science, engineering and health studies

Research Article

Science, Engineering and Health Studies
https://lio1.tci-thaijo.org/index.php/sehs
ISSN (Online): 2630-0087

Bug reports identification using multiclassification method

Jantima Polpinij[1], Khanista Namee[2] and Bancha Luaphol[3]*

[1] Intellect Laboratory, Faculty of Informatics, Mahasarakham University, Mahasarakham 44150, Thailand
[2*] Faculty of Industrial Technology and Management, King Mongkut's University of Technology North Bangkok, Bangkok 10800, Thailand
[3] Department of Digital Technology, Faculty of Administrative Science, Kalasin University, Kalasin 46230, Thailand

***Corresponding author**:
Khanista Namee
khanista.N@fitm.kmutnb.ac.th

***Received**: 28 September 2022*
***Revised**: 17 November 2022*
***Accepted**: 14 December 2022*
***Published**: 27 December 2022*

***Citation**:
Polpinij, J., Namee, K., and Luaphol, B. (2022). Bug reports identification using multiclassification method. Science, Engineering and Health Studies, 16, 22020009.*

ABSTRACT

Whenever software defects (or bugs) are detected, they must be fixed immediately to allow the software to perform properly. The classification task for bug reports includes not only binary classification but also multiclassification. Therefore, multiclassification for bug reports was chosen as the challenge in this study. The proposed method aimed to classify bug reports into three classes, namely real-bug, enhancement, and task. The method began with bug report pre-processing, and then the vector of bug reports was used to develop the multiclassifier models. Eight machine learning algorithms namely multinomial naïve Bayes, logistic regression, random forest, support vector machines, *k*-nearest neighbor, extreme gradient boosting, neural networks and decision trees were compared. Finally, the classifier was chosen as the best model for the proposed method, and compared with the baseline. The Matthews correlation coefficient, area under the curve, F1 and accuracy scores of the best classifier from the proposed method showed improvement from the baseline at 4.09%, 2.71%, 1.83% and 1.69%, respectively.

Keywords: bug reports; multiclassification; supervised learning algorithms; natural language processing; Firefox; Bugzilla

1. INTRODUCTION

Today, many examples of large open-source software have been developed to promote the free distribution of information. Unfortunately, no software is free from errors and defects, also known as bugs. Reports detailing errors and information defects, called bug reports are essential for software maintenance (Ramay et al., 2019). Numerous projects utilize bug reports as guidelines for maintaining and improving software quality and efficiency. Software end-users are primary sources for gathering and reporting bugs as software defects. Bug reports can easily be collected from global software users through a bug tracking system (BTS) (Bhattacharya and Neamtiu, 2011; Jalbert and Weimer, 2008). Consequently, many BTSs such as Mantis, Bugzilla, Trace, Jira, Backlog and FogBugz have been developed to manage bug reporting and bug triaging. Many reports relating to software problems are continually generated and uploaded by software end-users to BTSs. However, some of these relate to non-bug reports (Polpinij, 2021). Therefore, all the reports must be analyzed to identify real-bug reports before utilization for software quality improvement or maintenance (Antoniol et al., 2008; Polpinij, 2021; Limsettho et al., 2014; Terdchanakul et al., 2017).

Traditionally, software experts called bug triagers manually analyze and filter non-bug reports from the bug report repository (Bhattacharya and Neamtiu, 2011; Jalbert and Weimer, 2008). By doing this, ninety days were

required to manually classify more than 7,000 bug reports (Antoniol et al., 2008; Herzig et al., 2013; Limsettho, et al., 2014; Pingclasai, et al., 2013). As a result, hand-crafted analysis takes time, escalates costs and can also introduce bias. Furthermore, after manually analyzing and classifying bug reports, 39% of the bug reports initially marked as defective never had a defect or bug (Herzig et al., 2013). This was called misclassification issue between real-bug and non-bug reports (Antoniol, et al., 2008; Herzig et al., 2013). Therefore, an automatic process of identifying and filtering out non-bug reports is required to reduce software cost and bias analysis. Several methods have been proposed for filtering out non-bug reports from the repositories before analysis (Antoniol et al., 2008; Polpinij, 2021; Limsettho et al., 2014; Pingclasai et al., 2013; Terdchanakul et al., 2017). Nowadays, studies to automatically identify real-bug reports are generally performed using binary classification. This may be because information in real-bug reports can be used to fix bug, where bug fixing is an urgent and more important task than other issues involving software maintenance. Therefore, a system to identify real-bug (defect) reports is urgently required to solve or fix software bugs. Reports that are not considered as real-bug or defect reports are often overlooked.

However, bug reports submitted to the system can also include enhancement and task bug reports. Enhancement bug reports describe new software features or user interface (UI) software performance that should be improved (Firefox, 2016; Mozilla, 2015). Therefore, information concerning enhancement bug reports can be utilized to improve software products without engineering change. Task bug reports consider refactoring, removal, replacement, enabling or disabling of functionality and any other engineering tasks (Firefox, 2016; Mozilla, 2015). These bug report types have been defined as useful for software quality improvement and maintenance but they have not yet been fully utilized and are considered as non-bug reports. When a bug is detected in software, bug fixing is an urgent and more important task than software quality improvement and maintenance. Furthermore, although binary classification to identify real-bug and non-bug reports has long been studied (Antoniol et al., 2008; Polpinij, 2021; Limsettho et al., 2014; Pingclasai et al., 2013; Terdchanakul et al., 2017, previously proposed methods did not optimally classify data for every dataset (Polpinij, 2021), and this problem remains understudied seriously. Furthermore, Limsettho et al. (2016) mentioned that software projects with insufficiently labeled data encounter difficulties in training classification models to predict bug types, and classifying bug reports into many classes (or types) has not yet been seriously studied. In general, it can be found multiclassification in other study domains of bug reports namely severity and priority analysis (Chaturvedi and Singh, 2012; Kukkar et al., 2019; Kumar and Singla, 2021; Menzies and Marcus, 2008). There are a few studies for identifying types of bug reports based on multiclassification problem domain (Kaewnoo and Senivongse, 2019). This aspect was taken up as the challenge for this study. Even after completely fixing bugs in the software, quality improvement and maintenance are still required, with the added necessity of enhancement and task reports.

In this study, a multiclassification method was used to classify bug reports into three classes: defect, enhancement, and task reports. Our method used unigram together with CamelCase words as bug report features, and compared two term weightings, namely term frequency (*tf*) and term frequency-inverse gravity moment (*tf-igm*). Eight machine learning algorithms, namely logistic regression (LR), multinomial naïve Bayes (MNB), support vector machines (SVM), decision trees (DT), *k*-nearest neighbor (k-NN), random forest (RF), extreme gradient boosting (XGBoost, XGB), and neural networks (NN), were compared in order to get the most appropriate classifier model. The best model from our proposed method was compared with the baseline method proposed by Kukkar et al. (2019) who presented a multiclass classification method to identify the severity level of bugs mentioned in reports.

2. MATERIALS AND METHODS

2.1 Dataset

The dataset was downloaded from the Bugzilla system. It was related to the open source FireFox. An example is shown in Figure 1 and Figure 2. Bug reports stored in bug repositories of Bugzilla include predefined fields, free-form text, attachments and dependencies. The predefined fields provide a variety of categorical data about the bug report. They also include product component, operating system, version, priority and severity. The free-form text includes the title of the report, a full description of the bug and additional comments, while attachments refer to non-textual additional information (e.g., a screenshot of erroneous behavior). The bug repository tracks which bugs block the resolution of other bugs. In this study used the title of the report (also known as the '*summary*' part) for the free-form text.

The dataset was downloaded on 30 September 2021, and the bug reports were uploaded to the Bugzilla system between 1 October 2018 and 30 September 2021. The dataset consisted of 21,920 reports split into three classes as defect, enhancement, and task, and contained 14,849 reports in the defect class, 4,242 reports in the enhancement class and 2,829 reports in the task class. The first stage, known as the bug report classifier modeling stage, used 2,500 bug reports per class to reduce the impact of imbalanced data resulting from the severely skewed class distribution. The remaining reports in each class were used as test sets for the second stage, called the experiment stage. After obtaining the best multiclassification model, the test sets were used to evaluate classifiers in the second stage.

2.2 The proposed method

Bug report classifiers were modeled based on multiclassification. A 10-fold cross-validation was applied when developing and validating the classification models. Figure 3 provides a general overview of the proposed method and each step of the proposed method was described below.

2.2.1 Bug report pre-processing

A bug report typically consists of three parts: title (or summary), description, and discussion. A bug report's summary is in the title, while description part details specific information of each report. The discussion contains an information detail about other end-users'

references or/and comments on that bug report. Each step of bug report pre-processing can be detailed as follows.

- Spelling corrections, to reduce language ambiguity.
- Tokenization, to separate text as "*words*".
- Stop-word removal, to remove uninformative words such as so, and, or and the from the bug report.
- Word inflection and lemmatization, to convert a unigram word into a singular form. Inflections create a variety of word forms and generate ambiguity during automatic language processing.

Unigram features together with CamelCase were used in this study. Unigram refers to single words, while CamelCase writes a word combining two words or abbreviations to yield a new word, without any punctuation and intervening spaces (e.g., browser views, UrlBar). CamelCase indicates the specificity of the software (Antoniol et al., 2008; Luaphol et al., 2021). To reduce the problem of short text expanded CamelCase features by splitting them into single words, and then both the original CamelCase words and their single words are used as features.

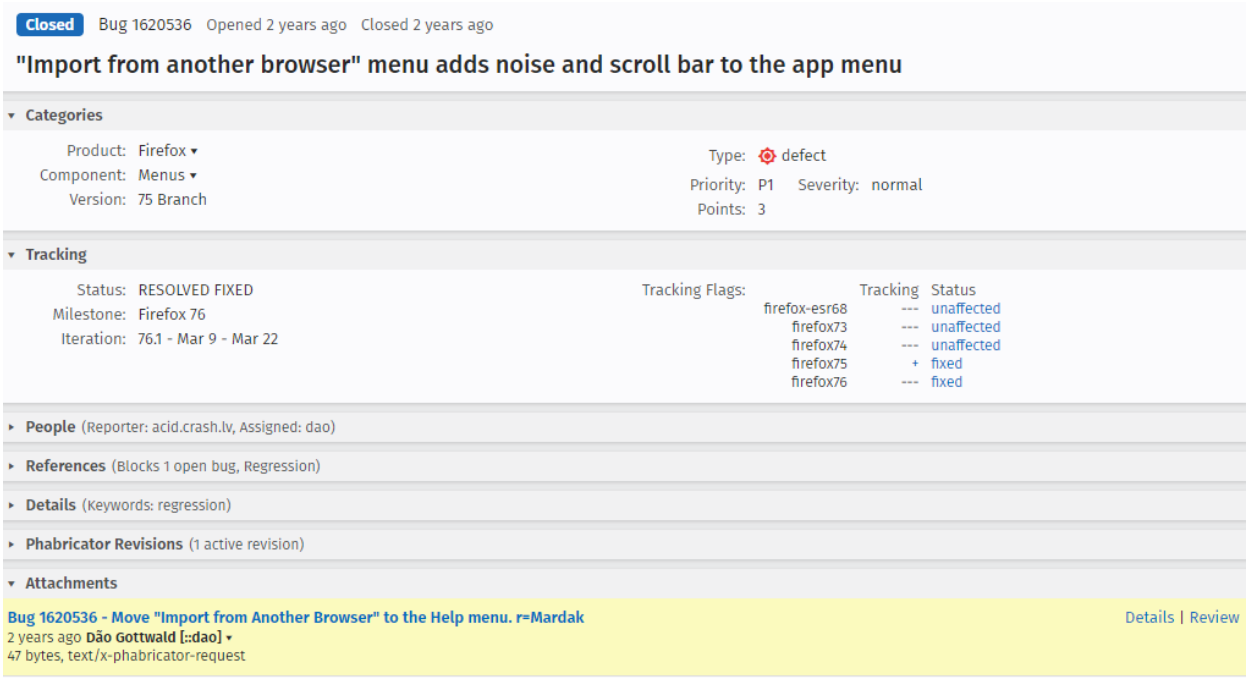

Figure 1. An example of Firefox bug report on the Bugzilla system

Figure 2. Example of Firefox bug report formatted as XML

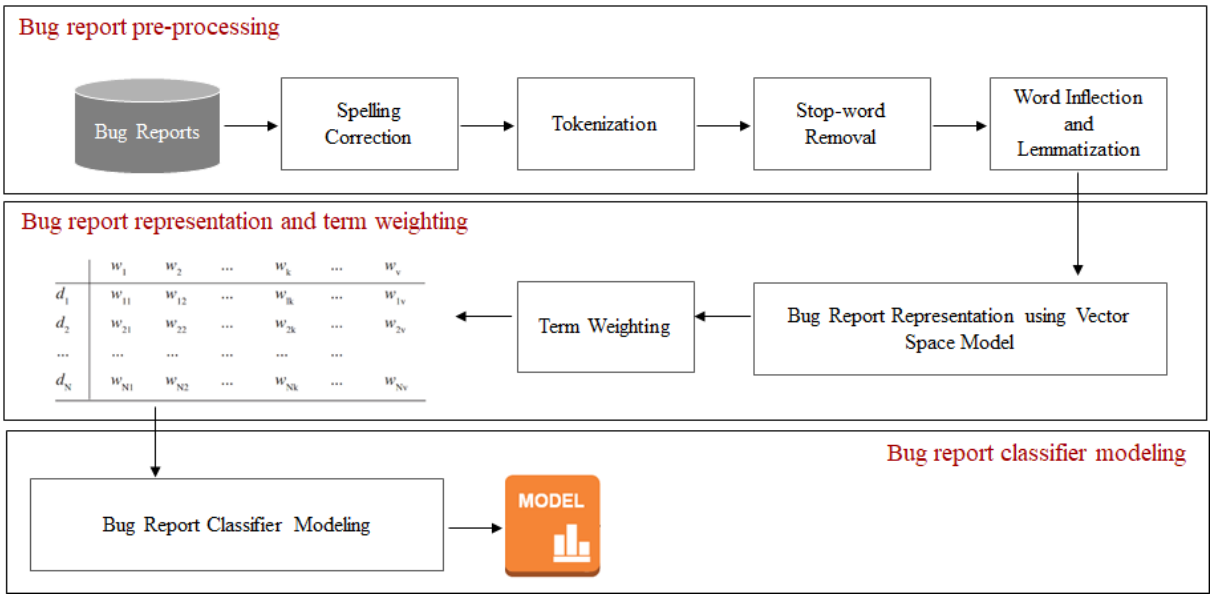

Figure 3. The proposed method overview

2.2.2 Bug report representation and term weighting

After pre-processing step, the pre-processed bug reports were represented in vector space model format (VSM), and then each bug report feature (or word) was assigned its weight using term weighting scheme. This study compared two term weighting schemes, *tf* and *tf-igm*

$$tf(t_i, d_j) = \quad tf(t_i, d_j) \tag{1}$$

Meanwhile, *tf-igm* is a supervised term weighting scheme that may be able to accurately calculate a word's distinguishing class, especially in multiclass cases as per the following equations.

$$tf - igm(t_i, d_j) = \quad tf(t_i, d_j) \times (1 + \lambda \times igm(t_i)) \tag{2}$$

$$igm(t_i) = \quad \frac{f_{i1}}{\sum_{r=1}^{M} f_{ir} \times r} \tag{3}$$

where $tf(t_i, d_j)$ indicates how many times a specific term-word t appears in bug report d. In *igm*, f_{ir} (r = 1, 2, ..., M) represents the total number of bug reports in the r-th class that contain the term t_i. These bug reports are sorted in descending order. Thus, f_{i1} represents the frequency of t_i in the class in which it appears the most frequently, while λ is an adjustable coefficient used to maintain the relative balance between the global factor *igm* and local factor *tf* in the weight of a term. In this study, the coefficient's default value (λ) used was 7.0, but this can be changed to a value between 5.0 and 9.0 (Chen et al., 2016).

2.2.3 Bug report classifier modeling and algorithm setting

After obtaining the training set of bug reports in the VSM format, this vector was used to model bug report classifiers based on multiclassification. Eight supervised machine learning algorithms were applied to create the bug report classifier models and these algorithms are described as follows:

***Logistic regression* (*LR*):** This algorithm can be used to solve classification issues by setting thresholds for the probability predicted for each class. LR classifiers use the weighted combination of the input features and pass them through a sigmoid function. Any real number can be transformed to a number between 0 and 1 using the sigmoid function.

***Multinomial naïve Bayes* (*MNB*):** MNB considers a feature vector where a given term-word represents the number of times that it appears. To develop the classifier model, MNB first calculates the fraction of documents in each class, denoted as $P(c)$, and then calculates the probability of each word for a given class, denoted as $\hat{P}(w|c)$. Finally, Bayes' rule is applied to estimate $P(c|d)$ for the test documents. To develop classifier models, MNB can be represented using the following equations.

$$P(c) = \quad \frac{N_c}{N} \tag{4}$$

where N_c represents the total number of bug reports found in each class, while N is the total number of bug reports in the training set, and

$$\hat{P}(w \mid c) = \quad \frac{count(w,c) + \alpha}{count(c) + |V| + 1} \tag{5}$$

where $count(w, c)$ represents the number of times that the term-word w appears in class c. Meanwhile, $count(c)$ refers to the total number of classes in the training set, and $|V|$ is the total number of distinct words in the training set. Since some words have zero counts, Laplace smoothing is performed with a low value of α = 0.001.

***Support vector machines* (*SVM*):** This algorithm calculates the distance between a line and the support vectors, called the margin, and identifies the points closest to the hyperplane that are termed support vectors. The goal is to maximize the margin as much as possible. A hyperplane with maximal margin is called the optimal hyperplane. The

decision boundary used to distinguish classes is as wide as possible when the maximal margin is obtained, allowing classes to be more clearly distinguished. The radial basis function (RBF) kernel was employed for the SVM method in this study because numerous investigations showed that this kernel function yielded adequate results.

***Random forest* (*RF*):** To develop each tree, RF employs bagging and feature randomization to produce an uncorrelated forest of trees that makes the prediction more accurate than using a single tree. This algorithm may be useful in preventing the problem of overfitting. It created 100 decision trees for our forest in this investigation.

***XGBoost* (*XGB*):** This algorithm is similar to RF but the XGB also utilizes a gradient boosting algorithm to enhance performance. XGB has proven to be highly efficient, adaptable, and portable. This algorithm may be useful in preventing the problem of overfitting. It generated 100 decision trees for XGB.

***k-nearest neighbor* (*k-NN*):** This algorithm is the simplest algorithm that calculates the approximate distances between vectors, and then assigns data instances that are not yet labeled to the class by ranking and considering the nearest k neighbors. In this study, closer neighbors in the same class were predicted with higher confidence. After experimenting with k = 3, k = 4 and k = 5, the trend of performance returned to upward when k = 5. Therefore, it used k = 5 for this study.

***Decision trees* (*DT*):** In this study, the C4.5 algorithm used the concept of information entropy to create decision trees from a set of training data. Features with the largest normalized information gain were chosen to determine the decision. The training data was a set of bug reports = br_1, br_2,..., br_i that were already assigned class labels. Each bug report br_i consisted of a p-dimensional vector ($w_{1,i}$, $w_{2,i}$, ..., $w_{p,i}$), where w_i represents feature values of the training set. The C4.5 algorithm selected the data feature that most successfully divided its training set into subsets enriched in one class or the other at each node of the tree. Normalized information gain was used as the splitting criterion.

***Neural network* (*NN*):** In general, NN consists of three main layers as an input layer, one or more hidden layers, and an output layer. Each neuron connects to another and has an associated weight and threshold. If any individual node's output exceeds the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed to the next layer. In this study, the Adam algorithm was used for adaptive learning rate optimization.

2.3 Measurement metrics

This study applied the metrics of accuracy (Acc), F1, the area under curve (AUC) and the Matthews correlation coefficient (MCC) to measure the performance of the proposed method. The formulae of accuracy and F1 are presented below as:

$$Acc = \frac{TP+TN}{TP+TN+FP+FN} \quad (6)$$

$$F1 = 2\times\frac{recall\times precision}{recall+precision} \quad (7)$$

$$recall = \frac{TP}{TP+FN} \quad (8)$$

$$precision = \frac{TP}{TP+FP} \quad (9)$$

where TP is the number of bug reports correctly identified as defect, TN is the number of bug reports correctly identified as other, FN is the number of bug reports incorrectly identified as defect and FP is the number of bug reports incorrectly identified as other.

AUC was used to measure the quality of classification by analyzing the area under the receiver operating characteristic (ROC) curve. The ROC curve was plotted with the true positive rate (TPR) value against the false positive rate (FPR), where the TPR is plotted on the y-axis and the FPR is plotted on the x-axis.

The MCC was used to measure the quality of the classifier model (Ramay et al., 2019). In general, the MCC is suitable for binary classes but many studies have applied MCC for multiclassification. The MCC formula can be presented as:

$$MCC = \frac{(TP*TN)-(FP*FN)}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \quad (10)$$

A value of MCC close to 1 means that all classes are well anticipated, even if some are disproportionately underrepresented (or overrepresented).

3. RESULTS AND DISCUSSION

3.1 The experimental results

To reduce the problem of short text expanded CamelCase features, they were split them into single words, and then both the original CamelCase words and their single words were used as features. Table 1 shows an example of pre-processing for a bug report title. The experimental results of the proposed method are presented in Table 2. Our testing consisted of 12,347 reports for the defect class, 1,742 reports for the enhancement class and 329 reports for the task class. Results in Table 2 showed that using the *tf-igm* term weighting scheme with every algorithm returned better results than the *tf* term weighting scheme because the *tf-igm* was a supervised term weighting (STW) scheme. The distinctive characteristic of the STW scheme is the ability to create class distinguishing power by determining the word importance in a bug report of a specific class. Simply speaking, the STW scheme indicates differences between word weights for words in different classes, although rare words occur in a few documents. This is the main reason why *tf-igm* returned better results than *tf*.

When considering all the algorithms used for modeling multi-classifiers for bug reports, the experimental results in Table 2 showed that the SVM multi-classifier with the *tf-igm* term weighting scheme returned the best results for accuracy, F1, AUC and MCC at 0.722, 0.723, 0.797 and 0.585, respectively while LR, MNB, RF, XGB, *k*-NN, DT and NN classifiers returned poorer results than the SVM classifier. This occurred because using only the title part of the bug report reduced the number of features. The SVM algorithm performed well for smaller datasets and outliers

had less impact, while NN required large training set data. Therefore, if the training set is small, the NN classifiers often produce poorer results. However, when using more data, the performance of the NN classifier improves but it still performs worse than the SVM classifier.

The LR classifiers returned lower results than SVM (Table 2). This is because the number of our features was quite small, but maybe still too many for LR. This caused the LR classifier models to over-fit on the training set, overstating the accuracy of predictions and reducing model accuracy in predicting results on the test set. Simply speaking, when the number of features exceeds the number of data points, the LR classifier model becomes underdetermined.

If considering performance of the MNB classifiers, they also returned lower results than SVM. This is because the main concept of MNB is to maintain a minimum error rate based on the assumption of class conditional independence. Unfortunately, this is not always true in reality and performances of MNB classifiers are often poor. The graphical representation of accuracy, F1, AUC, and MCC scores can be seen in Figure 4.

The RF and XGB algorithms consist of many decision trees and this may impact irrelevant features. Meanwhile, *k*-NN classifiers are very sensitive to the scale of the data and irrelevant features impact their accuracy that depends on the quality of the data. If considering for the DT classifiers, the DT algorithm is unstable, and a slight change in data often results in a significant change in the structure of the best decision tree. As a result, DT classifiers frequently yield inaccurate results.

The RF, XGB and NN multiclassifier models require longer computational time for modeling and testing than the other models. Finally, the SVM multiclassifier model with the *tf-igm* term weighting scheme was chosen as the best bug report classifier from the proposed method and compared against the baseline method proposed by Kukkar et al. (2019).

3.2 Comparing the proposed method with the baseline method

In the baseline method, all parts of bug reports (i.e., title, description and summary) were used as input data for developing classifier models. However, to ensure that the dataset used was the same in our domain, it used only the title of bug reports as the input data to compare both methods. Pre-processing of the baseline model performed tokenization, stop-word removal and stemming. During feature extraction, the content of the bug report was represented as a vector of feature (word) counts. However, the baseline method also expanded bug report content using several sets of *n*-gram data (i.e., unigram, bigram and trigram) extracted by convolutional neural networks (CNN) Methods. Lastly, to classify multiple bug report classes, RF with boosting was used. Results are shown in Table 3.

Table 1. Example results of pre-processing bug report

Processing step	Results
Original bug report	Staff cannot use the AutoComplete function. Its work is wong.
After tokenizing text	Staff / cannot / use / the / AutoComplete / function / Its / work / is / wong
After removing stop-words	Staff / cannot / use / AutoComplete / function / work / wong
After spelling correction, performing word inflection and lemmatization	Staff / cannot / use / AutoComplete / function / work / wrong
After expanding word features by using words separated from CamelCase	Staff / cannot / use / AutoComplete / Auto / Complete / function / work / wrong

Table 2. The experimental results for proposed method

Algorithm	tf				tf-igm			
	Acc	F1	AUC	MCC	Acc	F1	AUC	MCC
LR	0.709	0.709	0.781	0.563	0.712	0.713	0.784	0.568
MNB	0.687	0.688	0.765	0.531	0.711	0.712	0.783	0.567
SVM	0.709	0.709	0.782	0.564	**0.722**	**0.723**	**0.797**	**0.585**
RF	0.690	0.691	0.768	0.536	0.689	0.690	0.767	0.543
XGB	0.628	0.632	0.721	0.448	0.626	0.630	0.719	0.445
k-NN	0.601	0.586	0.701	0.409	0.661	0.662	0.745	0.492
DT	0.616	0.616	0.712	0.425	0.623	0.623	0.717	0.435
NN	0.655	0.654	0.741	0.482	0.677	0.676	0.757	0.515
MNB	0.687	0.688	0.765	0.531	0.711	0.712	0.783	0.567

Figure 4. Graphical representation of accuracy, F1, AUC, and MCC scores

Table 3. Comparing the proposed method with the baseline method

Method	Acc	F1	AUC	MCC
Proposed method	0.722	0.723	0.797	0.585
Baseline (Kukkar et al., 2019)	0.710	0.710	0.776	0.562

In Table 3, the average accuracy, F1, AUC and MCC scores of the proposed method slightly outperformed the baseline method, with improved scores of accuracy, F1, AUC and MCC at 1.69%, 1.83%, 2.71% and 4.09%, respectively. This occurred for two reasons. Firstly, our proposed method also employed CamelCase as a feature to indicate the specificity of problem domains. Using CamelCase as a feature also helped to increase class distinguishing power. Secondly, the classification method of the baseline model used RF with boosting. The RF algorithm with boosting consists of many decision trees that may impact irrelevant features, while also lacking interpretability because the ensemble of decision trees may fail to evaluate the significance of each feature. The computational time of modeling multi-classifiers and model testing was also compared. The baseline method using the RF algorithm for modeling multi-classifiers required more training time, compared to our proposed method using the SVM algorithm. The RF algorithm generated a lot of trees and made decisions on the majority of votes. This required more computational power, with increased training time.

4. CONCLUSION

From this study, 21,920 bug reports related to the Firefox opensource were used as the dataset. Our dataset contained 14,849 reports in the real-bug class, 4,242 reports in the enhancement class and 2,829 reports in the task class. The proposed method consisted of four main processing steps as pre-processing, bug report representation and term weighting, modeling multi-classifiers and evaluation. After evaluating the multi-classifier models by MCC, AUC, F1, and accuracy, the classifier model developed by SVM with RBF provided the best results. Therefore, SVM with the RBF classifier model was chosen to be compared to the baseline method. The MCC, AUC, F1, and accuracy scores of the best multiclassifier model obtained from the proposed method was marginally superior to the baseline method, with improved MCC, AUC, F1, and accuracy scores at 4.09%, 2.71%, 1.83% and 1.69%, respectively.

ACKNOWLEDGMENT

This research project was financially supported by Mahasarakham University.

REFERENCES

Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., and Guéhéneuc, Y. G. (2008). Is it a bug or an enhancement? a text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, pp. 304-318. Ontario, Canada.

Bhattacharya, P., and Neamtiu, I. (2011). Bug-fix time prediction models: can we do better? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pp. 207-210. Honolulu HI, USA.

Chaturvedi, K., and Singh, V. (2012). Determining bug severity using machine learning techniques. In *Proceedings of the 2012 CSI Sixth International Conference on Software Engineering*, pp. 1-6. Indore, India.

Chen, K., Zhang, Z., Long, J., and Zhang, H. (2016). Turning from tf-idf to tf-igm for term weighting in text classification. *Expert Systems with Applications*, 66, 245-260.

Firefox. (2016). Bug types. [Online URL: https://firefox-source-docs.mozilla.org/bug-mgmt/guides/bug-types.html] accessed on October 25, 2021.

Herzig, K., Just, S., and Zeller, A. (2013). It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 35th International Conference on Software Engineering*, pp. 392-401. San Francisco, CA, USA.

Jalbert, N., and Weimer, W. (2008). Automated duplicate detection for bug tracking systems. In *Proceedings of the 2008 IEEE International Conference on Dependable Systems and Networks with FTCS and DCC*, pp. 52-61. Anchorage, AK, USA.

Kaewnoo, P., and Senivongse, T. (2019). Identification of software problem report types using multiclass classification. In *Proceedings of the 2019 3rd International Conference on Software and e-Business*, pp. 104-109. Tokyo, Japan.

Kibriya, A. M., Frank, E., Pfahringer, B., and Holmes, G. (2004). Multinomial naive Bayes for text categorization revisited. In *Proceedings of the Australasian Joint Conference on Artificial Intelligence*, pp. 488-499. Cairns, Australia.

Kukkar, A., Mohana, R., Nayyar, A., Kim, J., Kang, B. G., and Chilamkurti, N. (2019). A novel deep-learning-based bug severity classification technique using convolutional neural networks and random forest with boosting. *Sensors*, 19(13), 2964.

Kumar, R., and Singla, S. (2021). Multiclass software bug severity classification using decision tree, naive band bagging. *Turkish Journal of Computer and Mathematics Education*, 12(2), 1859-1865.

Limsettho, N., Hata, H., Monden, A., and Matsumoto, K. (2014). Automatic unsupervised bug report categorization. In *Proceedings of the 6th International Workshop on Empirical Software Engineering in Practice*, pp. 7-12. Osaka, Japan.

Limsettho, N., Hata, H., and Monden, A. (2016). Unsupervised bug report categorization using clustering and labeling algorithm. *International Journal of Software Engineering and Knowledge Engineering*, 26(07), 1027-1053.

Luaphol, B., Polpinij, J., and Kaenampornpan, M. (2021). Mining bug report repositories to identify significant information for software bug fixing. *Applied Science and Engineering Progress*, 15(3), 1-14.

Menzies, T., and Marcus, A. (2008). Automated severity assessment of software defect reports. In *Proceedings of the 2008 IEEE International Conference on Software Maintenance*, pp. 346-355. Beijing, China.

Mozilla. (2015). *Bugzilla field descriptions*. [Online URL: https://wiki.mozilla.org/BMO/UserGuide/BugFields?fbclid=IwAR2OZBRIwIn-wereb2r6C4-KhZwYtD1lPhnW0kQZeEJyrk4P3_fqHzXcNBw#bug_type] accessed on October 25, 2021.

Pandey, N., Hudait, A., Sanyal, D. K., and Sen, A. (2017). Automated classification of issue reports from a software issue tracker. In *Proceedings of the Progress in Intelligent Computing Techniques: Theory, Practice, and Applications*, pp. 423-430. Singapore.

Pingclasai, N., Hata, H., and Matsumoto, K. (2013). Classifying bug reports to bugs and other requests using topic modeling. In *Proceedings of the 20th Asia-Pacific Software Engineering Conference*, pp. 13-18. Bangkok, Thailand.

Polpinij, J. (2021). A method of non-bug report identification from bug report repository. *Artificial Life and Robotics*, 26, 318-328.

Ramay, W. Y., Umer, Q., Yin, X. C., Zhu, C., and Illahi, I. (2019). Deep neural network-based severity prediction of bug reports. *IEEE Access*, 7, 46846-46857.

Terdchanakul, P., Hata, H., Phannachitta, P., and Matsumoto, K. (2017). Bug or not? Bug report classification using N-gram IDF. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pp. 534-538. Shanghai, China.