Predicting Bugs' Components via Mining Bug Reports

Deqing Wang, Hui Zhang, Rui Liu, Mengxiang Lin, and Wenjun Wu
State Key Laboratory of Software Development Environment, Beihang University,
No.37 Xueyuan Road, Haidian District, Beijing, 100191, P.R.China
Email: {dqwang, hzhang, liurui, mxlin,wwj}@nlsde.buaa.edu.cn

***Abstract*—The number of bug reports in complex software increases dramatically. Since bugs are still triaged manually, bug triage or assignment is a labor-intensive and time-consuming task. Without knowledge about the structure of the software, testers often specify the component of a new bug incorrectly. Meanwhile, it is difficult for triagers to determine the component of the bug only by its description. For instance, we dig out the components of 28,829 bugs from the Eclipse bug project, which have been specified incorrectly and modified at least once, and indicated that these bugs have to be reassigned and the process of bug fixing has to be delayed. The average time of fixing incorrectly specified bugs is longer than that of correctly specified ones. In order to solve the problem automatically, we use historical fixed bug reports as training corpus and build classifiers based on support vector machines and Naïve Bayes to predict the component of a new bug. The best predicting precision reaches up to 81.21% on our validation corpus of Eclipse project.**

***Index Terms*—bug reports, bug triage, text classification, predictive model**

I. INTRODUCTION

The number of bug reports (BRs) in complex software increases dramatically. According to our statistics, the total number of bugs in Eclipse bug project [2] has reached up to 320,000 until July 31, 2010. In 2010, there are about 21 BRs per day submitted to Eclipse, on average. When a new release is forthcoming, the average number of BRs is up to 30. Complex software projects have to rely on a bug tracking system (BTS) for the management of BRs during development and maintenance.

J. Anvik, L. Hiew, and G. C. Murphy [1] illustrated the life cycle of a bug report for the Eclipse bug project, as shown in Fig. 1. Because bugs are triaged manually, the highlighted **ASSIGNED** step (bug triage or assignment) is a labor-intensive and time-consuming task, especially for complex software projects. In the reference [3], G. Jeong, S. Kim, and T. Zimmermann found that a bug took 16.7 days to have the first action and then 23.6 days to be assigned by analyzing the first 145,000 BRs from Eclipse and 300,000 BRs from Mozilla [4]. Meanwhile, bug triage is often an error-prone task, because triagers[1] have to assign each bug to one of the several thousand developers only by reading bug description and relying on their former experiences. In order to help triagers reduce search ranges and locate bugs quickly, some intelligent assistant tools should be developed.

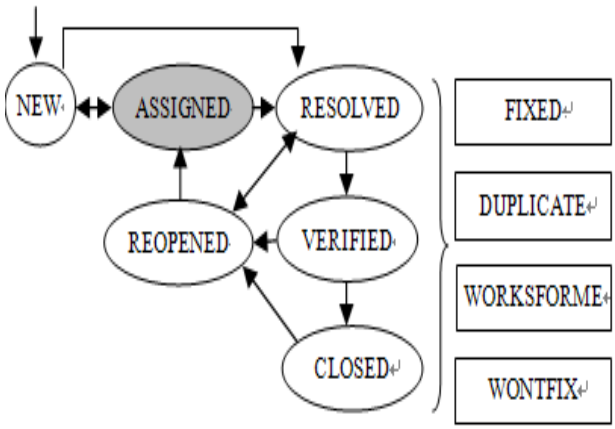

Figure 1. The life-cycle of an Eclipse bug report.

Before submitting a new bug to the BTS, end users are asked to specify pre-defined fields, such as the component of the bug, affected version and so on. Here ***the component of a bug means where the bug can be localized in source code***. The information of component can enable triagers to locate bugs quickly and find the most likely team that is responsible for the program component where the bug most likely can be identified and corrected. Therefore, specifying the component of a bug correctly is of good assistance for bug triagers.

However, without knowledge about the structure of the software, end users can only write the description of a bug and have no idea which component the bug actually relates to. He may either randomly pick up a component from the list or select nothing. We found such a phenomenon often occurs in complex software projects. For instance, we located the components of 28,829 bugs distributed in 30 components in Eclipse bug project, which had been specified incorrectly during submissions and processing. The wrong specification resulted in bug

[1] Triagers, people who help filter the reports down to those representing real issues and who help assign reports to developers.

doi:10.4304/jsw.7.5.1149-1154

reassignment, increased the burdens of triagers and delayed the process of bug fixing.

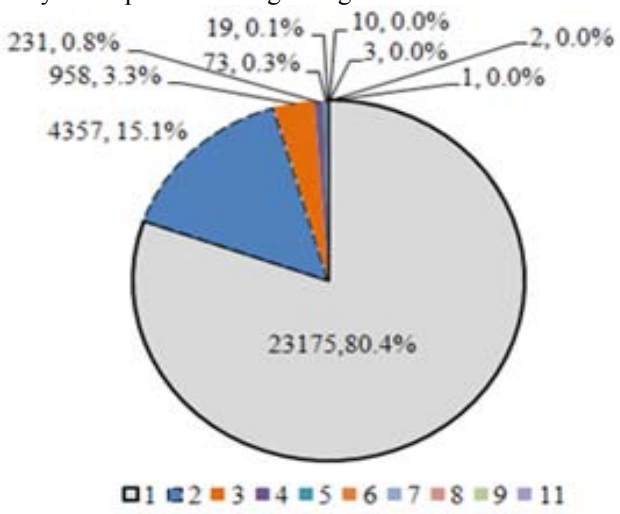

Figure 2. Frequencies of component modification. If a component is modified once, its frequency is one.

Among 28,829 bugs, we counted the number of times that the component was modified and discovered some interesting results. As shown in Fig. 2, the components of 23,175 bugs (accounting for 80.4% of the total) were modified once, the components of 4357 bugs (accounting for 15.1% of the total) were modified twice; And the rest 4.5% were updated more than twice. To our surprise, the component of bug-82839 was modified as many as 11 times. Details about the modifications can be found at https://bugs.eclipse.org/bugs/show_activity.cgi?id=82839 . It is obvious that even triagers and developers could not confirm the component of a bug, which seriously delays the process of bug fixing. After all, the information of bug description is not enough to determine the component easily. In the preceding example, it took developers about 5 months (from Jan 14, 2005 to Jun 22, 2005) to fix the bug. Statistics has shown significant differences between the average fixing time of correctly specified bugs and that of incorrectly specified bugs. The average time for fixing a correctly-specified bug is 136.2 days, while the time for a incorrectly-specified bug is 190.5 days.

Our motivation is "**Can we help end users specify the component of a new bug automatically or help triagers to predict its component correctly?**" This paper focuses on applying text classification techniques to predict bugs' component based on historical fixed BRs and implementing an intelligent software toolkit for triagers to assign bugs to corresponding teams. Triagers can ask team leaders to confirm whether the assignment is accurate. After all, team leaders know more about the component than triagers do. The tool enables end users to report bugs easily and enables triagers to speed up bug assignment.

This paper makes the following two contributions:

- We evaluate the impact of incorrectly specified bugs. Through the detailed statistical results on Eclipse project, we find the components of bugs are often specified incorrectly. It affects triagers and developers to assign and locate bugs correctly and then delays the processes of bug fixing.
- Predict bugs' components via mining historical BRs. We use all fixed BRs to build classifiers based on support vector machines and Naïve Bayes, and then apply the classifiers to predict the components of bugs. The approach can assist end users to specify the component of a new incoming bug correctly and enable triagers to speed up bug assignment and reduce bug reassignment. In addition, the prediction accuracy on validation corpus for Eclipse reaches up to 81.21%.

II. Related Work

In the recent years, many machine learning and data mining techniques have been applied to software engineering tasks, especially bug assignment. Here we briefly describe recent research papers.

D. Cubranic and G. C. Murphy [5] applied text categorization to predict bug assignment based on the descriptions of bugs. Their prototype, using supervised Bayesian learning, correctly predicted 30% of the report assignments to developers on a collection of 15,859 BRs from Eclipse project. The precision is not good enough. Then J. Anvik, L. Hiew, and G. C. Murphy [1] used support vector machines (SVM) [6] as a classifier, reaching precision levels of 57% and 64% on Eclipse and Firefox respectively. G. Jeong, S. Kim, and T. Zimmermann [3] introduced a graph model based on Markov chains to solve bug reassignment, which increased automatic bug assignment accuracy by up to 23% in Mozilla and Eclipse. M. Gegick, P. Rotella and T. Xie [7] applied text mining to identify security bug reports (SBRs) from manually mislabeled non-security bug reports (NSBRs). The model successfully classified 78% of the SBRs from a sample of Cisco BRs. K. Yi, H. Choi, J. Kim, and Y. Kim [15] applied a variety of classifiers to categorize alarms found by static analyzers and found random forest and boosting worked better in their study. P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy [8] built a statistical model to predict the probability that a new bug is to be fixed based on bug report edits and relationships between people involved in handling bugs. They got a precision of 68% and recall of 64% when predicting Windows 7 bugs. These tools have not been applied to practical projects because of their low prediction accuracy. Now manual bug triage is still the most common approach, but we can provide triagers with better assistance tools to help them avoid many wrong assignments.

III. Approach

Our approach builds a supervised classifier trained on historical BRs to predict the component of a new bug. It consists of training process and predicting process, as shown in Fig.3. In the training process, we extract the summary, description and comments of a bug as its content. Then we convert the content into bag of words and calculate their TF-IDF (Term Frequency-Inverse

Document Frequency) weighting values, including stop words filtering, word stemming and feature selection (χ^2 statistics). Last, we apply three classifiers to train our models. In the predicting process, we just extract the summary and description of a new bug and represent it as a feature vector, and then we predict the component of the bug using our classifier models. After the prediction is done, the predicted component label is forwarded to reporters or triagers. Reporters can use the predicted component label as the option of the component field when they submit a bug. Moreover, when the component field is empty or wrong, triagers can utilize the predicted component as the most likely one and ask corresponding team leader to confirm. It saves reporters and triagers time for bug submission and assignment, and it reduces bug reassignments.

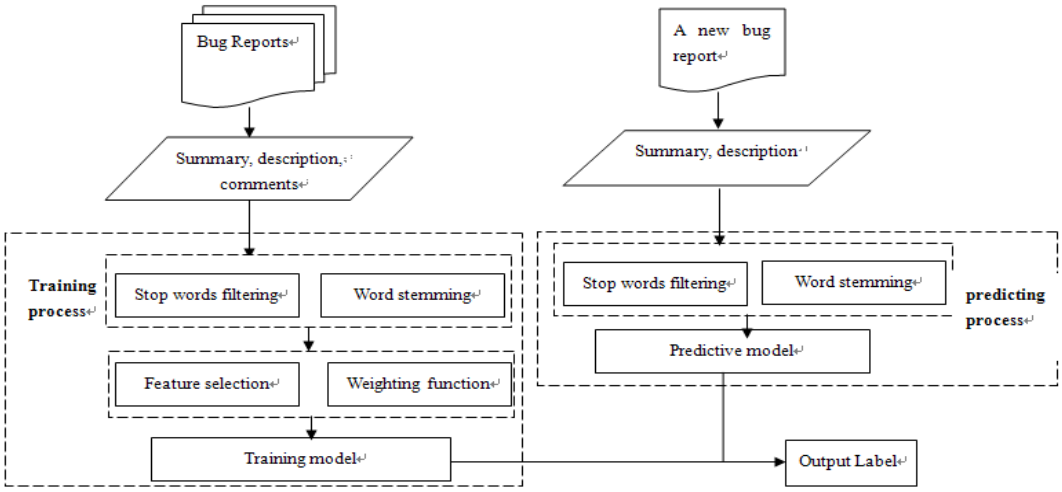

Figure.3. The dataflow of our approach, consists of training process and predicting process.

A. *Preprocessing*

Although a bug report contains a substantial amount of information, only part of the report is useful for the construction of classifiers. We extract id, status, resolution, component, summary, description and comments from each bug report. The value of component denotes the class label of one bug. The text of summary, description and comments represents the content of one bug. In order to characterize a bug report, each bug report is converted into a feature vector. Stop words filtering and word stemming are also introduced to perfect the feature vector.

Stop words are very common words that are useless in text classification. For example, usually articles, conjunction and prepositions are stop words. In our corpus, we list 322 stop words and filter them automatically during the preprocessing. After stop words filtering, many words have the same word stem, such as "reporting" and "reported". We use Snowball [11] to accomplish word stemming to reduce the dimension of the feature vector.

B. *TF-IDF Weighting Function and Feature Selection*

After the preprocess of BRs, each bug report is converted into a set of keywords. The original dimension of the feature vector reaches up to 400,000. We apply feature selection to remove non-informative terms according to corpus statistics. Yiming Yang and Jan O. Pedersen [10] presented a comparative study of five feature selection methods in statistical learning of text categorization. They found χ^2 statistic (Chi square statistics, CHI) is most effective. We applied CHI to select features from our BRs corpus. The CHI value of between a term *t* and a category *c* is defined to be

$$\chi^2(t,c) = \frac{\mathrm{N}\times(\mathrm{AD}-\mathrm{CB})^2}{(\mathrm{A}+\mathrm{C})\times(\mathrm{B}+\mathrm{D})\times(\mathrm{A}+\mathrm{B})\times(\mathrm{C}+\mathrm{D})} \quad (1)$$

where A is the number of times *t* and *c* co-occur, B is the number of times *t* occurs without *c*, C is the number of times *c* occurs without *t*, D is the number of times neither *c* nor *t* occurs, and N is the total number of documents. We compute for each category the χ^2 statistic between each unique term in the training corpus and that category, and then sum the category-special scores of each term into one score.

$$\chi^2_{\mathrm{avg}}(t) = \sum_{\mathrm{i}=1}^{m} P(c_i)\chi^2(t,c_i) \quad (2)$$

where $P(c_i)$ equals the number of documents in class c_i divided by the total number of documents, *m* is the total number of categories. Sorting terms by the value of $\chi^2_{\mathrm{avg}}(\mathrm{t})$ in descending order, we select top *K* terms as features of the BRs corpus.

The function of term weight is used to evaluate the weight of term *t* in document *d* after feature selection. A number of term weighting functions and their variants have been proposed in text classification. TF-IDF weighting function is a statistical measure for evaluating how important a word is to a document in a corpus. In our approach, we use the best term weighting formula (*tfc*) [9] to calculate the weight of one term. The formula is defined to be

$$w_{ij} = \frac{tfidf(t_i,d_j)}{\sqrt{\sum_{k=1}^{|V|}[tfidf(t_k,d_j)]^2}} \quad (3)$$

where w_{ij} is the weight value of *i*th term in the *j*th document, $|V|$ denotes the total number of unique terms contained in the training document d_j. The definition of $tfidf(t_i, d_j)$ is given by

$$tfidf(t_i, d_j) = tf(t_i, d_j) \times log\frac{|Tr|}{\#Tr(t_i)} \quad (4)$$

where $|Tr|$ and $\#\mathrm{Tr}(t_i)$ denote the total number of documents, the number of documents containing term t_i in training set *Tr*, respectively. $\mathrm{tf}(t_i, d_j)$ is raw term frequency.

C. Classification Model

We treat the problem of predicting bugs' components as an instance of text classification. More specifically, it is a multi-class, single-label classification problem: each component corresponds to a single category, and each bug report is predicted into only one component. A number of supervised learning techniques have been applied for text classification. For instance, Naïve Bayes, K-nearest neighbor, regression model, and support vector machines. (Details about their performances can be found from Yang [12].) In our approach, we use support vector machines and Naïve Bayes as our classification algorithms. Three open source classification tools including LIBSVM [13], LIBLINEAR [14] and Bow [16] are chosen for implementations of SVM and Naïve Bayes classifiers because of their outstanding performances and efficiencies. LIBSVM is an integrated software library for support vector classification, regression and distribution estimation. LIBLINEAR is a linear classifier for data with millions of instances and features. Bow is a toolkit for statistical text classification.

We use radial basis function as kernel function and first run cross-validation to select the optimal parameters C and γ (C is the penalty parameter and γ is the kernel parameter). Then we apply the optimal parameters to train our classification model. To LIBLINEAR, we use default parameters to train the model. We use word stemming and stop words filtering to preprocessing of BRs and use Naïve Bayes as the classifier when using Bow. Last, we run our predictive models on test set and compare accuracies of three classifiers.

IV. Results and Discussion

We applied our classifier on a collection of 90,768 BRs collected from Eclipse project and tested its accuracy in predicting component labels. Then we discussed the applications of our approach.

A. Data Sets and Measure

We selected 90,768 BRs as our data set from 320,000 BRs in the BTS of Eclipse project because they have been fixed or closed, which means components of these bugs have been confirmed. The status of each bug is "resolved", "closed", or "verified" and its resolution is "fixed". As shown in Table I, these BRs are scattered in 30 components. We find the numbers of BRs in UI component (25), SWT component (20), DEBUG component (8), CORE component (6), TEXT component (22) and TEAM component (21) are 34148, 9764, 8822, 7938, 6232 and 4308, respectively, accounting for 37.6%, 10.8%, 9.7%, 8.8%, 6.9% and 4.8% of the total, respectively. Since Eclipse starts as a Java IDE and provides developers a platform to debug and run java program, many bugs happened in UI, SWT, DEBUG and TEXT components. We observe the data set is unbalanced by analyzing the data in Table I. There are eight components that have not more than 30 bugs. In our experiments, the 90,768 BRs generate two data sets (DS_I and DS_{II}). In DS_I, we extract summary, description, and comments to form the content of each bug, whereas we only extract summary and description, and ignore the comments in DS_{II}. The DS_{II} is used to validate the effect of comments of bugs. We randomly select 20% instances from each category as testing set and the rest as training set in each data set. Besides, 28,829 incorrectly specified bugs form a validation corpus, and we generate two validation sets VS_I and VS_{II}, In VS_{II}, comments of each bug is excluded. Through the corpus, we can estimate the precision and the time saved on fixing these bugs.

We evaluate the performance of our approach using the measure of precision , which measures how often the approach makes an appropriate prediction of component label for a bug. The formula is defined to be

$$Precision = \frac{\#\ of\ appoppriate\ prediction}{\#\ of\ prediction} \times 100\% \quad (5)$$

TABLE I.
THE NUMBER AND PERCENTAGE OF BUGS IN EACH COMPONENT, THE NAME OF COMPONENT IS SUBSTITUTED BY NUMERIC CHARACTER.

Label	Num	%	Label	Num	%	Label	Num	%
1	2407	2.65%	11	454	0.50%	21	4308	4.75%
2	830	0.91%	12	2	0.00%	22	6232	6.87%
3	242	0.27%	13	239	0.26%	23	1	0.00%
4	1103	1.22%	14	19	0.02%	24	25	0.03%
5	972	1.07%	15	1615	1.78%	25	34148	37.62%
6	7938	8.75%	16	2768	3.05%	26	1808	1.99%
7	1663	1.83%	17	1199	1.32%	27	2522	2.78%
8	8822	9.72%	18	7	0.01%	28	30	0.03%
9	830	0.91%	19	645	0.71%	29	27	0.03%
10	19	0.02%	20	9764	10.76%	30	129	0.14%

B. Results and Discussions

In our experiments, we constructed three groups of experiments to verify our approach on our four data sets. Through the three groups of experiments, we can find better dimension of features, compare the precision with bug comments or without comments, and detect the better classifier on the BRs corpus.

B.1 Feature Selection

After preprocessing of our BRs corpus, the dimension of the feature vector reaches up to 400,000. If we directly applied LIBSVM to the raw feature vector, we would spend about 30 days to obtain the optimal parameters C and γ. The training time is unacceptable. Therefore, we applied CHI for feature selection. How many features can represent the corpus better? We obtained the number of features from a heuristic experiment on DS_I. As shown in Fig. 4, when the dimension of the feature vector is in [10k, 150k], the precision of LIBLINEAR is higher than that of LIBSVM and the both precision curves increase flatly. When the dimension is greater than 150k, the precision of LIBSVM is better than that of LIBLINEAR and the both precision curves increase abruptly. In order to make a reasonable tradeoff between the precision and training cost, we have take a balanced approach in the feature selection: When users want a fast classifier, we use LIBLINEAR as a classifier and the dimension is set as 10k. When users want a precise classifier, we use LIBSVM as a classifier and the dimension is set as 400k.

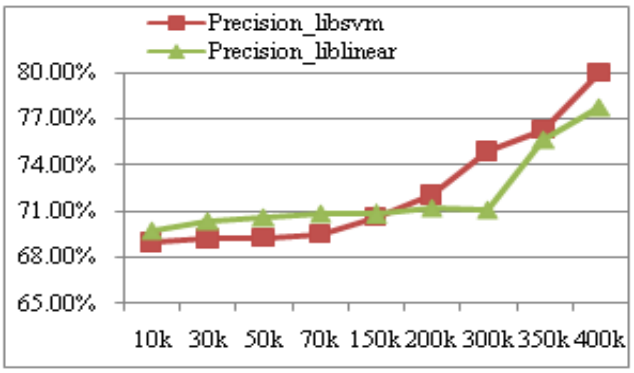

Figure 4. The precisions of different dimensions of feature vector using CHI.

B.2 With or Without Comments

Comments of BRs are very important for triagers or developers to assign or fix bugs. We think comments should play an important role in prediction of bugs' components. In order to measure the contribution of comments to the prediction, we compare the precision of prediction with comments to that without comments in six experiments on DS_I (with comments) and DS_{II} (without comments). As shown in table II, the precisions of LIBSVM, LIBLINEAR and Naïve Bayes improve by 7.75%, 5.8% and 3.89%, respectively. Moreover, the best precision of LIBSVM on DS_I reaches up to 80%. Therefore, we conclude comments are necessary in prediction of bugs' components through experiments. In our designed classifiers, comments of bugs are included.

TABLE II.
THE PRECISIONS OF CLASSIFIERS ON DIFFERENT DATA SETS

	DS_I	DS_{II}	Improvement	VS_I	VS_{II}
LIBSVM	80.00%	72.15%	7.75%	84.12%	**81.21%**
LIBLINEAR	77.42%	71.62%	5.80%	80.81%	73.92%
Naïve Bayes	66.82%	62.93%	3.89%	59.31%	52.76%

B.3 Prediction and Discussion

Our approach is to predict the component of an incoming bug via mining historical BRs. In order to verify our approach, we use all fixed bugs as training instances, denoted by DS_{all}. Then we run our predictive models on the validation corpus, i.e., VS_I and VS_{II}. The difference between the two data sets is whether it includes comments of bugs. VS_{II} is more realistic situation, because a new incoming bug report only contains the title and description. Therefore, we select VS_{II} as corpus when we estimate the time saved on repairing bugs. The results are shown in the right two columns of table II. The precisions using LIBSVM (C = 512 and γ =0.5) on VS_I and VS_{II} reach up to 84.12% and 81.21%, respectively. The precisions using LIBLINEAR on VS_I and VS_{II} reach 80.81% and 73.92%, respectively. However, the precisions using Naïve Bayes just reach 59.31% and 52.76%. We think the unbalance of corpus affects the performance of Naïve Bayes. Because the prior probability $p(C_k)$ is estimated by the number of documents in C_k divided by the number of total documents, the posterior probability $p(C_k|x)$ will trend to classes that have more documents. Through the comparative experiments of different classifiers, we find LIBSVM classifier performs better than LIBLINEAR and Naïve Bayes on our BRs corpus.

The best precision on the realistic data set reaches 81.21%, which shows that the LIBSVM classifier can assist testers and triagers to accomplish bug submission and assignment in an acceptable accuracy. In our validation corpus containing 28,829 incorrectly specified bugs, our classifier can predict 81.21% of the total accurately just using descriptions of bugs. Meanwhile, our classifiers can help testers determine the component of a new incoming bug and enable triagers to assign the bug to corresponding team.

In sum, predicting components of bugs via mining BRs has the following benefits: 1). It can make it easier for reporters to specify the component of a new bug. They just need to write the description of a bug and the classifier can automatically fill in the corresponding component. 2).It can assist triagers and developers to assign and locate bugs quickly. If the components of bugs are empty or wrong, triagers can consult the classifier to label bugs and then assign bugs to the most likely team. 3). It can save triagers and developers time spent on fixing bugs.

V. CONCLUSIONS

Incorrectly specified bugs often result in bug reassignment and delay the process of bug fixing. This paper investigates the impact of incorrectly specified bugs and attempts to address the problem using data mining techniques. We present a predictive model based on historical fixed bug reports and three classifiers to predict the component label of a new incoming bug. In our experiments on Eclipse bug corpus, the accuracy of our model based on LIBSVM classifier reaches up to 81.21%. It means our model can specify most bugs' components correctly according to historical data. In the future, we will develop a user interface and provide our software tool to Eclipse bug tracking system.

ACKNOWLEDGMENT

We thank Matt Ward (the webmaster of Eclipse bug tracking system) for providing us the bug reports corpus. He and Wayne Beaton are so nice to answer our questions about bug triage. The research is supported by the 863 High-Tech Program under Grant No. 2007AA010403.

REFERENCES

[1] J. Anvik, L. Hiew, G. C. Murphy, "Who should fix this bug?" In Proceedings of *the 28th international conference on Software engineering (ICSE '06)*, 2006.
[2] Eclipse bug project, http://bugs.eclipse.org/bugs.
[3] G. Jeong, S. Kim, 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 (ESEC-FSE'09)*, 2009.
[4] Mozilla, https://bugzilla.mozilla.org.
[5] D. Cubranic, G. C. Murphy, "Automatic bug triage using text classification", In Proceedings of *Software Engineering and Knowledge Engineering*, 2004, pp. 92–97.
[6] S. R. Gunn, "Support Vector Machines for classification and regression". Technical report, University of Southampton, Faculty of Engineering, Science and Mathematics; School of Electronics and Computer Science, 1998.
[7] Michael Gegick, Pete Rotella, Tao Xie, "Identifying Security Bug Reports via Text Mining: An Industrial Case Study", In Proceedings of *the 7th Working Conference on Mining Software Repositories (MSR 2010)*, Cape Town, South Africa, 2010.
[8] P. J. Guo, T. Zimmermann, N. Nagappan, B. Murphy, "Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows", In Proceedings of *the 32th International Conference on Software Engineering (ICSE'10)*, 2010.
[9] G. Salton, C. Buckley, "Term-weighting approaches in automatic text retrieval", *Information Processing & Management*, 24 (5), pp. 513–523.
[10] Yiming Yang, Jan O. Pedersen, "A Comparative Study on Feature Selection in Text Categorization", In Proceedings of *the Fourteenth International Conference on Machine Learning*, 1997, pp. 412 – 420.
[11] Martin F. Porter, "Snowball: A language for stemming algorithms". Published online, Accessed 11-03,.2008, http://snowball.tartarus.org/texts/introduction.html.
[12] Y. Yang, "An evaluation of statistical approaches to text categorization", *Information Retrieval*, 1(1-2), 1999, pp. 69-90.
[13] Chih-Chung Chang, Chih-Jen Lin, "LIBSVM: a library for support vector machines", 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.
[14] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, C.-J. Lin, "LIBLINEAR: A library for large linear classification", J*ournal of Machine Learning Research*, 9 (2008), pp. 1871-1874.
[15] Kwangkeun Yi, Hosik Choi, Jaehwang Kim, Yongdai Kim, "An empirical study on classification methods for alarms from a bug-finding static C analyzer", *Information Processing Letters*, vol. 102(2-3), 2007, pp. 118-123.
[16] A. McCallum, "Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering", http://www.cs.cmu.edu/~mccallum/bow, 1996.

Deqing Wang is a Ph.D. student in School of Computer Science & Engineering at Beihang University, P.R.China. Deqing WANG, born on Mar. 1982, in Shandong Province, P.R.China, received his Master degree in computer science from Beihang University, P.R.China on Dec 2007. His research focuses on data mining for software engineering and machine learning.

Hui Zhang is a professor in School of Computer Science & Engineering at Beihang University, P.R.China. He received his Doctor degree in computer science from Beihang University, P.R.China on Dec 2009. His research focuses on web mining and data mining for software engineering.

Rui Liu is a professor in School of Computer Science & Engineering at Beihang University, P.R.China. He received his Doctor degree in computer science from Beihang University, P.R.China in 2011. His research focuses on information extraction, and data mining for software engineering.

Mengxiang Lin is a lecturer at Beihang University, P.R.China. She received her M.E. and Ph.D. degrees from Beihang University in 1993 and 2008, respectively. Her research interests include program analysis and comprehension, data mining and software security.

Wenjun Wu is a professor in School of Computer Science & Engineering at Beihang University, P.R.China. He received his Doctor degree in computer science from Beihang University, P.R.China in 2002. His research focuses on cloud computing and data mining for software engineering.