Volume 2, Issue 4, April 2017 ISSN (online): 2456-0006
International Journal of Science Technology Management and Research
Available online at: www.ijstmr.com

Structured Information Retrival Based Bug Localization

Shraddha Kadam[1]
Department of Computer Engg.,
K.K.Wagh Institute of Engineering Education and Research
Nashik, India

Rutuja Deore[2]
Department of Computer Engg,
K.K.Wagh Institute of Engineering Education and Research
Nashik, India

Kalyani Pawar[3]
Department of Computer Engg,
K.K.Wagh Institute of Engineering Education and Research
Nashik, India

Amol Patil[4]
Department of Computer Engg,
K.K.Wagh Institute of Engineering Education and Research
Nashik, India

Prof. Priti. P. Vaidya[5]
Department of Computer Engg,
K.K.Wagh Institute of Engineering Education and Research
Nashik, India

Abstract: ***Challenging task because defects can deflate from a large variety of sources. So, researchers proposed several automated bug localization techniques where the accuracy can be improved. In this paper, an information retrieval based bug localization technique has been proposed, where buggy files are identified by measuring the similarity between bug report and source code. Besides this, source code structure and frequently changed files are also incorporated to produce a better rank for buggy files. To evaluate the proposed approach, a large-scale experiment on three open source projects, namely SWT, ZXing and Guava has been conducted. The result shows that the proposed approach improves 7% in terms of Mean Reciprocal Rank (MRR) and about 8% for Mean Average Precision (MAP) compared to existing techniques. Software companies spend over 45 percent of cost in dealing with software bugs. An inevitable step of fixing bugs is bug triage, which aims to correctly assign a developer to a new bug. To decrease the time cost in manual work, text classification techniques are applied to conduct automatic bug triage. In this paper, we address the problem of data reduction for bug triage, i.e., how to reduce the scale and improve the quality of bug data. We combine instance selection with feature selection to simultaneously reduce data scale on the bug dimension and the word dimension. To determine the order of applying instance selection and feature selection, we extract attributes from historical bug data sets and build a predictive model for a new bug data set. We empirically investigate the performance of data reduction on totally 600,000 bug reports of two large open source projects, namely Eclipse and Mozilla. The results show that our data reduction can effectively reduce the data scale and improve the accuracy of bug triage. Our work provides an approach to leveraging techniques on data processing to form reduced and high-quality bug data in software development and maintenance.***

Keywords: ***Phishing, URL-Based, Neuro-Fuzzy.***

I. INTRODUCTION

Software bug localization is a technique by which the faulty or buggy locations of the source code are identified. It ensures the quality of software by locating the bugs within the source code and helps the developers to fix those. A large software system often contains numerous bugs. For example, in 2009, users reported 4414 bugs against the Eclipse project [1]. Identifying such number of bugs is time consuming if manual debugging is performed. It is noted that bug fixing consumes 50% to 80% of the total development and maintenance effort [2], [3]. As it is difficult for developers to manually localize and fix those bugs, an automated technique is required. In general, bug

fixing is initiated when the Quality Assurance (QA) team or user reports against the fault of a specific scenario. When the developers receive the bug report, they try to find the buggy locations based on that bug report. For doing this, developers debug the source code and after finding the faulty code, developers fix that bug. This is usually a manual and tedious task, which increases the project time by a large amount. In recent years, some researchers have proposed Information Retrieval (IR) based bug localization techniques where buggy locations are identified using bug reports [1], [4]–[9]. Generally, in bug localization schemes, authors create text corpus from the source code and bug report. Then text corpuses are processed to create relevant information so that word matching between source code and bug report can be performed. Finally, for ranking the source code files, several IR based techniques are applied, returning a ranked list of candidate source files which may contain the bug [1], [4]. Kim et. al proposed a machine learning based bug localization technique where the accuracy depends on the classifier and training data [9]. Moreover, due to considering all words with same weight, the accuracy may be degraded. Zhou et al. proposed a static IR based bug localization technique using revised Vector Space Model () namely "Bug Locator" [1]. Due to providing same weight for all words of the class, the accuracy maybe degraded. The reason is that if a source code contains thousands or more lines and exact matching is performed with a bug report and that source code class or method names, that class may not get higher priority. An extended version of [1] proposed by Ripon et al. where source code structured information (e.g. source code class, method, variable names and comments) is taken into account and matched with the bug report. Due to excluding programming language specific keywords, confusion may arise which may degrade the bug localization accuracy. Moreover, the authors do not give any emphasis on the frequently changed files though those are more liable to create new bugs [3]. In this paper, an automatic bug localization technique has been proposed where bugs are fixed based on the version history e.g. frequently changed source code files) and structural information (e.g. source code class, method name). Bug report features such as summary and description have been extracted and pre-processed to obtain the valid information. In the same way, the source code is also pre-processed. Then a comparison is made between bug report and source code with a view to getting a score for each class. This score is used as ranking and defined as the ranking score of that class. Besides this, for frequently changed files, ranking scores are measured based on the number of times a file has been changed. Finally, a weighted score is calculated using the aforementioned scores.

II. RELATED WORK

In this section, we review existing work on modeling bug data, bug triage, and the quality of bug data with defect prediction.

A. Modeling Bug Data

To investigate the relationships in bug data, Sandusky et al. [10] form a bug report network to examine the dependency among bug reports. Besides studying relationships among bug reports, Hong et al. [11] build a developer social network to examine the collaboration among developers based on the bug data in Mozilla project. This developer social network is helpful to understand the developer community and the project evolution. By mapping bug priorities to developers, Xuan et al. [12] identify the developer prioritization in open source bug repositories. The developer prioritization can distinguish developers and assist tasks in software maintenance. To investigate the quality of bug data, Zimmermann et al. [13] design questionnaires to developers and users in three open source projects. Based on the analysis of questionnaires, they characterize what makes a good bug report and train a classifier to identify whether the quality of a bug report should be improved. Duplicate bug reports weaken the quality of bug data by delaying the cost of handling bugs. To detect duplicate bug reports, Wang et al. [14] design a natural language processing approach by matching the execution information; Sun et al. [15] propose a duplicate bug detection approach by optimizing a retrieval function on multiple features. To improve the quality of bug reports, Breu et al. [9] have manually analyzed 600 bug reports in open source projects to seek for ignored information in bug data. Based on the comparative analysis on the quality between bugs and requirements, Xuan et al. [16] transfer bug data to requirements databases to supplement the lack of open data in requirements engineering.

B. Bug Triage

Bug triage aims to assign an appropriate developer to fix a new bug, i.e., to determine who should fix a bug. Cubrani c and Murphy [12] first propose the problem of automatic bug triage to reduce the cost of manual bug triage. They apply text classification techniques to predict related developers. Anvik et al. [1] examine multiple techniques on bug triage, including data preparation and typical classifiers. Anvik and Murphy [3] extend above work to reduce the effort of bug triage by creating development-oriented recommenders. Jeong et al.

[17] find out that over 37 percent of bug reports have been reassigned in manual bug triage. They propose a tossing graph method to reduce reassignment in bug triage. To avoid low-quality bug reports in bug triage, Xuan et al. [18] train a semi-supervised classifier by combining unlabeled bug reports with labeled ones.

Park et al. [19] convert bug triage into an optimization problem and propose a collaborative filtering approach to reducing the bug fixing time. For bug data, several other tasks exist once bugs are triaged. For example, severity identification [20] aims to detect the importance of bug reports for further scheduling in bug handling; time prediction of bugs models the time cost of bug fixing and predicts the time cost of given bug reports; reopened-bug analysis identifies the incorrectly fixed bug reports t avoid delaying the software release. In data mining, the problem of bug triage relates to the problems of expert finding and ticket routing. In contrast to the broad domains in expert finding or ticket routing, bug triage only focuses on assign developers for bug reports. Moreover, bug reports in bug triage are transferred into documents (not keywords in expert finding) and bug triage is a kind of content-based classification (not sequence-based in ticket routing).

C. Data Quality in Defect Prediction

In our work, we address the problem of data reduction for bug triage. To our knowledge, no existing work has investigated the bug data sets for bug triage. In a related problem, defect prediction, some work has focused on the data quality of software defects. In contrast to multiple-class classification in bug triage, defect prediction is a binary class classification problem, which aims to predict whether a software artifact (e.g., a source code file, a class, or a module) contains faults according to the extracted features of the artifact. In software engineering, defect prediction is a kind of work on software metrics. To improve the data quality, We examine the techniques on feature selection to handle imbalanced defect data. Shivaji et al. [49] proposes a framework to examine multiple feature selection algorithms and remove noise features in classification-based defect prediction. Besides feature selection in defect prediction, We present how to measure the noise resistance in defect prediction and how to detect noise data. Moreover, Bishnu and Bhattacherjee [7] process the defect data with quad tree based k-means clustering to assist defect prediction.

III. LITERATURE SURVEY

Analyzing the existing knowledge about automatic software bug localization, it is found that most of the researchers use IR based bug localization technique to suggest a list of probable buggy files. Zhou et al. proposed an IR based bug localization technique where similarity score was calculated between bug report and source code using revised Vector Space Model (rVSM) [1]. However, if the bug report contained an exact source code class name, the possibility of suggesting that file was very small because the bug report was matched with the whole source code and no priority was given to the class name. Although the authors considered similar bug reports, they did not give any emphasis on the frequently changed files. Due to those reasons, the accuracy of their technique can be improved. Ripon et al. proposed a technique where class, method, variable names and comments were extracted from the source code [4]. At first, authors took the structural information (class, method, variable, comments) from the source code and performed separate search into bug report for each of the structural information and then the total scores across all eight searches were combined with the weight. Still, the weight assignment on each of the structural information was questionable. Besides, although authors gave different weight for different types of words, due to not excluding the programming

language specific keywords, the accuracy of bug localization technique may degrade. For example, if a bug report contains “In dashboard the public data cannot be viewed” and if a source code contains a large number occurrences of “public” word as programming language keyword, the results may be affected due to including this “public” keyword. Brent D. Nichols proposed a bug localization technique based on the previous bug reports [6]. First of all, the semantic meanings of the source code were extracted, then past bugs information were taken into consideration and the similarity was checked between the semantic meanings of the source code and previous bug reports. From that a method document was prepared which contains the meanings of each method. When a new bug was reported, the Latent Semantic Indexing (LSI) was applied on the method document to identify the relationship between the terms of bug reports and the concepts of method documents. However, due to depending on the predefined dictionary of words, this greedy approach may fail if inadequate previous bugs were maintained. Shaowei et al. proposed another bug localization technique where similar bug reports, version history and structure of the source code were amalgamated [7]. Although version history was taken into consideration, the authors did not

consider the number of times a file had been changed rather they just considered how previously a file was modified. Kim et al proposed a two phase recommendation model for localizing software bugs where first phase identified whether the bug report was predictable or deficient by training the system using previous bug information [9]. If first phase identified a bug report as deficient, second phase did not work; otherwise buggy files were being recommended using the static analysis. Still the accuracy was questionable because accuracy depends on the quality of classifier and training data

IV. PRAPOSED SYSTEM

The proposed method introduces a bug localization technique that considers similarity between bug reports and source codes, frequency of fixed source files and occurrence of classes and method names in the bug report to effectively locate bugs. By analyzing the bug reports, it has been seen that a file which is changed frequently to fix a bug, is changed again to fix another one when new bug reports are submitted [3]. As the frequently changed files are more likely to contain bugs, these files should be considered to compute ranking scores. The proposed system also considers class and method names that appear in the bug report because these class and method names are directly mentioned in the bug report [4] and so, these should be given more priority when a final score is calculated. The proposed technique combines all these with the existing rVSM method [1] to further improve the accuracy of the bug localization process. Fig. 1 shows the overall structure of the proposed system. The system is divided into three named components, these are- Modified rVSM (MrVSM) Component, Prioritization Component and Combination Component. The whole process is described in the following subsections.

A. Modified rVSM (MrVSM) Component

The MrVSM component is a combination of applying rVSM method [1] and combining the score with frequency of previously changed files to obtain a MrVSM score, as shown in Fig. 1.

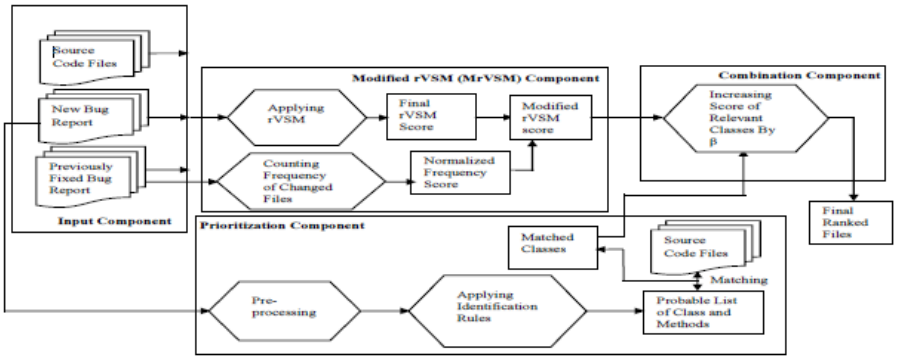


Fig 1:Block Diagram

The MrVSM component is used for improving the accuracy of the rVSM method. The rVSM is a method that modifies commonly used Vector Space Model by considering document length and similar bugs to calculate a final rVSM score. In the rVSM approach, various information such as the source code files, software size and similar bugs are considered for retrieval and ranking. Source code files are considered as documents, and bug reports are considered as queries. Then cosine similarity between source code files and bug report is calculated. As different documents are of different lengths, the weights of different documents are not comparable. So, in cosine similarity, the weight of a vector is divided by its length to make it relative. Thus different similarity values become comparable while ranking.

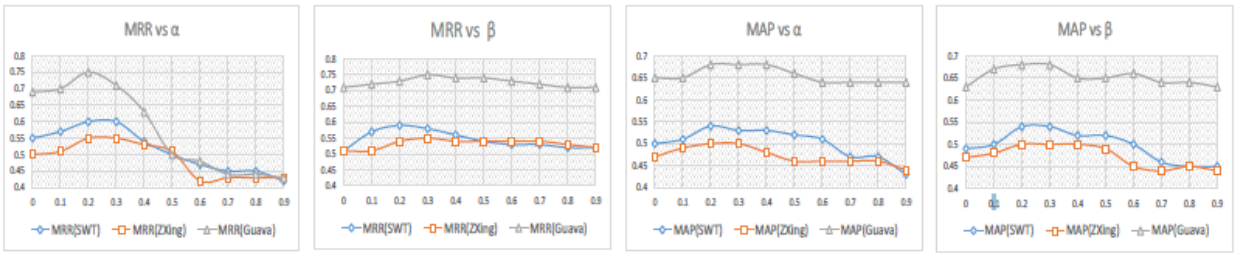
Lukins et al [10] proposed a bug localization technique to extract the semantic and syntactic meaning of source code comments and statements. Authors created a Latent Dirichlet Allocation (LDA) model from the source code which provided word-topic modeling and topic-document distribution. Then the similarity score between each word of the bug report and topics of the LDA model was measured. However, the technique may not be able to predict the appropriate topic because it followed a generative topic model in a probabilistic way [10]. From the above discussion, it is clear that among all existing literatures, most of the existing researches followed IR based technique and the accuracy of those were not as satisfactory as expected. This is because most of the existing works did not consider two

important features and those are version history with frequently changed files and the priority of some structural information such as method and class names.

The proposed system has been tested on three well-known projects collected from Github repositories and all of those contain complete bug database and change history. The results are compared with BugLocator, proposed by Zhou et al. [1].

A. Metrics and Environmental Setup

To measure the performance of the system, Top N Rank, MRR (Mean Reciprocal Rank) and MAP (Mean Average Precision) have been used as metrics. These are widely- used metrics for measuring the effectiveness of a retrieval and ranking system .For all of these, the higher the value,the better the performance. Details are discussed below.



(a) MRR on different α values (b) MRR on different β values (c) MAP on different α values (d) MAP on different β values

Fig. 2: Impact on MAP and MRR for different values of α and β

Top N Rank: This is the number of bugs that are localized in the top N (N = 1, 5, 10 for this system) rank. If one of the fixed files of a bug is in the result set, it is marked as localized.

- MRR: A reciprocal rank is the multiplicative inverse of the rank of the first correct result of a query. For example, if a bug is localized in rank position 4, the reciprocal rank is 1
- MRR is the average of all the ranks of results of a set of query if *n* and *ri* is the number of queries and rank of a query *i*, respectively. MRR is calculated using Equation (1). *MRR* =1*n*_*ni*=11*ri*(1)
- MAP: MAP indicates how successfully the system is able to locate all the buggy files, not only the first one as considered in MRR. MAP is the mean of the average precision values of a set of query. The precision of top n files is calculated using Equation (3).*P*(*n*) =number of buggy files localized in top n*n*(3)Average precision of a query (bug report) is calculated using Equation (4). *AverageP* =_*Kn*=1*p*(*n*) × *b*(*n*)number of buggy files (4) Where *n* is the rank, *K* is the total number of ranked files, *b*(*n*) indicates if *nth* file contains bug or not and *p*(*n*) is the precision of top *n* files as mentioned before. MAP is calculated by taking mean of these average precision values . The experiments have been conducted on 98 fixed bugs of SWT (v3.1), 20 fixed bugs of ZXing and 11 fixed bugs of Guava. SWT contains 484, ZXing contains 391 and Guava contains 1494 source files. To collect the bugs BugZilla was used. All the bugs, along with their bug IDs, open dates, fixed dates, summaries, descriptions and fixed files, have been written to an XML file for experiment.

  Table I: The performance of the proposed system vs Bug Locator (α=0.2 and β=0.2) System Method Top 1

System	Method	Top 1	Top 5	Top 10	MRR	MAP
SWT	Proposed Method	47 (47.9%)	70 (71.4%)	81 (82.7%)	0.60	0.54
	Bug-Locator	35 (35.7%)	68 (69.4%)	79 (80.6%)	0.50	0.42
ZXing	Proposed Method	9 (45)	14 (70%)	15 (75%)	0.55	0.50
	Bug-Locator	8 (40%)	12 (60%)	14 (70%)	0.50	0.44
Guava	Proposed Method	6 (54.6%)	8 (72.7%)	9 (81.8%)	0.75	0.68
	Bug-Locator	5 (45.4%)	7 (63.6%)	8 (72.7%)	0.70	0.63

B. Experimental Result

Fig. (2a) shows the MRR of the system for α ranging from 0.0 to 1.0 for the data sets used. It can be observed that MRR increases from α=0.0 and achieves the highest value between α=0.2 and 0.3 in all the cases. After α=0.3, MRR starts to degrade. So, it achieves highest performance for α between 0.2 and 0.3. In Fig. (2b), MRR seems to show a similar pattern for β ranging from 0.0 to 1.0. It shows that MRR increases from β=0.0 and obtains the highest value between β=0.2 and 0.3. After β=0.3, MRR starts to degrade rapidly. So, the system achieves highest performance for β between 0.2 and 0.3. In Fig. (2c), MAP shows highest value within α=0.2 and 0.3. And in Fig. (2d) MAP is highest for β=0.2 and 0.3. Table I shows the performance of the system and compares it with Bug Locator. For α=0.2 and β=0.2, the proposed method can localize 12%, 2% and 2% bugs more accurately than Bug Locator [1] for SWT in Top 1, Top 5 and Top 10, respectively. For ZXing, using previous α, β, the proposed solution localizes 5%, 10% and 5% more bugs accurately than Bug Locator in Top 1, Top 5 and Top 10, accordingly. For Guava project, using α=0.2 and β=0.3, the proposed solution outperforms 9%, 1% and 1% in Top 1, Top 5 and Top 10, respectively. On an average, the proposed system outperforms Bug Locator by 5.22%. For MRR, The proposed system outperforms Bug Locator by 0.10 for SWT, 0.05 for ZXing and 0.05 for Guava. The MRR improves by 0.07 on average. MAP improves by 0.12

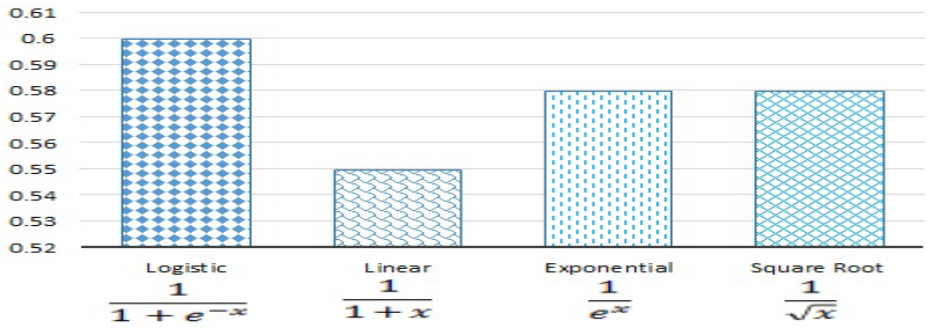


Fig. 3: Influence on MRR values using different functions for SWT, 0.06 for ZXing and 0.05 for Guava.
The average improvement is 0.077. The improvement of these metrics shows the effectiveness of this system.

From Table I and Fig. 2, it is observed that,

1) When β is set to 0, the system still outperforms Bug Locator from the figures because, for β=0, *performance of the proposed system* > *performance of Bug Locator*. So, even if a bug report does not carry class or method names directly from the source codes, it still manages to achieve a better performance.

2) By considering β, performance is improved in all cases because for all β, *MRR of the proposed system* > *MRR of Bug Locator*.

The logistic function has been used for considering frequently changed files. The reason for using the logistic function (Equation (9)) is shown in Fig. 3. From Fig. 3, it is seen that this experiment was conducted by varying the function for α=0.2 and β=0.2. Considering the logistic function results in the highest MRR. So, logistic function was used to calculate the impact of frequently changed files on the proposed system. From the experiment, it is observed that 79 bug reports (81.61%) in an average contain classes and methods directly from the source files on average which clearly shows that a new bug report has higher probability of carrying classes and methods of source files directly. Even if a bug report does not contain these, the proposed method still outperforms the existing methods as mentioned before. So, in all the cases, this method performs better than the already existing ones.

V. SCREENSHOTS OF PROPOSED SYSTEM

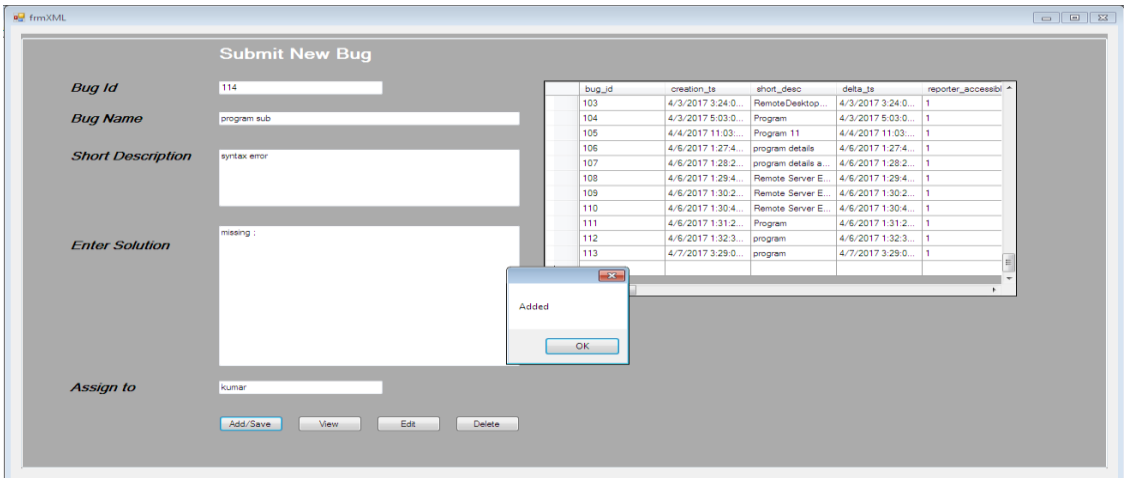


Fig 4: Add Bug Details

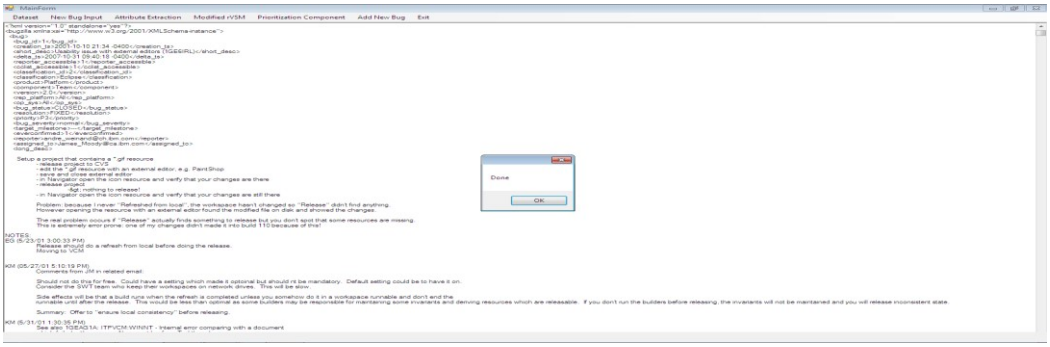


Fig 5: Load Data

Fig 6: Login

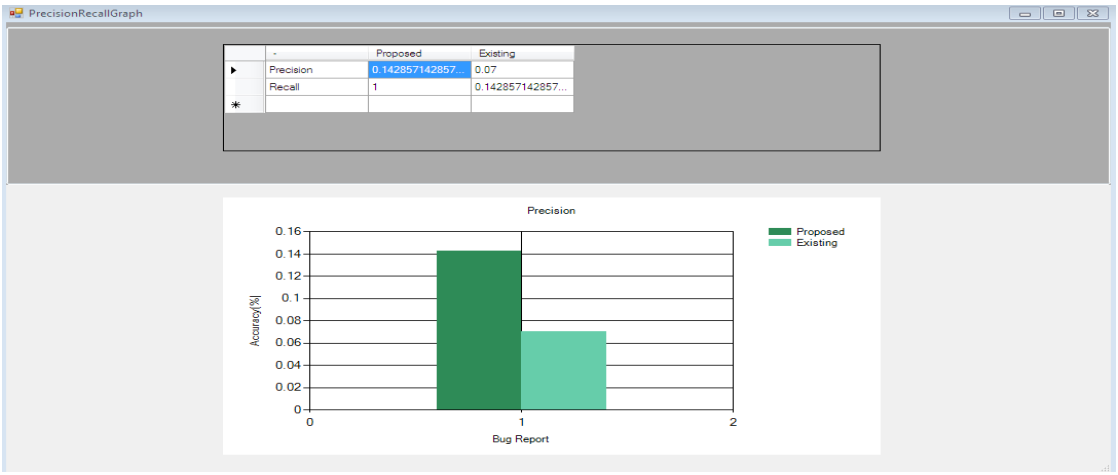


Fig 7: Precision Recall

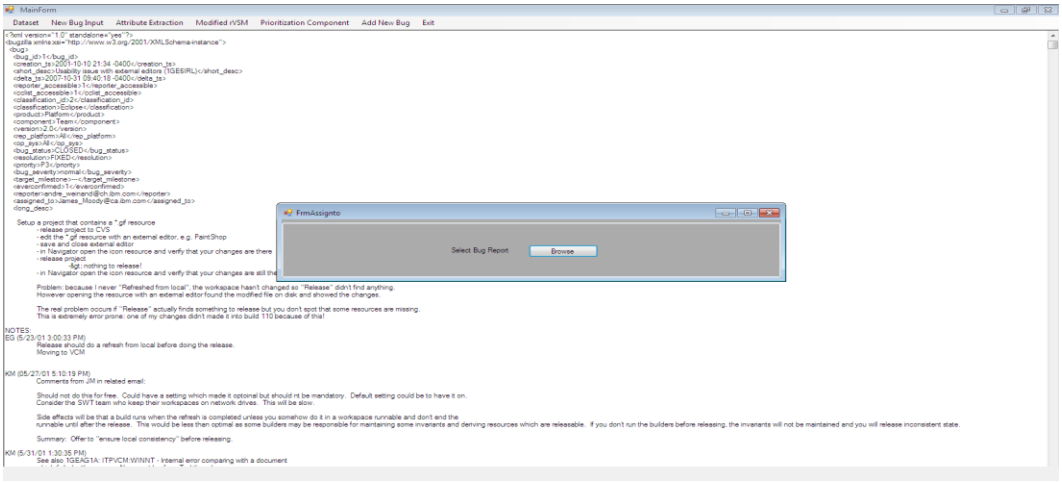


Fig 8: Select Input Bug

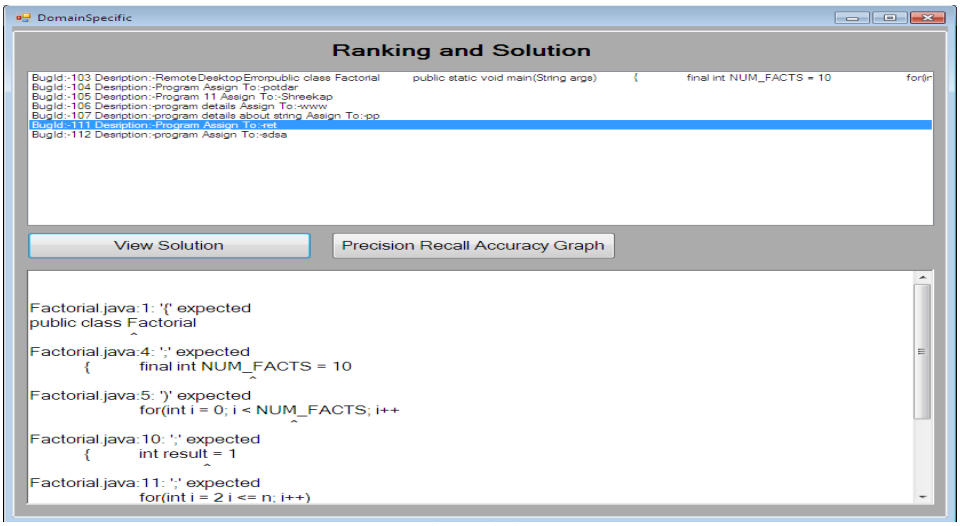


Fig 9:Solution

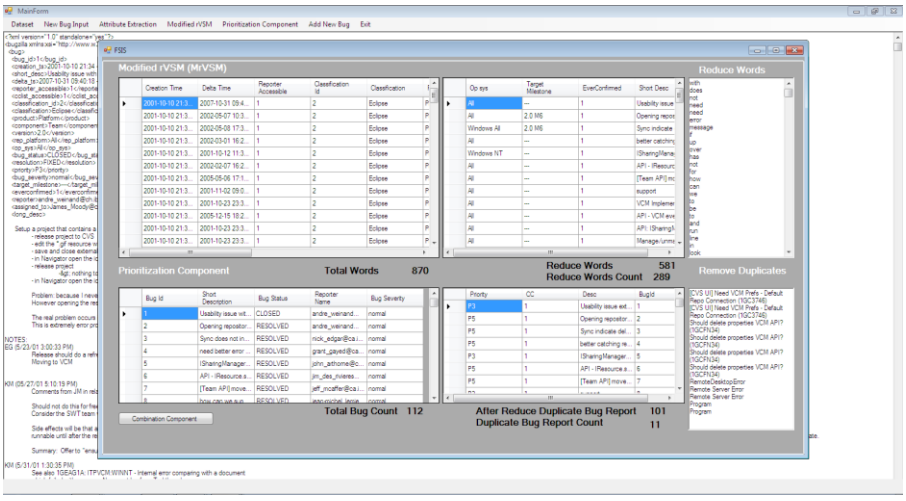


Fig 10: rVSM

CONCLUSION AND FUTURE WORK

In this paper, an IR-based bug localization technique has been proposed to locate relevant source code files using bug report information. The proposed approach introduces Modified revised Vector Space Model (MrVSM) which is derived by considering similar bug reports and source code structure. Moreover, historical data has been incorporated to give source code files more priority.

Experiments on three real world open source projects show that the proposed solution can outperform existing methods namely Bug Locator. In future, the proposed approach will be applied to industrial projects to evaluate its effectiveness in practice.

REFERENCES

1. J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in Proc. 28th Int. Conf. Softw. Eng., May 2006, pp. 361–370.
2. S. Artzi, A. Kie_zun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in web applications using dynamic test generation and explicit-state model checking," IEEE Softw., vol. 36, no. 4, pp. 474–494, Jul./Aug. 2010.
3. J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," ACM Trans. Soft. Eng. Methodol., vol. 20, no. 3, article 10, Aug. 2011.
4. C. C. Aggarwal and P. Zhao, "Towards graphical models for text processing," Knowl. Inform. Syst., vol. 36, no. 1, pp. 1–21, 2013.
5. Bugzilla, (2014). [Online]. Avaialble: http://bugzilla.org/
6. K. Balog, L. Azzopardi, and M. de Rijke, "Formal models for expert finding in enterprise corpora," in Proc. 29th Annu. Int. ACM SIGIR Conf. Res. Develop. Inform. Retrieval, Aug. 2006, pp. 43–50.
7. [P. S. Bishnu and V. Bhattacherjee, "Software fault prediction using quad tree-based k-means clustering algorithm," IEEE Trans. Knowl. Data Eng., vol. 24, no. 6, pp. 1146–1150, Jun. 2012.
8. H. Brighton and C. Mellish, "Advances in instance selection for instance-based learning algorithms," Data Mining Knowl. Discovery, vol. 6, no. 2, pp. 153–172, Apr. 2002.
9. S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information needs in bug reports: Improving cooperation between developers and users," in Proc. ACM Conf. Comput. Supported Cooperative Work, Feb. 2010, pp. 301–310.
10. R. J. Sandusky, L. Gasser, and G. Ripoche, "Bug report networks: Varieties, strategies, and impacts in an F/OSS development community," in Proc. 1st Intl. Workshop Mining Softw. Repositories, May 2004, pp. 80–84.
11. Q. Hong, S. Kim, S. C. Cheung, and C. Bird, "Understanding a developer social network and its evolution," in Proc. 27th IEEE Int. Conf. Softw. Maintenance, Sep. 2011, pp. 323–332.
12. J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer prioritization in bug repositories," in Proc. 34th Int. Conf. Softw. Eng., 2012, pp. 25– 35.
13. T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schr€oter, and C. Weiss, "What makes a good bug report?" IEEE Trans. Softw. Eng., vol. 36, no. 5, pp. 618–643, Oct. 2010.
14. 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 Proc. 30th Int. Conf. Softw. Eng., May 2008, pp. 461–470.
15. C. Sun, D. Lo, S. C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng., 2011, pp. 253–262.
16. J. Xuan, H. Jiang, Z. Ren, and Z. Luo, "Solving the large scale next release problem with a backbone based multilevel algorithm," IEEE Trans. Softw. Eng., vol. 38, no. 5, pp. 1195–1212, Sept./Oct. 2012.
17. G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with tossing graphs," in Proc. Joint Meeting 12th Eur. Softw. Eng. Conf. 17th ACM SIGSOFT Symp. Found. Softw. Eng., Aug. 2009, pp. 111–120.
18. J. Xuan, H. Jiang, Z. Ren, J. Yan, and Z. Luo, "Automatic bug triage using semi-supervised text classification," in Proc. 22nd Int. Conf. Softw. Eng. Knowl. Eng., Jul. 2010, pp. 209–214.
19. J. W. Park, M. W. Lee, J. Kim, S. W. Hwang, and S. Kim, "Costriage: A cost-aware triage algorithm for bug reporting systems,"
20. A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in Proc. 7th IEEE Working Conf. Mining Softw. Repositories, May 2010, pp. 1–10. in Proc. 25th Conf. Artif. Intell., Aug. 2011, pp. 139–144.