Severity Prediction of Software Bugs

Ahmed Fawzi Otoom, Doaa Al-Shdaifat, Maen Hammad, Emad E. Abdallah
Faculty of Prince Al-Hussein Bin Abdullah II for Information Technology, The Hashemite University
Zarqa, Jordan

***Abstract*—we target the problem of identifying the severity of a bug report. Our main aim is to develop an intelligent system that is capable of predicting the severity of a newly submitted bug report through a bug tracking system. For this purpose, we build a dataset consisting of 59 features characterizing 163 instances that belong to two classes: severe and non-severe. We combine the proposed feature set with strong classification algorithms to assist in predicting the severity of bugs. Moreover, the proposed algorithms are integrated within a boosting algorithm for an enhanced performance. Our results show that the proposed technique has proved successful with a classification performance accuracy of more than 76% with the AdaBoost algorithm and cross validation test. Moreover, boosting has been effective in enhancing the performance of its base classifiers with improvements of up to 4.9%.**

Keywords—severity predection, software bugs, machine learning , Adaboost.

I. INTRODUCTION AND REALTED WORK

Maintenance activities are an essential component of software building and accounts for high percentage of effort and cost of the software development life cycle. A key factor for reducing the effort and cost of maintenance is enhancing the bug fixing process. In open source projects, this process begins by the software users or developers submitting bug reports through a bug tracking system for potential help in fixing these bugs which leads to an improved software quality [6].

Bug tracking systems allow users to report, describe, and track bug reports [6]. Thus, it is a central medium for communication between users and developers that facilitates application maintenance. It allows users to inform developers of problems and request new features. It also enables developers of tracking unresolved bugs and requesting more information from users [8]. A popular example of bug reporting systems is Bugzilla [5] which is used by many software projects such as Mozilla, Eclipse and Gnome. These applications receive hundreds of bug reports every day [4].

There are three major research issues that are related to bug tracking systems: bug duplication, bug assignment and bug severity assessment. Bug duplication occurs when users describe problems already filed. According to [2], about 30% of Mozilla reports and around 20% of Eclipse reports are duplicates. This will be demanding for more effort of developers which can be instead employed for effective dealing of bug reports and improving quality. Extensive research have been carried out to deal with this problem (e.g. [1],[3],[6]). For example, In the work of [6], a model have been proposed based on surface features of reports, textual similarity and clustering algorithms for duplicate identification. Once a duplicate report is found, it is filed for future reference. In the work of [3], the authors argued that duplicate reports can provide additional information for actual problem diagnosis. A method is proposed to merge bug duplicates and make use of the extra information provided in the duplicates using Support Vector Machines (SVM) and Bayes Theory. This can ease the process of assigning right developers for dealing with the problem.

Recently, there has been an extensive research in the area of bug assignment (e.g. [2],[4],[7]). It is the assignment of an appropriate developer for resolving the bug. This process, if done manually, is labor-intensive, time-consuming and fault-prone [4]. To deal with these problems, machine learning techniques can be very effective in automating the process of bug assignment. For example, the authors in [2] used the history of bug reports and developers that fixed them to build a classifier that can automatically predict the most suitable developer for fixing a new bug based on the keywords in the submitted report. Recently, the authors in [4] considered the problem of bug assignment as a process of two steps. The first step assigns a bug for the first time to a developer and the second step reassigns it to another developer in case the first one is unable to resolve it. This process is based on the use of machine learning techniques and tossing graphs. An accuracy of around 85% has been reported for the prediction in the first step. Moreover, the authors reported a reduction of tossing paths of up to 86% of correct predictions.

In recent years, a large attention of researchers worldwide has been paid to the problem of severity assessment (e.g. [9-11]). Once users submit bug reports, they can identify the severity of the report. Although there are guidelines that determine how to assess the severity of software bugs, they are not always adapted by the users. Hence, developers must go manually through these bugs to identify how severe they are which is a time consuming effort considering the high numbers of reports submitted daily [11]. High severity reports represent fatal errors whereas low severity usually represent slight problems. It is important to assess the severity of bugs to prioritize them in a way that can identify how urgent it is to fix the bug from a business perspective [9]. As the number of reports is very high, it becomes urgent to build an automated tool that can predict the severity of a newly submitted report using a history of past reported reports.

In the work of [9], the authors built a system that its feature set is based on a set of specific terms' frequencies from Eclipse and Gnome reports that reflect the severity associated with bug reports. This feature set is combined with

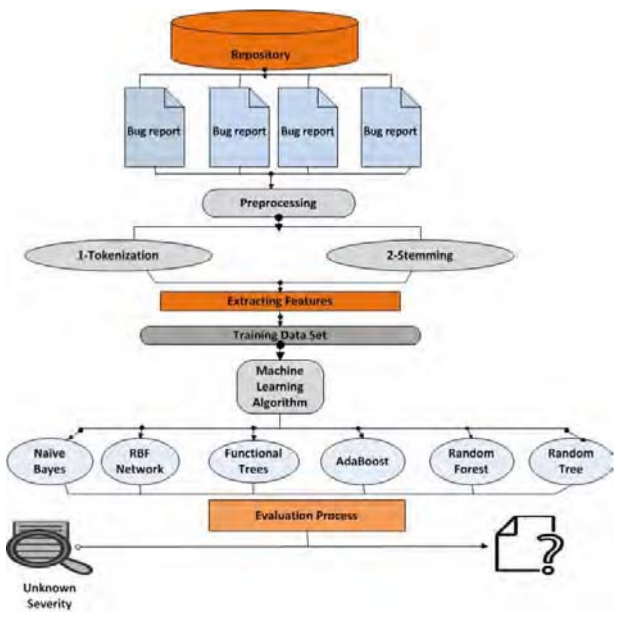

Fig. 1. The classification model.

machine learning algorithms that include SVM, Naive Bayes multinomial Naive Bayes and *K*-nearest neighbor for predicting the severity of a bug report. The highest reported results were achieved by the Naive Bayes multinomial using the *area under curve* measure. Another work that targeted this area is presented by the authors in [11]. In this work, the authors implement an information retrieval based nearest neighbor approach to predict the severity labels of bug reports. It is based on studying the similarity of different bug reports and based on this similarity, the most similar past bug reports are recovered. The approach is based mainly on textual documents comparisons.

In this work, we target the area of bug severity prediction. Our main aim is to build a strong classifier that is capable of identifying the severity of a newly submitted bug report. A strong feature set that is based on the frequencies of commonly used terms that reflect bug severity is used. These terms are based on the summary and description of bug reports which is different from related works that were using only the summary of the reports. The number of features is 59 features that characterize a mixture of two projects: Eclipse and Mozilla which makes this feature set different from previous works that were implementing a distinct feature set for each project separately. The extracted features are combined with strong learning algorithms that are used to build the classifier for prediction. Another difference from past works is the integration of the classifiers within a boosting algorithm for an enhancement in its prediction capability.

The rest of this paper is organized as follows: In section 2, we explain the feature extraction part. Section 3 gives an overview about the classification algorithms used in our experiments. In section 4, we present the experimental results. Finally, section 5 concludes the paper and suggests future work.

TABLE I. # OF INSTANCES PER CLASS

Class	# of instances
Severe	97 reports
Non-Severe	66 reports

TABLE II. PREPROCESSING OF BUG REPORTS

Original text	After tokenization	After stemming
renaming a java project wipes out linked sourcefolders.20030214 as a consequence of core making SHALLOW not default.	renaming a java project wipes out linked sourcefolders consequence of core making shallow not default	renam a java project wipe out link sourcefold consequ of core make shallow not default

II. FEATURE EXTRACTION

The first step in any classification problem is feature extraction where features are extracted for bug reports that can be effective in characterizing these reports. After extracting features, a classifier is trained using a classification algorithm and then a model is built that can be used for prediction. This process is explained in detail in Fig. 1.

For this purpose, we collected 163 bug reports from the reporting system Bugzilla and are related to Eclipse and Mozilla projects. These reports belong to two classes: severe and non-severe. The number of reports (instances) per class is explained in Table I. After collecting the bug reports, we analyzed these reports and intensively studied and examined them to identify what are the common words that usually exist in each class. Certain words like *crash*, *deadlock, slow, fault, thread, memory,* and *core* are commonly used in severe bug reports. On the other hand, other words like *extra*, *style, tick* , *typo, license, title bar, outbox, inconsistent,* and *pad* are usually presented in non-severe reports. The selection of the abovementioned features is inspired by the work of the authors in [9]. However, our feature set is higher of dimensionality compared to that in the work of [9] and it characterizes a mixture of two projects: Eclipse and Mozilla which makes it different from the work of [9] that was implementing a distinct feature set for each project separately. These collected reports are then processed in two main steps:

- Tokenization: large strings are split into tokens (words). This step is also combined with punctuation removal and the replacement of capitalized letters with small ones.
- Stemming: it reduces each term to its basic form. porter stemming algorithm is used for this reduction.

The abovementioned steps are applied to the summary and description parts of the bug reports. Table II gives an example of these two steps. Once preprocessing is completed, frequencies of 59 words that characterize the desired classes are extracted from reports. These frequencies are then normalized in regard to the length of each document. Thus, we end up with a *59-dimension* feature vector that describes 163 bug reports.

III. CLASSIFICATION

The classifiers that have been used for the classification experiments are Naive Bayes (NB), RBF Networks, Functional Trees (FT), Random Trees (RT), Random Forests (RF) and AdaBoost algorithms [12]. Naive Bayes is a probabilistic classifier based on applying Bayes' theorem with Naive independence assumptions. RBF networks implements RBF functions in Neural Networks which map complex relations between inputs and outputs. Functional Trees are classification trees that could have logistic regression functions at the inner nodes and/or leaves. Random Trees is an ensemble classifier that consists of directed graphs build with a random process. Random Forests is an ensemble classifier that consists of many decision trees and outputs the class that is the mode of the classes output by individual trees [12]. AdaBoost or adaptive boosting is a well-know method for boosting. AdaBoost produces a very accurate classification rule by combining moderately inaccurate weak classifiers.

The performance of the classifier is evaluated in terms of classification accuracy. Classification accuracy is calculated as the number of correctly classified samples divided by the total number of samples.

IV. EXPEREMINTAL RESULTS AND ANALYSIS

Experiments are conducted in order to evaluate the performance of our proposal. For this purpose, we carry out two experiments: the first experiment implements Naive Bayes, RBF Networks, Functional Trees , Random Trees, and Random Forests classification algorithms. In the second experiment, the aforementioned algorithms are integrated within an AdaBoost algorithm. In each experiment we carry out two tests: hold-out test and 10-fold cross validation test. In the hold-out test, the dataset is divided into two sets: training data and testing data. Almost 2/3 of the data is used to train the classifier and build the classification model using a certain classification algorithm. After that, the testing data are tested by the model and the predicted instances are compared with the original ones and the accuracy is calculated based on the number of correctly classified samples. On the other hand, the cross validation test divides the dataset into ten disjoint subsets where nine of them are used for training and the 10th one is used for testing. The algorithm is run for ten times and the average accuracy across all folds is calculated.

A. Experiment 1 : classification with five machine learning algorithms

In this experiment, we evaluate the performance of the feature set with five classifiers: Naive Bayes, RBF Networks, Functional Trees , Random Trees, and Random Forests. Table III. Shows the accuracy performance results of these five algorithms using the hold-out test. It is clear from this table that the proposed feature set is capable of discriminating between the two classes with the highest performance of 74.5% with Random Trees and Random Forests classifiers. This accuracy is reasonably high given that the problem in stake is not an easy problem as there are different types of ways for describing problems in the submitted bug reports which makes it a challenging problem to deal with.

TABLE III. ACCURACY RESULTS WITH HOLD-OUT TEST

Classification Algorithm	Accuracy %
Naive Bayes	67.3
RBF Networks	69.1
Functional Trees	72.7
Random Trees	**74.5**
Random Forests	**74.5**

TABLE IV. ACCURACY RESULTS WITH 10-FOLD CROSS VALIDATION

Classification Algorithm	Accuracy %
Naive Bayes	65.0
RBF Networks	65.0
Functional Trees	68.1
Random Trees	71.2
Random Forests	**73.0**

Table IV. Shows the accuracy performance results of these five algorithms using the cross validation test. It is clear from this table that the highest accuracy is achieved with the Random Forests algorithm with an accuracy result of 73.0%. This is motivating considering that the cross validation test provides a higher insight in regard to performance evaluation. In the 10-fold cross validation, the algorithm runs for ten times instead for one time by the hold out test. Hence, it is expected to have a decrease in the accuracy results and this is noted in Table. IV with slight decreases in accuracy results across the five algorithms.

B. Experiment 2 : classification with AdaBoost algorithm

In this experiment, we integrate the five abovementioned classification algorithms within an AdaBoost algorithm. These algorithms serve as *base* (*weak*) classifiers within the boosting algorithm. Boosting is a general method for producing a very accurate classification rule by combining inaccurate or weak classifiers. AdaBosst or adaptive boosting is applied widely within the pattern recognition community. AdaBoost by theory must be effective for improving the base classifier performance unless there some issues like overfitting or having a very weak base classifier.

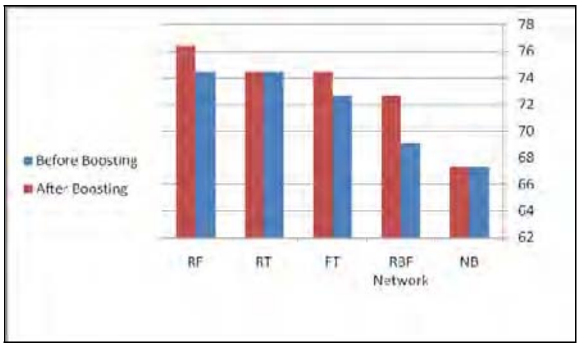

Fig. 2. Boosting results with hold-out test.

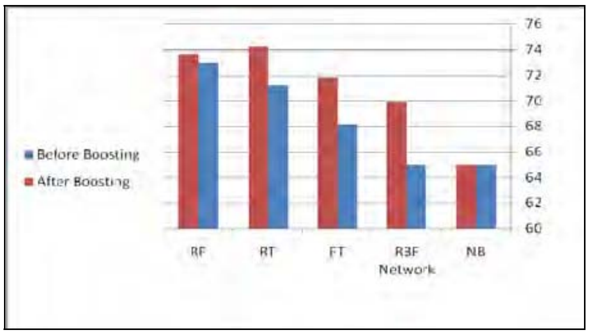

Fig. 3. Boosting results with cross validation test.

Fig. 2 and fig. 3 illustrate the AdaBoost effect on the base classifiers using both validation tests. It is clear from these two figures that AdaBoost has improved the performance of the majority of base classifiers.

In fig 2, with the hold-out test, the highest achieved results are with the Random Forest base classifier with an accuracy of 76.4%, with an improvement of 1.9% to the best performing classifier from Table III. Generally, there has been improvements ranging from to 0.0% to 3.6%. On the other hand, in fig 3, with the cross validation test, the highest achieved results are with the Random Trees base classifier with an accuracy of 74.2%, with an improvement of 1.2% to the best performing classifier in Table IV. Generally, there has been improvements ranging from to 0.0% to 4.9%. Generally, AdaBoost has been effective in enhancing the performance of the base classifiers which conforms with its basic theory.

V. Conclusion

Software bugs severity assessment is essential for developers to prioritize bug reports. If done manually, it is labor-intensive and time-consuming. Hence, it becomes urgent to automate its process. A number of research works have been proposed for this automation mainly based on machine learning techniques and text matching. In this work, we targeted this problem on both components: feature extraction and classification. For the feature extraction part, we thoroughly studied high number of bug reports to try to have a connection between the existence of certain terms and their relation to the labels of severity assigned to them ending up with 59 important text words that are discriminative between the two classes: severe and non-severe. The frequencies of these terms are extracted from 163 bug reports of Eclipse and Mozilla projects from the popular database Bugzilla. These features are then fed for five classification algorithms and the results were promising achieving an accuracy of 74.5% and 73%, with hold-out and cross validation test, respectively. For an enhanced performance, we integrated the classification algorithms within a popular boosting algorithm, named AdaBoost. Boosting have proved successful in enhancing the base algorithms with improvements of up to 4.9% with cross validation test. By using AdaBoost, the classification accuracy reached 76.4% with hold-out test and 74.2% with cross validation test.

In the future, we plan to increase the number of bug reports used for training the classifiers. Moreover, we plan to experiment with feature reduction (both linear and non linear) and feature selection techniques for an enhancement in the classification performance.

References

[1] 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, 2005, pp. 35-39.

[2] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?," in Proceedings of the 28th international conference on Software engineering, 2006, pp. 361-370.

[3] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful… really?," in Software maintenance, 2008. ICSM 2008. IEEE international conference on, 2008, pp. 337-345.

[4] P. Bhattacharya, I. Neamtiu, and C. R. Shelton, "Automated, highly-accurate, bug assignment using machine learning and tossing graphs," Journal of Systems and Software, vol. 85, pp. 2275-2292, 2012.

[5] Bugzilla. Available from : https://www.bugzilla.org/

[6] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on, 2008, pp. 52-61.

[7] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2009, pp. 111-120.

[8] S. Just, R. Premraj, and T. Zimmermann, "Towards the next generation of bug tracking systems," in Visual languages and Human-Centric computing, 2008. VL/HCC 2008. IEEE symposium on, 2008, pp. 82-85.

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

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

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

[12] I. Witten and E. Frank, "Data Mining: Practical machine learning tools with Java implementations, ed," M. Kaufmann, San Francisco, 2000.