be close to 1 if both reports are similar.

#### 2.4.2. MATF approach for text similarity

MATF is a technique to determine the similarity between documents, and was proposed by J.H. Paik [53] in 2013. This technique may aid handling problem of document length difference, where this problem affects the document ranking. This technique was designed and developed to focus on both short and long documents. Simply, although lengthy documents in a collection favor retrieving and ranking documents, shorter documents are also respected. In this study, the document refers to a bug report.

The MATF similarity combines two main variables: term frequency factors (TFF) and term discrimination factor (TDF). The MATF similarity is:

$$MATF(Q, D) = \frac{\sum_{i=1}^{|Q|} TFF(q_i, D) \times TDF(q_i, C)}{\sum_{i=1}^{|Q|} TDF(q_i, C)} \tag{3}$$

in theory, TFF combines two $tf$ aspects: relative the intra-document (RITF); and length regularized TF (LRTF). The RITF is a value measured by considering term frequency, denoted as $tf(q_i, D)$, relative to the average tf of the document, denoted as $Avg.tf(D)$. Therefore, the RITF formula is:

$$RITF(q_i, D) = \frac{log_2(1 + tf(q_i, D))}{log_2(1 + Avg.tf(q_i, D))} \tag{4}$$

meanwhile, LRTF is a value that normalizes the $tf$ by quantifying terms that are present in a document. The LRTF can be:

$$LRTF = (q_i, D) = tf(q_i, D) \times log_2\left(1 + \frac{ADL(C)}{len(D)}\right) \tag{5}$$

where $ADL(C)$ is the average document length of the collection, while $len(D)$ is the document length $D$.

After obtaining the values of RITF and LRTF, those values will be normalized by following function [54], [55].

$$f(x) = x/(1 + x) \tag{6}$$

This function normalizes the values of RITF and LRTF upper bound to 1 [54], [55]. Finally, the formulas of RITF and LRTF should be:

$$BRITF(q_i, D) = \frac{RITF(q_i, D)}{1 + RITF(q_i, D)} \tag{7}$$

$$BLRTF(q_i, D) = \frac{LRTF(q_i, D)}{1 + LRTF(q_i, D)} \tag{8}$$

consequently, the formula of TFF can be:

$$TFF(q_i, D) = w \times BRITF(q_i, D) + (1 - w) \times BLRTF(q_i, D) \tag{9}$$

where $w$ is calculated by considering the number of term words in a query. It is noted that our query is a meta-bug report. The formula used to calculate $w$ is:

$$w = \frac{2}{1 + log_2(1 + |Q|)} \tag{10}$$

then, $|Q|$ is the total number of terms found in the query. However, the value of $w$ should be between 0 and 1.

TDF serves to assign a higher score to the documents containing rare terms in the collection and combines two tf aspects: Inverse document frequency (IDF); and average elite set term frequency (AEF). In this case, the IDF then applies the standard IDF measure. Its formula is

$$IDF(q_i, C) = log\left(\frac{CS(C) + 1}{df(q_i, C)}\right) \tag{11}$$

in this study, $CS(C)$is the entire number of bug reports in the collection, while $df(q_i, C)$ is the number of bug reports containing term q-th. Meanwhile, the AEF can be defined as

$$AEF(q_i, C) = \frac{CTF(q_i, C)}{df(q_i, C)} \tag{12}$$

where $CTF(q_i, C)$ is defined as the total occurrence of the terms q-th of $Q$ in the collection. After obtaining the value of AEF, this value should be normalized using the formula (6). Finally, the formula of TDF can be:

$$TDF(q_i, D) = IDF(q_i, C) \times \frac{AEF(q_i, C)}{1 + AEF(q_i, C)} \tag{13}$$

### 2.4.3. BM25 approach for text similarity

BM25 is a well-known ranking function that ranks matching relevant documents according to their relevance to a given search query ($Q$), regardless of the inter-relationship between the query terms within a document [56], [57]. It notices that 'query' in this study referred to meta-bug report. The BM25 formula is:

$$BM25(Q, D) = \sum_{i=1}^{|Q|} idf(q_i) \times \left(\frac{tf(q_i, D) \times (k_1 + 1)}{f(q_i, D) + k_1 \times (1 - b + b \times \frac{|D|}{DL_{avg}})}\right) \tag{14}$$

in this study, $tf(q_i, D)$ is the term frequency. It is used to define the number of times of the query term q-th appearing in bug report document $D$. While $|D|$ is defined as the length of bug report document $D$ and $DL_{avg}$ is the average length of all bug reports in the collection. $b$ is the free parameter of the normalization method for $tf(q_i, D)$. It is only valid within [0, 1] but The standard setting for $b$ should be $0.5 < b < 0.8$ [56], [58]. While $k_1$ is also the free parameter used to control the value given by $(1 - b + b \times \frac{|D|}{DL_{avg}})$. The standard setting for $k_1$ should be 1.2 [56], [58]. However, the most common settings of $k_1$ and $b$ should be 2.0 and 0.8 respectively [56]. Consider $k_1$, where $idf$ referred to the inverse document frequency of the term q-th of $Q$. Its formula is:

$$idf(q_i) = log\left(\frac{N - df(q_i) + 0.5}{df(q_i) + 0.5}\right) \tag{15}$$

in this study, $N$ represents the whole number of bug reports in the collection. While $df(q_i)$ is the number of bug reports holding the term q-th of $Q$.

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 showed as the formula (6) also applies in this case.

### 2.4.4. The proposed method: threshold-based similarity analysis

To obtain the most appropriate model for assembling the dependent bug reports, we also provide thresholds to determine the similarity score. These thresholds are from 0 to 1 with step 0.1. This concept is similar to Gopalan and Krishna [24], and Rocha *et al*. [43], [44]. When the similarity score of the meta-bug report and a bug report 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 meta-bug report and a bug report is below the threshold, those bug reports may be irrelevant. The pseudocode of assembling the dependent bug reports is presented as Algorithm 1.

```
Algorithm 1. Assembling of dependent bug reports with a threshold-based similarity
analysis
Input: $M$ is a set of meta-bug reports
Input: $B$ is a set of bug reports
Input: $T$ is a set of thresholds, {0.1, 0.2, 0.3, …, 1.0}
Output: Clusters of each meta-bug report and its relevant bug reports
Parameter: $R$: a set of $M \cup B$
Parameter: $m_i$: the current meta-bug report that is analyzed
Parameter: $r_i$: the current bug report that is analyzed
Parameter: $Sim$: Similarity measure with similarity analysis techniques {CS, MATF,
BM25}
Parameter: $C_{mi}$: Cluster of $m_i$
Let $R$ be $M \cup B$
while not end of $M$ do
    $m_i \leftarrow M$ //read the next meta-bug report;
    while not end of $R$ do
        $r_i \leftarrow R$ //read the next bug report;
        similarity score $\leftarrow$ $Sim(m_i, r_i)$; //$r_i \neq m_i$
        if similarity score $\geq T$ then
            Add $r_i$ into $C_{mi}$;
        end
    end
end
```

In Algorithm 1, a meta-bug report is considered as the centroid for each cluster. Beginning with the bug reports in the corpus, the similarity of each bug report with the centroid of the existing clusters is computed as a similarity score of the summary part. If the maximum of the similarity scores for the bug report with the centroid is over a given threshold, it is inserted into the cluster. If the maximum similarity is below the threshold, that bug report may be analyzed with other clusters. 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. Figure 4 shows that the bug report 'a' is dependent on both meta-bug reports M1 and M2.

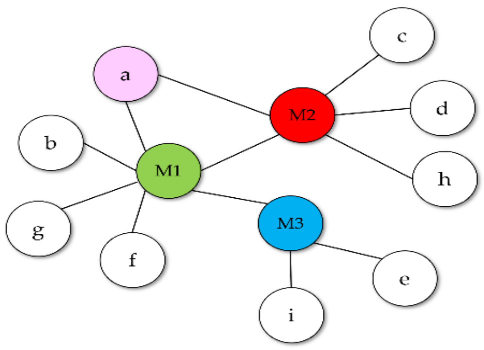
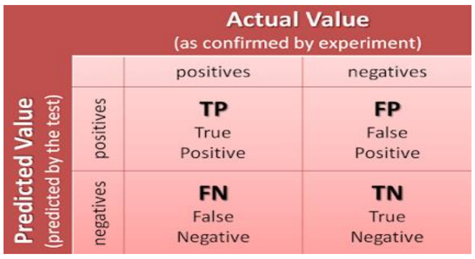


Figure 4. Expected results representing meta-bug reports and their dependent bug reports

## 3. RESULTS AND DISCUSSION

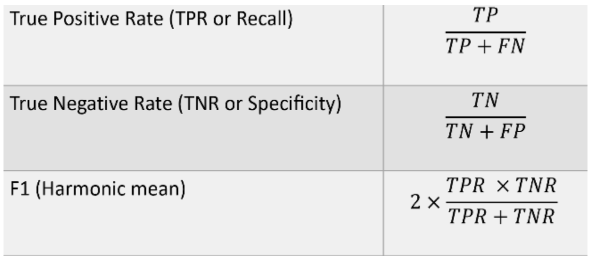
### 3.1. The measurements techniques for evaluation

This section presents the experimental results of the automatic dependent bug report assemblage by using true positive rate (TPR) [3], [5], [15], [16], [24], [59]-[63], true negative rate (TNR) [3], [5], [24], [62], [63], and F1 [5], [24], [59], [60]. TPR (also called sensitivity or recall) measures the proportion of actual positives that are correctly identified. Meanwhile, the TNR (or specificity) measures the proportion of actual negatives that are correctly identified. Finally, the F1 is the harmonic mean of the TPR and TNR. This measure is used to determine test accuracy. The best value for F1 is 1 and the worst value is 0. Consider the confusion matrix, shown as Figure 5(a), and then the formulas of those measurement techniques can be summarized as Figure 5(b).



| Predicted Value (predicted by the test) | | Actual Value (as confirmed by experiment): positives | Actual Value (as confirmed by experiment): negatives |
|---|---|---|---|
| | positives | **TP** True Positive | **FP** False Positive |
| | negatives | **FN** False Negative | **TN** True Negative |

(a)



| | |
|---|---|
| True Positive Rate (TPR or Recall) | $\frac{TP}{TP+FN}$ |
| True Negative Rate (TNR or Specificity) | $\frac{TN}{TN+FP}$ |
| F1 (Harmonic mean) | $2 \times \frac{TPR \times TNR}{TPR+TNR}$ |

(b)

Figure 5. The confusion matrix and the formulas of TPR, TNR, and F1

Furthermore, the results of TPR and TNR can be used to analyze the receiver operator characteristic (ROC) curve and area under curve (AUC), respectively. The ROC curve [3], [15], [16], [59], [60], [62]-[64] is used to measure how well a dependent bug report can be detected from a dataset of bug reports, while the AUC [15], [16], [59], [60], [62]-[64] represents the degree or measure of separability. The ROC curve is plotted with TPR against the false positive rate (FPR or 1-TNR) [3], [15], [16], [60], with TPR on the y-axis and FPR on the x-axis. The area under the ROC curve is termed AUC. These measurements are two of the most important evaluation metrics 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.

However, to compare our proposed with measurement techniques used in the-state-of-the-art method, feedback, precision, and likelihood [43], [44], [65] must be included. Formulas for feedback, precision and likelihood are explained as detailed below; however, before presenting these formulas to calculate the metric, the following sets require definition. Let $BR_q$ be the set of dependent 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 dependent bug reports with their answers. Meanwhile, $Z$ is the total number of meta-bug reports at 478 in total, and $Z_k$ is a subset of $Z$ that can retrieve the dependent bug reports at least $k$. These definitions help to define feedback, precision, and likelihood. Feedback involves measuring the number of bug reports that are retrieved when using a given query as a meta-bug report. Formally, the feedback of $k$, denoted as $FB(k)$, is the percentage of queries with at least $k$ bug reports retrieved. The feedback formula can be defined as:

$$FB(k) = \frac{|Z_k|}{Z} \tag{16}$$

for example, suppose a system performed 10 meta-bug reports as queries ($|Z| = 10$). If all these meta-bug reports each returned at least 1 relevant bug report, then feedback for $k = 1$. Thus, $FB(k = 1)$ would be 100%. Conversely, if only 3 of the meta-bug reports returned at least 3 bug reports, then $FB(k = 3)$ would be 30%.

Precision, denoted as $P(k)$, measures the ratio of dependent bug reports that are retrieved. The formula for precision can be expressed as:

$$P_q(k) = \frac{BR_q(k) \cap R_q}{BR_q(k)} \tag{17}$$

In addition, overall precision in our dataset collection is defined as the average precision achieved for each meta-bug report that is considered as query. The formula of average precision can be presented as:

$$P(k) = \frac{1}{|Z_k|} \sum_{q \in Z_k} P_q\,(k) \tag{18}$$

suppose that a meta-bug report returned 5 bug reports. If the first bug report is relevant, then P(1)=100%; otherwise, if the first bug report is not relevant, then P(1)=0%. Besides, if among all 5 bug reports only the third one is relevant, then the precision values would be P(2)=0%, P(3)=33%, P(4)=25%, and P(5)=20%.

The likelihood is a binary measure, denoted as $L(k)$. It is a common measure used to assess the advantage of retrieving relevant bug reports. In this context, the likelihood checks whether there is a dependent bug report among the top-*k* suggested issues. The likelihood of the top-*k* dependent bug reports can be defined as:

$$L_q(k) = \begin{cases} 1 \text{if} BR_q(k) \cap R_q \neq \emptyset \\ 0 \text{otherwise} \end{cases} \tag{19}$$

if at least one relevant bug report exists among the top-$k$ bug reports that are retrieved, the answer is returned one; if not, the return is zero. The overall likelihood in our dataset collection, defined as the average likelihood measured for each meta-bug report, can be represented as:

$$L(k) = \frac{1}{|Z_k|} \sum_{q \in Z_k} L_q(k) \tag{20}$$

## 3.2. Evaluation of the proposed method

Table 2 and Table 3 present the experimental results of the proposed method. The dataset used for our experiment consisted of 478 meta-bug reports and 10,581 bug reports dependent on these meta-bug reports. Table 2 presents the experimental results of assembling dependent bug reports using unigram as features, while Table 3 presents the experimental results of assembling dependent bug reports using unigram+CamelCase as features.

Table 2 shows the experimental results using unigram as features. BM25 with thresholds of 0.1-0.4 returned the best results for TPR, TNR, and F1, while MATF with thresholds of 0.1-0.3 returned the best results for TPR, TNR, and F1. However, when considering the evidence of CS, a threshold of 0.1 returned the best results for TPR, TNR, and F1. The best TPR, TNR, and F1 scores of assembling dependent bug reports were 0.654, 0.921 and 0.765, respectively.

However, when considering Table 3, it is the experimental results when using unigram+CamelCase as features. The evident of Cosine Similarity and MATF returned the similar results shown as Table 2. Nevertheless, the BM25 returned the best results for TPR and F1, when using the threshold at 0.5. Then, it returns the best TPR, TNR, and F1 scores of assembling dependent bug reports at 0.696, 0.918, and 0.792 respectively.

Table 2. The experimental results when using unigram as features

| Threshold | CS | | | MATF | | | BM25 | | |
|---|---|---|---|---|---|---|---|---|---|
| | TPR | TNR | F1 | TPR | TNR | F1 | TPR | TNR | F1 |
| 0.1 | 0.651 | 0.925 | 0.764 | 0.654 | 0.921 | 0.765 | 0.654 | 0.921 | 0.765 |
| 0.2 | 0.485 | 0.978 | 0.648 | 0.654 | 0.921 | 0.765 | 0.654 | 0.921 | 0.765 |
| 0.3 | 0.312 | 0.995 | 0.475 | 0.654 | 0.921 | 0.765 | 0.654 | 0.921 | 0.765 |
| 0.4 | 0.195 | 0.998 | 0.326 | 0.653 | 0.921 | 0.764 | 0.654 | 0.921 | 0.765 |
| 0.5 | 0.110 | 0.999 | 0.198 | 0.425 | 0.960 | 0.589 | 0.643 | 0.930 | 0.760 |
| 0.6 | 0.058 | 1.000 | 0.110 | 0.021 | 0.998 | 0.041 | 0.576 | 0.964 | 0.721 |
| 0.7 | 0.026 | 1.000 | 0.051 | 0.000 | 1.000 | 0.000 | 0.428 | 0.990 | 0.598 |
| 0.8 | 0.007 | 1.000 | 0.014 | 0.000 | 1.000 | 0.000 | 0.213 | 0.999 | 0.351 |
| 0.9 | 0.001 | 1.000 | 0.002 | 0.000 | 1.000 | 0.000 | 0.019 | 1.000 | 0.037 |
| 1.0 | 0.001 | 1.000 | 0.002 | 0.000 | 1.000 | 0.000 | 0.000 | 1.000 | 0.000 |

Table 3. The experimental results when using unigram+CamelCase as features

| Threshold | CS | | | MATF | | | BM25 | | |
|---|---|---|---|---|---|---|---|---|---|
| | TPR | TNR | F1 | TPR | TNR | F1 | TPR | TNR | F1 |
| 0.1 | 0.688 | 0.918 | 0.787 | 0.696 | 0.918 | 0.792 | 0.696 | 0.918 | 0.792 |
| 0.2 | 0.496 | 0.979 | 0.658 | 0.696 | 0.918 | 0.792 | 0.696 | 0.918 | 0.792 |
| 0.3 | 0.322 | 0.995 | 0.487 | 0.696 | 0.918 | 0.792 | 0.696 | 0.918 | 0.792 |
| 0.4 | 0.205 | 0.998 | 0.340 | 0.695 | 0.908 | 0.787 | 0.696 | 0.918 | 0.792 |
| 0.5 | 0.123 | 0.999 | 0.219 | 0.461 | 0.951 | 0.621 | 0.696 | 0.918 | 0.792 |
| 0.6 | 0.058 | 1.000 | 0.110 | 0.024 | 0.997 | 0.047 | 0.610 | 0.960 | 0.746 |
| 0.7 | 0.029 | 1.000 | 0.056 | 0.000 | 1.000 | 0.000 | 0.459 | 0.988 | 0.627 |
| 0.8 | 0.009 | 1.000 | 0.018 | 0.000 | 1.000 | 0.000 | 0.249 | 0.998 | 0.399 |
| 0.9 | 0.001 | 1.000 | 0.002 | 0.000 | 1.000 | 0.000 | 0.037 | 1.000 | 0.071 |
| 1.0 | 0.001 | 1.000 | 0.002 | 0.000 | 1.000 | 0.000 | 0.000 | 1.000 | 0.000 |

As described above, it was not possible to specify the best threshold for BM25 and MATF since the thresholds which had the best performance of for these techniques are between 0.1 and 0.5. To specify the best threshold for CS, BM25, and MATF, the ROC curve and AUC were applied, in which the ROC curve is a measure of the usefulness of a test in general, while a greater area means the test is more useful. The areas under the ROC curves, called AUC, are used to compare the usefulness of the tests. Figure 6(a) depicts the results of the ROC curve and AUC scores of the dependent bug report assemblage using unigram as features, while Figure 6(b) shows the results of the ROC curve and AUC scores of the dependent bug report assemblage using unigram+CamelCase as features.

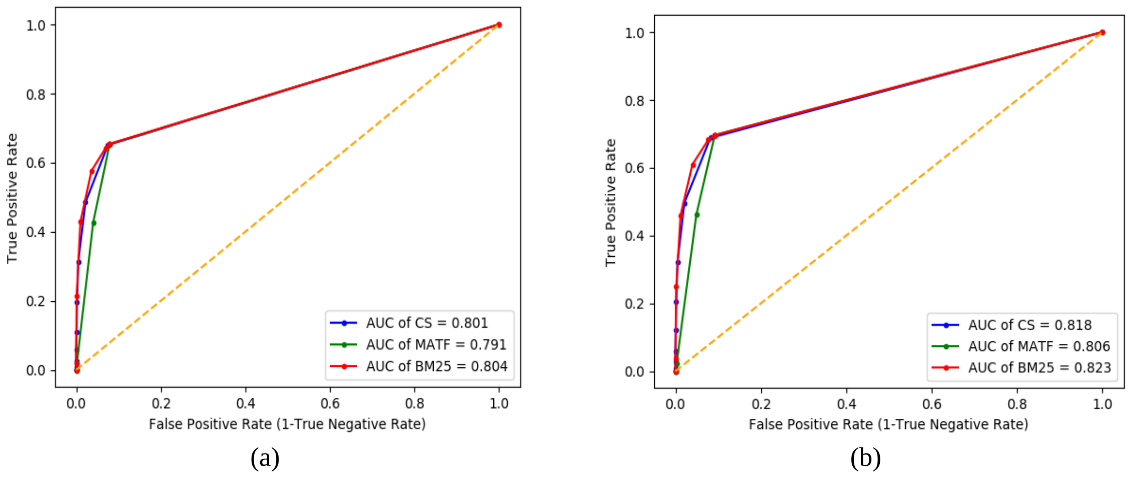


Figure 6. The Results of the ROC curves and AUC scores: (a) the models using unigram as features, (b) the models using unigram+CamelCase as features

The figures above indicate that the best CS threshold should be 0.1, while the best MATF threshold should be 0.3. However, the best BM25 threshold in Tables 2 and 3 are slightly different. The best BM25 threshold in Table 2 should be 0.4 but the best BM25 threshold in Table 3 should be 0.5. Results of the selected models in our study are summarized in Table 4.

Table 4. The summary of selected models

| Algorithms | Features | The Best Threshold | TPR | TNR | F1 |
|---|---|---|---|---|---|
| CS | unigram | 0.1 | 0.651 | 0.925 | 0.764 |
| | unigram+CamelCase | 0.1 | 0.688 | 0.918 | 0.787 |
| MATF | unigram | 0.3 | 0.654 | 0.921 | 0.765 |
| | unigram+CamelCase | 0.3 | 0.696 | 0.918 | 0.792 |
| BM25 | unigram | 0.4 | 0.654 | 0.921 | 0.765 |
| | unigram+CamelCase | 0.5 | 0.696 | 0.918 | 0.792 |

Results in Table 4 show that the MATF model with a threshold as 0.3 and the BM25 model with a threshold as 0.5 returned the best scores of TPR, TNR, and F1 at 0.696, 0.918 and 0.792, respectively. The background of MATF was similar to BM25, meaning that this technique also returned similar results of assembling dependent bug reports. Nonetheless, MATF had faster processing times than BM25, while CS had faster processing times than both BM25 and MATF.

The best models in Table 4 were chosen for comparison with the state-of-the-art method [43]. They are the CS model with threshold as 0.1, the MATF model with threshold as 0.3, and the BM25 model with threshold as 0.5. These models use unigram+CamelCase as features.

Furthermore, if considering the results in Table 4 in term of features usage are considered, it can be seen that using unigram+CamelCase as features can return better results than using only unigram as features. This is because the CamelCase can help to identify the specificity of the particular software because it may include “function names”, “variables”, “API specifications”, and so on which can be found in the software. Therefore, it was unsurprising that unigram+CamelCase as features returned more satisfactory results for assembling dependent bug reports. Then, our reason may be similar to [10], [13], [33]-[35].

### 3.3. Comparison of the proposed method and the -state-of-the-art method

As mentioned earlier, our proposed method was compared with the state-of-the-art method proposed by Rocha *et al.* [43]. They used feedback, precision, and likelihood as their evaluation metrics as also used here. Rocha *et al.* used only the summary component of bug reports, similar to our study. They also used unigram features, while the main mechanism for identifying similar bug reports was cosine similarity with a threshold as 0.1. Interestingly, Rocha *et al.* retrieved only the first five recommended bug reports. Our proposal was compared with the same environment as used by Rocha *et al.* [43]. Table 5 shows a comparison of the results.

Table 5. The results of comparisons between the proposed method and the state-of-the art method suing feedback, precision, and likelihood considering *k*=1 to *k*=5

| Metrics | *k* | Rocha *et al.* | The Proposed Method | | |
|---|---|---|---|---|---|
| | | | CS | MATF | BM25 |
| Feedback | 1 | 1.000 | 1.000 | 1.000 | 1.000 |
| | 2 | 1.000 | 1.000 | 1.000 | 1.000 |
| | 3 | 1.000 | 1.000 | 1.000 | 1.000 |
| | 4 | 1.000 | 1.000 | 1.000 | 1.000 |
| | 5 | 0.995 | 0.995 | 0.995 | 0.995 |
| | *Avg.* | 0.999 | 0.999 | 0.999 | 0.999 |
| Precision | 1 | 0.430 | 0.460 | 0.510 | 0.510 |
| | 2 | 0.413 | 0.443 | 0.487 | 0.490 |
| | 3 | 0.392 | 0.415 | 0.447 | 0.452 |
| | 4 | 0.371 | 0.389 | 0.425 | 0.436 |
| | 5 | 0.359 | 0.371 | 0.404 | 0.410 |
| | *Avg.* | 0.393 | 0.415 | 0.454 | 0.460 |
| Likelihood | 1 | 0.430 | 0.460 | 0.510 | 0.510 |
| | 2 | 0.570 | 0.615 | 0.650 | 0.650 |
| | 3 | 0.630 | 0.690 | 0.714 | 0.715 |
| | 4 | 0.680 | 0.720 | 0.760 | 0.765 |
| | 5 | 0.725 | 0.735 | 0.765 | 0.770 |
| | *Avg.* | 0.607 | 0.644 | 0.680 | 0.682 |

Results in Table 5 show that our proposed method returned better results than the state-of-the-art method proposed by Rocha *et al.* [43], with improved scores of precision and likelihood at 12% and 12.4%, 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 bug reports, and thus to derive the relevance of a bug report to a given more accurately by taking more elaborate information of terms, bug reports, and bug report collection into consideration, rather than only term appearance in the traditional similarity scheme (cosine similarity). For example, the weighting model of BM25 incorporates bug report length, average length of all bug reports in the collection, as well as the term frequency normalization effect. This technique is subsequently able to return better performance than the CS technique.

## 4. CONCLUSIONS

One task, known as the "bug dependency problem", affects the completion of software bug fixing. The bug dependency problem can be described as a situation in which an unfixed bug "x" affects bug "y". Then, bug "y" continues to occur despite being fixed if bug "x" is not yet completely fixed. Despite being mentioned by various previous studies, this problem has never been fully investigated. Therefore, this study addressed the bug dependency issue. The most relevant studies related to automatic bug dependency analysis were presented as a method for recommending similar bug reports to the report under consideration. Therefore, their proposed method was used as the state-of-the-art method for comparison with our proposal. Here, a method was presented to assign dependent bug reports into specific groups having meta-bug reports considered as the center points. The proposed method was called "threshold-based similarity analysis". To obtain the most appropriate model, two feature types namely unigram and unigram+CamelCase, and three similarity techniques namely CS, MATF, and BM25 were compared. Experimental results indicated that unigram+CamelCase returned the most appropriate results, while BM25 was better than CS and MATF. Furthermore, after evaluating all models in an experimental environment, BM25 with a threshold of 0.5 was determined as the most accurate. Therefore, this model was chosen to compare with the state-of-the-art. After comparison for feedback, precision, and likelihood, the feedback rate was the same but our proposed method improved the precision and likelihood over the state-of-the-art by 12% and 12.4% respectively. Therefore, our results can be used to encourage developers to recognize all dependent bugs in the same problem domain. It is well-known that performing this task in software development and maintenance is time-consuming and labor-intensive when performed manually. However, this study focuses on the bug reports from Mozilla Firefox for the evaluation of the proposed approach. The results of the proposed method might not be guaranteed with the inclusion of bug reports from other software projects. In the future, we plan to address this issue with two information parts of bug reports for study. They are the summary and descriptive parts. This is because the descriptive part may contain significant information that will help to improve the performance of automatic assembly of dependent bug reports.

## ACKNOWLEDGEMENTS

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

## REFERENCES

[1] K. Aggarwal, T. Rutgers, F. Timbers, A. Hindle, R. Greiner and E. Stroulia, "Detecting duplicate bug reports with software engineering domain knowledge," *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 211-220, doi: 10.1109/SANER.2015.7081831.

[2] 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*, May. 2011, pp. 207-210, doi: 10.1145/1985441.1985472.

[3] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008, pp. 52-61, doi: 10.1109/DSN.2008.4630070.

[4] R. J. Sandusky, L. Gasser, and G. Ripoche, "Bug report networks: Varieties, strategies, and impacts in af/oss development community," in *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR 2004)*, May. 2004, pp. 80-84, doi: 10.1049/ic:20040481.

[5] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *2012 16th European Conference on Software Maintenance and Reengineering*, Mar. 2012, pp. 385-390, doi: 10.1109/CSMR.2012.48.

[6] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th International Conference on Software engineering*, May. 2008, pp. 461-470, doi: 10.1145/1368088.1368151.

[7] 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, Feb. 2015, doi: 10.1007/s11432-014-5241-2.

[8] N. Bettenburg, S. Just, A. Schröter, C. Weiß, R. Premraj, and T. Zimmermann, "Quality of bug reports in Eclipse," in *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, Oct. 2007, pp. 21-25, doi: 10.1145/1328279.1328284.

[9] 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*, Nov. 2008, pp. 308-318, doi: 10.1145/1453101.1453146.

[10] 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*, vol. 8, pp. 304-318, Oct. 2008, doi: 10.1145/1463788.1463819.

[11] 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*, May. 2013, pp. 392-401, doi: 10.1109/ICSE.2013.6606585.

[12] 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*, Nov. 2014, pp. 7-12, doi: 10.1109/IWESEP.2014.8.

[13] 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*, May. 2019, pp. 69-78, doi: 10.1007/978-3-030-19861-9_7.

[14] 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*, Jul. 2017, pp. 423-430, doi: 10.1007/978-981-10-3373-5_42.

[15] A. Lamkanfi, S. Demeyer, E. Giger and B. Goethals, "Predicting the severity of a reported bug," 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), 2010, pp. 1-10, doi: 10.1109/MSR.2010.5463284.

[16] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *2011 15th European Conference on Software Maintenance and Reengineering*, Mar. 2011, pp. 249-258, doi: 10.1109/CSMR.2011.31.

[17] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *2008 IEEE International Conference on Software Maintenance*, Oct. 2008, pp. 346-355, doi: 10.1109/ICSM.2008.4658083.

[18] A. F. Otoom, D. Al-Shdaifat, M. Hammad, and E. E. Abdallah, “Severity prediction of software bugs,” in *2016 7th International Conference on Information and Communication Systems (ICICS)*, Apr. 2016, pp. 92-95, doi: 10.1109/IACS.2016.7476092.

[19] Y. Tian, C. Sun, and D. Lo, “Improved duplicate bug report identification,” in *2012 16th European Conference on Software Maintenance and Reengineering (CSMR)*, Mar. 2012, pp. 385-390, doi: 10.1109/CSMR.2012.48.

[20] C. Yang, C. Hou, W. Kao and I. Chen, "An Empirical Study on Improving Severity Prediction of Defect Reports Using Feature Selection," *2012 19th Asia-Pacific Software Engineering Conference*, 2012, pp. 240-249, doi: 10.1109/APSEC.2012.144.

[21] S. Guo, R. Chen, H. Li, T. Zhang, and Y. Liu, “Identify severity bug report with distribution imbalance by CR-SMOTE and ELM,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 2, pp. 139-175, 2019, doi: 10.1142/S0218194019500074.

[22] A. Kukkar, R. Mohana, A. Nayyar, J. Kim, B.-G. Kang, and N. Chilamkurti, “A novel deep-learning-based bug severity classification technique using convolutional neural networks and random forest with boosting,” *Sensors*, vol. 19, no. 13, p. 2964, Jul. 2019, doi: 10.3390/s19132964.

[23] W. Y. Ramay, Q. Umer, X. C. Yin, C. Zhu and I. Illahi, "Deep Neural Network-Based Severity Prediction of Bug Reports," in *IEEE Access*, vol. 7, pp. 46846-46857, 2019, doi: 10.1109/ACCESS.2019.2909746.

[24] R. P. Gopalan and A. Krishna, “Duplicate bug report detection using clustering,” in *2014 23rd Australian Software Engineering Conference*, Apr. 2014, pp. 104-109, doi: 10.1109/ASWEC.2014.31.

[25] C. Lee, D. Hu, Z. Feng and C. Yang, "Mining Temporal Information to Improve Duplication Detection on Bug Reports," *2015 IIAI 4th International Congress on Advanced Applied Informatics*, 2015, pp. 551-555, doi: 10.1109/IIAI-AAI.2015.180.

[26] P. Runeson, M. Alexandersson and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 499-510, doi: 10.1109/ICSE.2007.32.

[27] A. Hindle, A. Alipour, and E. Stroulia, “A contextual approach towards more accurate duplicate bug report detection and ranking,” *Empirical Software Engineering*, vol. 21, no. 2, pp. 368-410, Jun. 2015, doi: 10.1007/s10664-015-9387-3.

[28] 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, Mar. 2012, doi: 10.1007/s11390-012-1230-3.

[29] L. Yu, W.-T. Tsai, W. Zhao, and F. Wu, “Predicting defect priority based on neural networks,” in *International Conference on Advanced Data Mining and Applications*, 2010, pp. 356-367, doi: 10.1007/978-3-642-17313-4_35.

[30] 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*, Sep. 2010, pp. 1-10, doi: 10.1109/ICSM.2010.5609736.

[31] G. Jeong, S. Kim, and T. Zimmermann, “Improving bug triage with bug tossing graphs,” in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 111-120: ACM, Aug. 2009, doi: 10.1145/1595696.1595715.

[32] 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, doi: 10.6688/JISE.201901_35(1).0010.

[33] R. Almhana, W. Mkaouer, M. Kessentini and A. Ouni, "Recommending relevant classes for bug reports using multi-objective search," *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 286-295.

[34] 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, Sep. 2015, doi: 10.1109/TSE.2015.2479232.

[35] 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 *2012 34th International Conference on Software Engineering (ICSE)*, Jun. 2012, pp. 14-24, doi: 10.1109/ICSE.2012.6227210.

[36] M. Fischer, M. Pinzger, and H. Gall, “Populating a release history database from version control and bug tracking systems,” in *International Conference on Software Maintenance, 2003. ICSM 2003*, Sep. 2003, pp. 23-32, doi: 10.1109/ICSM.2003.1235403.

[37] M. Fischer, M. Pinzger and H. Gall, "Analyzing and relating bug report data for feature tracking," *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, 2003, pp. 90-99, doi: 10.1109/WCRE.2003.1287240.

[38] J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?,” in *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1-5: ACM, Jul. 2005, doi: 10.1145/1082983.1083147.

[39] M. Ohira, A. E. Hassan, N. Osawa, and K.-i. Matsumoto, “The impact of bug management patterns on bug fixing: A case study of Eclipse projects,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2012, pp. 264-273, doi: 10.1109/ICSM.2012.6405281.

[40] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, “How long will it take to fix this bug?,” in *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, May. 2007, pp. 1-1, doi: 10.1109/MSR.2007.13.

[41] J. Anvik, L. Hiew, and G. C. Murphy, “Coping with an open bug repository,” in *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, Oct. 2005, pp. 35-39, doi: 10.1145/1117696.1117704.

[42] Anatomy of a Bug.[Online]. Available: https://www.bugzilla.org/docs/4.4/en/html/bug_page.html (accessed May 18).

[43] 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, Apr. 2015, doi: 10.1186/s40411-015-0018-x.

[44] H. S. C. Rocha, G. A. de Oliveira, H. T. Marques-Neto, and M. T. O. 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)*, vol. 2, pp. 53-60, 2014.

[45] 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, Apr. 2016, doi: 10.1007/s10462-016-9478-6.

[46] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, Nov. 2007, pp. 34-43, doi: 10.1145/1321631.1321639.

[47] 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, Feb. 2016, doi: 10.1002/smr.1770.

[48] T. Verma, R. Renu, and D. Gaur, "Tokenization and filtering process in RapidMiner," *International Journal of Applied Information Systems*, vol. 7, no. 2, pp. 16-18, Apr. 2014, doi: 10.5120/ijais14-451139.

[49] N. Pingclasai, H. Hata, and K.-i. Matsumoto, "Classifying bug reports to bugs and other requests using topic modeling," in *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2013, vol. 2, pp. 13-18, doi: 10.1109/APSEC.2013.105.

[50] 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*, Sep. 2018, pp. 1-4, doi: 10.1145/3275219.3275239.

[51] P. Willett, "The Porter stemming algorithm: then and now," *Program: Electronic Library and Information Systems*, vol. 40 no. 3, pp. 219-223, Jul. 2016, doi: 10.1108/00330330610681295.

[52] M. Goswami, A. Babu, and B. Purkayastha, "A comparative analysis of similarity measures to find coherent documents," *International Journal of Management, Technology And Engineering*, vol. 8, no. 11, pp. 786-797, Nov. 2018, doi: 16.10089.IJMTE.2018.V8I11.17.2100.

[53] J. H. Paik, "A novel TF-IDF weighting scheme for effective ranking," in *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, Jul. 2013, pp. 343-352, doi: 10.1145/2484028.2484070.

[54] G. Amati and C. J. Van Rijsbergen, "Probabilistic models of information retrieval based on measuring the divergence from randomness," *ACM Transactions on Information Systems (TOIS)*, vol. 20, no. 4, pp. 357-389, Oct. 2002, doi: 10.1145/582415.582416.

[55] S. E. Robertson and S. Walker, "Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval," in *SIGIR'94*, pp. 232-241, 1994, doi: 10.1007/978-1-4471-2099-5_24.

[56] 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*, Nov. 2012, pp. 33-38, doi: 10.1109/TAAI.2012.20.

[57] S. B. B. Rodzman, N. K. Ismail, N. Abd Rahman, S. A. Aljunid, Z. M. Nor, and K. M. N. K. Khalif, "Expert judgment Z-Numbers as a ranking indicator for hierarchical fuzzy logic system," *IAES International Journal of Artificial Intelligence*, vol. 8, no. 3, p. 244, Sep. 2019, doi: 10.11591/ijai.v8.i3.pp244-251.

[58] R. Baeza-Yates and B. Ribeiro-Neto, Modern Information Retrieval, New York: ACM press, May. 1999, doi: 10.5555/553876.

[59] N. Japkowicz and M. Shah, Evaluating Learning Algorithms: A Classification Perspective. Cambridge University Press, Aug. 2011, doi: 10.1017/CBO9780511921803.

[60] G. Forman, "An extensive empirical study of feature selection metrics for text classification," *Journal of Machine Learning Research*, vol. 3, pp. 1289-1305, 2003.

[61] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, May. 2010, pp. 45-54, doi: 10.1145/1806799.1806811.

[62] A. M. N. Alzubaidi and E. S. Al-Shamery, "Projection pursuit Random Forest using discriminant feature analysis model for churners prediction in telecom industry," *International Journal of Electrical and Computer Engineering*, vol. 10, no. 2, pp. 1406-1421, Apr. 2020, doi: 10.11591/ijece.v10i2.pp1406-1421.

[63] W. F. W. Yaacob, S. A. M. Nasir, W. F. W. Yaacob, and N. M. Sobri, "Supervised data mining approach for predicting student performance," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 16, no. 2, pp. 1584-1592, Dec. 2019, doi: 10.11591/ijeecs.v16.i3.pp1584-1592.

[64] A. Adeleke, N. Samsudin, A. Mustapha, and S. A. Khalid, "Automating quranic verses labeling using machine learning approach," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 10, no. 1, pp. 925-931, Apr. 2018, doi: 10.11591/ijeecs.v10.i1.pp925-931.

[65] 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, Jul. 2005, doi: 10.1109/TSE.2005.72.