# Improving Bug Triage with Relevant Search

Xinyu Peng, Pingyi Zhou, Jin Liu*, Xu Chen
State Key Lab of Software Engineering
Computer School, Wuhan University
Wuhan, China
*Corresponding author
{ pengxinyu, zhou_pinyi, jinliu, xuchen}@whu.edu.cn

***Abstract*—Bug triage is a process where bugs are assigned to developers. In large open source projects such as Mozilla and Eclipse, bug triage is time-consuming because numerous bugs are submitted everyday. To improve bug triage, many studies have proposed automatic approaches to recommend proper developers for resolving bugs. These approaches are based on machine learning algorithms, which treat bug triage like text classification. Although they are effective, the accuracy of them can be further improved. Our goal is to propose a method not only has good performance but also is simple. We propose a method based on relevant search technique to recommend developers for the given bugs. First, we construct an index for bugs to make them searchable. Then, for a given bug to be assigned, we utilize the index to search for the bugs related to it. Finally, we analyze these related bugs and recommend developers based on them. We conduct experiments on bugs of Mozilla and Eclipse to evaluate our method. The results indicate that our method has a good performance and outperforms machine learning algorithms like Naïve Bayes and SVM.**

***Keywords—bug triage; bug report assignment; issue tracking; problem tracking***

## I. INTRODUCTION

In software maintenance, bug resolution plays an important role. To effectively manage bugs and coordinate efforts, large open source projects often incorporate bug tracking systems. Bugzilla is a typical example of bug tracking systems, which proposed by Mozilla and adopted by many famous open source projects like Eclipse. Bugs in bug tracking systems are represented by bug reports, which are documents with many fields to record information about the corresponding bugs. The developers in projects depend on the bug reports to manage bugs and exchange opinions.

Figure 1 shows an Eclipse bug report. It has freeform text such as summary and description, which demonstrate what the bug is about and the steps to reproduce the bug. It has some category fields such as product and component. This bug belongs to product JDT and component UI. If a bug has been assigned to a developer like this one, the field named assigned to would refer to the developer. In this case, the developer is Claude Knaus. Besides, status field indicates the stage of the bug in the life-cycle of bug reports. When a bug report is submitted, its status is UNCONFIRMED. The status changes to NEW after a developer verify it. The same developer is often responsible for finding an appropriate developer to fix the bug. After that, the status changes to ASSIGNED. If the bug has been fixed and been verified, its status changes to RESOLVED and VERIFIED or CLOSED respectively. The resolution of a bug in status field could be any among FIXED, DUPLICATE, WORKFORME, WONTFIX, INVALID. For example, the status in Figure 1 is VERIFIED FIXED.

**Product**: JDT **Component**: UI

**Version**: 2.0 **Platform**: PC Windows 2000

**Severity**: normal **Priority**: P3

**Status**: VERIFIED FIXED

**Assigned to**: Claude Knaus

**Summary**: Template - pressing new presents an error

**Description**: go to Preferences->Java->Templates - press the new button - you get an error saying that the template name must not be null. This is annoying since I didn't have the change to specify one.

Figure 1. Bug Report #4353 of Eclipse

The process where a developer determine which developer is most appropriate to a bug and assigned it to the developer is called bug triage. If a bug is assigned to an improper developer, it leads to the bug re-assignment. The probability of a bug been fixed decreases as the number of re-assignment increases [3]. As the open source projects become more complicated, the number of bug reports submitted everyday constantly increase. It is inefficient that all bug reports are manually assigned. The software development is delayed by the inefficacy. Besides, it gives users an impression that developers are unresponsive and disregard the users' bug reports. This terrible impression will destroy the project's community [1]. To improve the efficiency of bug triage, several studies have proposed automatic methods to recommend appropriate developers for any given bug [1] [2]. These methods are based on machine learning algorithms. They treat bug triage as text classification. Although the methods are effective, the accuracy of them can be further improved.

In this paper, we propose a method utilizing relevant search technique. Our method essentially builds a search application for bug reports. With this search application, we can find relevant bug reports for a given bug then analyze them to get the corresponding developers. To make the bug reports searchable, we first construct an index for them. Given a new bug, we build

DOI reference number: 10.18293/SEKE2017-041

a query based on its summary and description and narrow the search scope with its product and component. Based on the search results, we use their relevance scores to rank developers corresponded to the search results of bug reports. The developers in top rank are recommended for the given bug. Lucene is used to implement the search application.

This paper makes two contributions:

1.It proposes a new relevant search based method to recommend developers for bug triage.

2.It conducts experiments to evaluate the approach on two datasets: Eclipse and Mozilla.

The remainder of the paper is organized as follows. Section 2 demonstrates our approach. Section 3 presents the experiment results. Section 4 discusses our method. Section 5 describes related work. Section 6 concludes this paper.

## II. APPROACH

Figure 2 shows our overall framework of bug triage. The framework consists of five steps. It first extracts necessary information such as summary, description, product and component from the original bug reports in step 1. Then it indexes the extracted information in step 2. When we need to recommend developers for a new bug report, it first executes the same information extraction and indexing steps like step 1 and step 2 but just for the given report in step 3 and step 4. Then it searches for the similar bug reports in step 5. Last it uses the searching results to rank the developers and generates a list of them as the recommendation in step 6.

### A. Data Preprocessing

The original bug reports have much information that our method does not need. We extracted the necessary information for our method: summary, description, product, component, fixer. Because summary and description are unstructured text, to eliminate data noise, we remove stop words and pure numbers, which do not contain useful information representing the bug reports, then stem them by using Porter stemmer. We join the summary and the description together as content after all.

### B. Indexing

To make the searching process possible, we first indexing bug reports. In the process of indexing bug reports, they are treated as documents consisting of fields. Specifically, a bug report is a document having fields like content, product, component. Moreover, the original texts of bug reports are tokenized to terms and transformed to a data structure called inverted index. An inverted index is a sorted list of terms. Each term is associated with pointers linked to documents containing the term. Because the inverted index is sorted and have the pointers for each term, it is efficient for locating a term in the inverted index and finding which documents have the term. It likes the index of books. If we are interested in a subject of a book, we can directly look for the terms about the subject in the index, and go to the related pages without scan through the entire book.

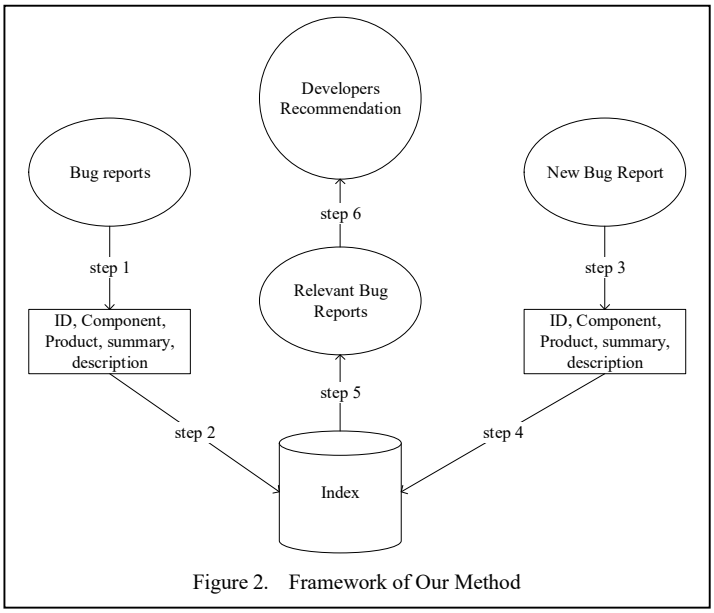

Figure 2. Framework of Our Method

### C. Searching

Given a bug to be assigned, we first index it, then construct a query based on its content. Besides, we use its component and product to narrow the searching scope for reducing the noise. At first, we only consider bugs have same component and product as the current bug. If the bug is the first one in its component and product, there are no search results. In this situation, we consider bugs only have the same product. If the bug even is the first one in its product, there are still no search results. Finally, to ensure results exist, we search for related bugs no matter what their component and product are.

The searching results are returned with scores, which represent the relevance between the current bug and others. For consistent with the relevant search context, we use query $q$ and document $d_i$ denote the current bug and one bug of the results respectively. The formula below is used to compute these scores.

$$score(q,d_i) = corrd(q,d_i) \times queryNorm(q) \times \sum_{t\ in\ q} \left(tf(t\ in\ d_i) \times idf(t)^2 \times t.getBoost \times norm(t,d_i)\right) \quad (1)$$

In the Equation 1, $tf(t\ in\ d_i)$ stands for the term's frequency, defined as the number of times term t appears in the current document $d_i$ . $idf(t)$ stands for inverse document frequency. This value correlates to the inverse of $docFreq$(the number of documents in which the term t appears).

$coord(q,d_i)$ is a score factor based on how many of the query terms are found in the specified document.

$queryNorm(q)$ is computed as Equation 2, which is a normalizing factor used to make scores between queries comparable. However, this factor does not affect document ranking, since all ranked documents returned by one query are multiplied by the same factor.

$$queryNorm(q) = \frac{1}{\sqrt{\left(q.getBoost^2 \times \sum_{t\ in\ q}(idf(t) \times t.getBoost)^2\right)}} \quad (2)$$

In the Equation 1 and 2, $t.getBoost$ is the weight of term $t$ which is specified in the query. It can be changed to make some term more important than others. However, we treat all terms equal and did not change the weight. $q.getBoost$ is the weight of query $q$, which is used to control the importance of queries and we did not change it too.

$$norm(t, d_i) = lengthNorm \times \prod_{field\ f\ in\ d\ named\ as\ t} f.boost \quad (3)$$

$norm(t, d_i)$ is computed as Equation 3. $f.boost$ is the weight of the field $f$, which play the same role as the two boost before. We did not change it neither. $lengthNorm$ is a factor in accordance with the number of tokens in a field, shorter fields contribute more to the score.

### D. Ranking

After searching, a bunch of related bugs ranked by the scores is returned. Assuming the $score(q, d_i)$ and $score_{max}$ represent the score of $i$th bug and the max score of all results respectively, we normalize the scores according to the Equation 4, which results in normalized scores within the range of [0, 1].

$$score_{norm}(q, d_i) = score(q, d_i) \div score_{max} \quad (4)$$

Because each bug in the results corresponds to a developer who fixed it. After analyzing searching results, we can get a set of developers. Assuming $D$ denotes the set, $dev_i$ denotes the score of the $i$th developer in $D$ and $A_i$ denotes the set of bugs in searching results which are assigned to the $i$th developer, we computed the $dev_i$ as in the Equation 5.

$$dev_i = \sum_{d_i\ in\ A_i} score_{norm}(q, d_i) \quad (5)$$

After computing the scores of the developers in $D$, we rank them by their scores. The top developers in the rank are recommended for the bug to be assigned.

## III. Experiments And Results

### A. Dataset

We evaluated our methods on two datasets from two open source software: Eclipse and Mozilla. All data were collected from the websites of their bug tracking systems. For Eclipse, we collected bug reports from 2001-10-11 to 2007-12-14. We drop bug reports before the bug whose id is 4354 because their descriptions are discussions among developers, which are not actual descriptions added when the bug reports are created. Besides, these bug reports were submitted in a very short time, which indicates they were migrated from other bug tracking system. For Mozilla, we collected bug reports from 1998-04-07 to 2008-08-11. We retained bug reports with status CLOSED and FIXED as prior studies did [2], [3], [6], [9].

We extracted bug fixers by checking the "assigned to" fields in the bug reports following previous studies. However, the "assigned to" fields in many bug reports are set to generic names which do not correspond to real people [19]. In Eclipse, bug reports are assigned to generic names like "JDT-UI-Inbox", "JDT-Text-Inbox", "JDT-Core-Inbox". In Mozilla, bug reports are assigned to "nobody". Because these generic names do not represent real developers, we do not recommend them and exclude bug reports whose fixers are among them from the data. Then to reduce noise, bug reports whose fixers appear less than ten times are excluded [19].

After above steps, 91251 bug reports, 650 fixers, 72 products and 450 components are left in Eclipse dataset, while 100964 bug reports, 777 fixers, 59 products and 492 components are left in Mozilla dataset.

To make our experiments are more like the situation in practice, we adopt the longitudinal data setup in [9] [19]. We perform a 10-round incremental analysis on the two project datasets. The bug reports extracted from the two projects are first sorted in chronological order of creation time and then divided into 11 equally-size folds. We form 10 rounds evaluation with the 11 folds. First, in round 0, we create an index using bug reports in fold 0, and we update the index and evaluate our method using the first bug, and then update the index and evaluate our method using the second bug report, and so on for all bug reports in fold 1. Then, in round 1, we proceed in a similar way like fold 1 to test using bug reports in fold 2, and so on. After round 10, we compute the average evaluation metric among all rounds.

To make our results comparable, we choose a metric called Recall@K used in many prior studies [5] [19]. Recall@K is the proportion of bugs whose associated developers is ranked in the top k (k = 1, 3, 5) of the returned results. Given a bug report, if the top k results contain the developer who fixed the bug, we consider the developer is located. So Recall@K of all test bugs equal to the proportion of how many recommendations contain the actual fixer.

### B. Research Questions

In this paper, we are interested in the following research questions:

**RQ1: How is our method effective compared with other baselines?**

To answer the question, we compare our method with the existing machine learning based developer recommendation methods, such as those based on Naïve Bayes and SVM [1], [2], [12]. We use the scikit-learn package to implementation Naïve Bayes and SVM respectively.

**RQ2: How does the performance of our method change with the recommendation list increase?**

To answer this question, we evaluated our method varying different recommendation list size from 1 to 10. We can see the trends after drawing the evaluated results.

Meanwhile, we used coverage rate to measure the best score our method can reach. The coverage rate is computed like Recall@K, but it considers all developers instead of Top K developers. So it gives the upper bound for our method. Because only if there exist the actual fixers in the search results, our method has the chance to pick them out, which means Recall@K is always smaller than or equal to the coverage rate.

**RQ3: How does product and component information influence the effectiveness of our method?**

TABLE I. EVALUATION RESULTS OF TWO PROJECTS

| Project | Rank | SVM | Naïve Bayes | Our Method |
|---|---|---|---|---|
| Eclipse | Top1 | 0.307 | 0.186 | 0.438 |
| | Top3 | 0.518 | 0.258 | 0.725 |
| | Top5 | 0.613 | 0.286 | 0.841 |
| Mozilla | Top1 | 0.223 | 0.147 | 0.333 |
| | Top3 | 0.426 | 0.227 | 0.551 |
| | Top5 | 0.518 | 0.293 | 0.634 |

To figure out the influence of the product and component combinations over our method, we compare the results between our method with and without narrowing the search scope using the product and component information.

## C. *Results*

**RQ1: How is our method effective compared with other baselines?**

Table I compares the performance of our method with baselines in terms of Recall@1, Recall@3, and Recall@5. For Eclipse, our method achieves average Recall@1 value 0.438, which means that for 43.8% bugs, it successfully recommends their associated developers as top 1. The Recall@5 value is 0.841, which means that for 73.85% bugs, their developers can be found in the top 5 return results. For Mozilla, our method achieves Recall@1 and Recall@5 values 0.333 and 0.634, respectively. It obtains better results than the conventional machine learning based recommendation methods. Comparing with Naïve Bayes, the results of our method are 116% ~ 194% better. Comparing with SVM, the results of our method are 22% ~ 49% better.

TABLE II. COVERAGE RATE OF TWO PROJECTS

| Project | metric | without feature | with feature |
|---|---|---|---|
| Eclipse | Coverage rate | 0.907 | 0.953 |
| Mozilla | Coverage rate | 0.821 | 0.795 |

**RQ2: How does the performance of our method change with the recommendation list increase?**

Figures 3, 4 present the values from Recall@1 to Recall@10 of our method with coverage rate for Eclipse and Mozilla respectively. We notice that Recall@K values increase along with K values increasing and tend to be stable finally. The Recall@K values are close to the coverage rate from Recall@8, which means the actual fixers are in the Top 8 list already and increasing size of recommendation list is almost useless.

**RQ3: How does product and component information influence the effectiveness of our method?**

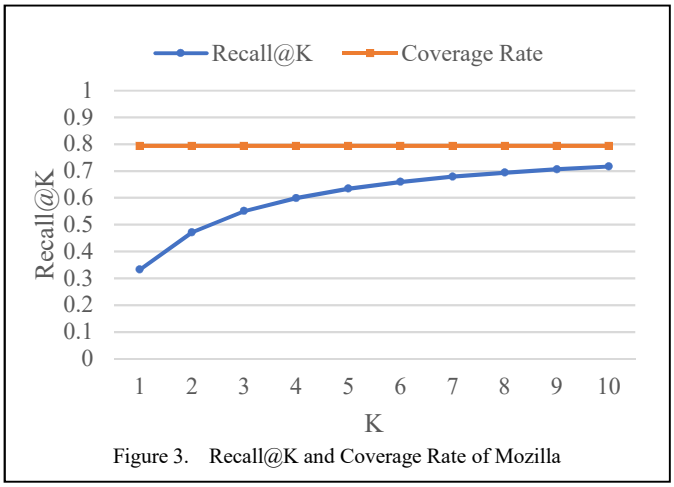

Figure 3. Recall@K and Coverage Rate of Mozilla

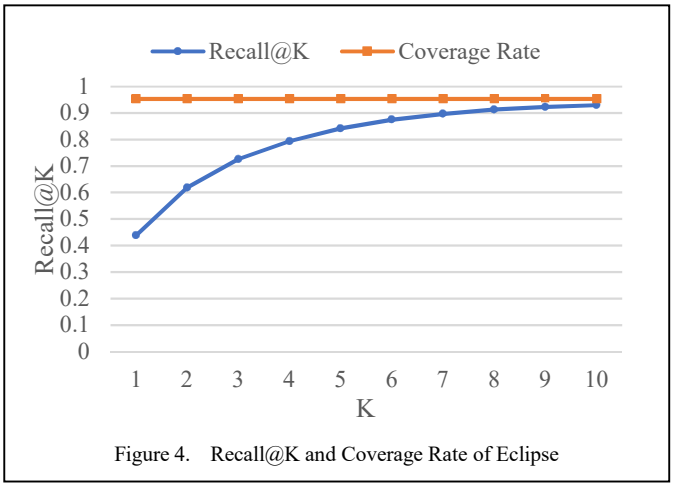

Figure 4. Recall@K and Coverage Rate of Eclipse

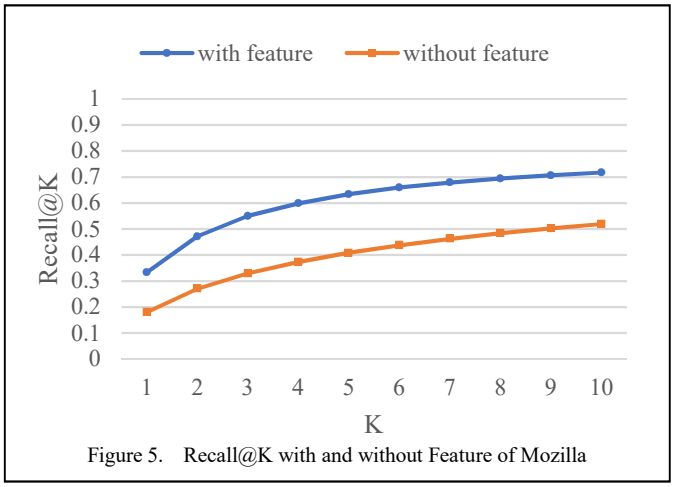

Figure 5. Recall@K with and without Feature of Mozilla

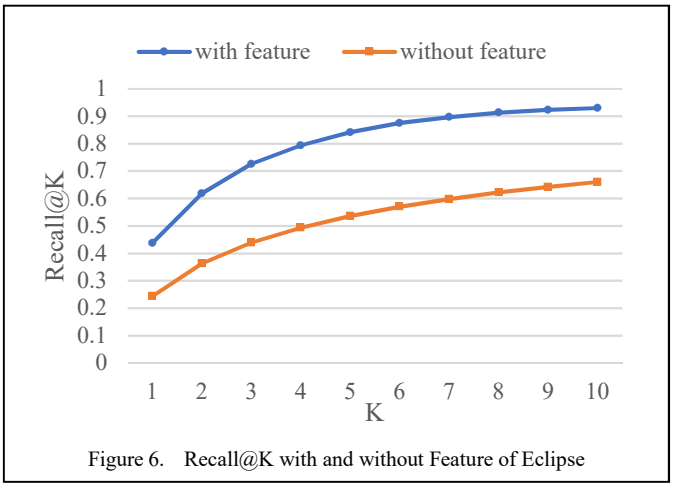

Figure 6. Recall@K with and without Feature of Eclipse

Table II presents the coverage rate of our method with and without product and component information for Eclipse and Mozilla. Note that the coverage rates are similar for both projects, which indicates our method has similar chance to pick out the actual fixer despite with or without narrowing the search scope.

Figures 5, 6 presents the values of Recall@1 to Recall@10 of our method with and without features. By narrowing the search scope with product and component information, the Recall@K values increase for both Eclipse and Mozilla. The relative improvements range from 41% to 80% for Mozilla and from 38% to 84% for Eclipse.

## IV. Discussion

### A. Advantages

In essence, our method is based on relevant search technique, while the machine learning based methods treat bug triage like a text classification task.

Machine learning based methods regard each developer as a class and each bug report as a document. If the number of classes is small, these methods are effective enough. However, when the number of classes increases to hundreds or even thousands, it is hard to design an accurate and efficient classifier [5].

Our method adopts relevant search technique to construct a search engine for bug reports. It utilizes the search results with the relevance scores to get a ranked recommendation list. Besides, to reduce the noise, it uses the component and product information of bug reports to narrow the search scope, which significantly improves the performance. Because our method only need add new bug reports to index for updating, and searching in the index is fast. It is easily extensible and able to deal with large-scale data using our method.

### B. Limitations

Our method essentially has some limitations. In some cases, it will fail to recommend the actual developer due to the practice in bug triage.

First, if a bug is fixed by a new developer within its search scope, our method cannot recommend the right developer. Because the fixer does not correspond to any bug in the search scope, our method only recommends developers who have fixed at least one bug before. Besides, some developers may have fixed many bugs relevant to a bug. However, at the time of bug triage, they are not free for some reasons, so the bug is assigned to another less relevant developer [5].

### C. Threats to Validity

Internal validity is an estimate of the degree to which a causal conclusion based on a study can be made. To improve the internal validity, we preprocess our datasets following previous studies [9] [19]. We only collected bug reports with status CLOSED and FIXED to ensure the bug reports are related to bugs and have been fixed. To determine the fixer of a bug, we examine the "assigned to" field of the bug report. However, other studies chose another way to determine fixers. They proposed a heuristic approach to extracted the fixers [1] [2]. We do not adopt their heuristics because recent studies do not follow them. Besides, we exclude the bug reports whose "assigned to" fields are generic names or appear less than 10 times to reduce the noise.

External validity is an estimate of the degree to which our experiment results can be generalized. Although we have analyzed 91251 and 100964 bug reports from Eclipse and Mozilla, our method may not appropriate to other projects. To improve external validity, we plan to experiment our method on more new bug reports from more projects in the future.

Construct validity is an estimate of the degree to which metrics can be trusted. To determine the effectiveness of our method, we need check the actual fixer is whether in the recommendation list or not. Recall@K is a proper metric for this purpose. It also has been adopted by many prior bug triage studies [5] [19].

## V. Related Work

According to the different techniques used to automatic bug triage, Tao Zhang et al. [4] classify the prior studies into five categories: machine learning-based approach, expertise model-based approach, tossing graph-based approach, social network-based approach and topic model-based approach.

As a pioneering study, Cubranic et al. [1] regard bug triage as a text categorization problem. In their work, each assignee was considered to be a single class, and each bug report was assigned to only one class. Anvik et al. [2][12] experimented several different machine-learning algorithms including Naïve Bayes, SVM, C4.5 to recommend a list of appropriate developers for fixing a new bug. The experiment results show SVM performed better than others on their datasets. Ahsan et al. [15] use feature selection and Latent Sematic Indexing [16] to reduce the dimensionality of the term-to-document matrix. Their results showed the bug triage system combined LSI and SVM has the best performance. Xuan et al. [13] proposed a semi-supervised text classification method, which utilizing expectation-maximization based on both labeled and unlabeled data to enhance Naïve Bayes classifier. To remove the noisy data, Zou et al. [18] adopted feature selection. Their results showed that feature selection could improve the performance using Naïve Bayes by up to 5%. Xia et al. [14] proposed a method using a composite model called DevRec, which combined bug reports-based and developers-based analysis.

Jeong et al. [3] proposed a tossing graph model to capture bug tossing history and use this model to improve bug triaging prediction accuracy. Their experiments demonstrated the tossing graph model could improve the accuracy of automatic bug triage using machine-learning algorithms such as Naïve Bayes and Bayesian Network [17] only. Bhattacharya et al. [6][7] improved the accuracy of bug triage by utilizing a multi-feature tossing graph and increment learning.

Matter et al.[8] modeled developers' expertise using the vocabularies found in their source code files and compare them to the terms appearing in corresponding bug reports. Tamrawi et al. [9] proposed Bugzie, an automatic bug triaging tool based on fuzzy set and cache-based modeling of the bug-fixing expertise of developers. Xuan et al. [10] achieved developer prioritization via social network analysis to improve the performance of

automatic bug triage using SVM or Naïve Bayes only. Hao Hu et al. [5] utilized the historical bug-fix information to constructed a network to captures the knowledge of "who fixed what, where". They use the network to recommend suitable developers.

Naguib et al. [11] adopted LDA to cluster the bug reports into topics. Then they created the activity profile for each developer of the bug tracking repository by mining history logs and bug report topic models. An activity profile consists of two parts, including developer's role and developer's topic associations. By utilizing activity profile and a new bug's topic model., they proposed a ranking algorithm to find the most appropriate developer to fix the given bug. Xin Xia et al. [19] proposed a specialized topic modeling named MTM which extends LDA by considering product and component of information of bug reports. They used MTM to get the topic distribution of a new bug report to assign an appropriate fixer based on the affinity of the fixer to the topics.

Regarding relevant search, Zhou et al. [20] utilized indexing and searching to recommend tags for software information sites.

## VI. CONCLUSION

Bug triage is a time-consuming process if all bugs are manually assigned to developers in large open source projects. In this paper, we proposed a method based on relevant search technique. To improve efficiency and effectiveness of bug triage, our method recommends developers for bugs to be assigned. Besides, we utilized component and product information of bug reports to improve the performance of our method further. We have evaluated our method on bug reports of Eclipse and Mozilla. For Eclipse, our method achieved 43.8% and 84.1% accuracies when recommending 1 and 5 developers. For Mozilla, the accuracies are 33.3% and 63.4% for recommending 1 and 5 developers. All of the results of our method outperforms the SVM and Naïve Bayes method.

With the advent of big data era [21][22], in future work, we plan to incorporate bug tossing graphs into our method to introduce more relevant developers in the top of recommendation. Also, we plan to utilize more features of bug reports to improve the performance further.

## ACKNOWLEDGEMENT

This work is partly supported by National Natural Science Foundation of China (NSFC) (grant No. 61572374, U163620068, U1135005), the Academic Team Building Plan from Wuhan University and National Science Foundation (NSF) (grant No. DGE-1522883).

## REFERENCES

[1] Murphy G, Cubranic D. Automatic bug triage using text categorization. InProceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering 2004.

[2] Anvik J, Hiew L, Murphy GC. Who should fix this bug?. InProceedings of the 28th international conference on Software engineering 2006 May 28 (pp. 361-370). ACM.

[3] Jeong G, Kim S, Zimmermann T. Improving bug triage with bug tossing graphs. InProceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering 2009 Aug 24 (pp. 111-120). ACM.

[4] Zhang T, Jiang H, Luo X, Chan AT. A Literature Review of Research in Bug Resolution: Tasks, Challenges and Future Directions. The Computer Journal. 2016 May 1;59(5):741-73.

[5] Hu H, Zhang H, Xuan J, Sun W. Effective bug triage based on historical bug-fix information. InSoftware Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on 2014 Nov 3 (pp. 122-132). IEEE.

[6] Bhattacharya P, Neamtiu I. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. InSoftware Maintenance (ICSM), 2010 IEEE International Conference on 2010 Sep 12 (pp. 1-10). IEEE.

[7] Bhattacharya P, Neamtiu I, Shelton CR. Automated, highly-accurate, bug assignment using machine learning and tossing graphs. Journal of Systems and Software. 2012 Oct 31;85(10):2275-92.

[8] Matter D, Kuhn A, Nierstrasz O. Assigning bug reports using a vocabulary-based expertise model of developers. InMining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on 2009 May 16 (pp. 131-140). IEEE.

[9] Tamrawi A, Nguyen TT, Al-Kofahi JM, Nguyen TN. Fuzzy set and cache-based approach for bug triaging. InProceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering 2011 Sep 5 (pp. 365-375). ACM.

[10] Xuan J, Jiang H, Ren Z, Zou W. Developer prioritization in bug repositories. InSoftware Engineering (ICSE), 2012 34th International Conference on 2012 Jun 2 (pp. 25-35). IEEE.

[11] Naguib H, Narayan N, Brügge B, Helal D. Bug report assignee recommendation using activity profiles. InMining Software Repositories (MSR), 2013 10th IEEE Working Conference on 2013 May 18 (pp. 22-30). IEEE.

[12] Anvik J, Murphy GC. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. ACM Transactions on Software Engineering and Methodology (TOSEM). 2011 Aug 1;20(3):10.

[13] Xuan J, Jiang H, Ren Z, Yan J, Luo Z. Automatic Bug Triage using Semi-Supervised Text Classification. InSEKE 2010 Jul (pp. 209-214).

[14] Xia X, Lo D, Wang X, Zhou B. Accurate developer recommendation for bug resolution. InReverse engineering (WCRE), 2013 20th working conference on 2013 Oct 14 (pp. 72-81). IEEE.

[15] Ahsan SN, Ferzund J, Wotawa F. Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine. InSoftware Engineering Advances, 2009. ICSEA'09. Fourth International Conference on 2009 Sep 20 (pp. 216-221). IEEE.

[16] Hofmann T. Probabilistic latent semantic indexing. InProceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval 1999 Aug 1 (pp. 50-57). ACM.

[17] Friedman N, Nachman I, Peér D. Learning bayesian network structure from massive datasets: the «sparse candidate «algorithm. InProceedings of the Fifteenth conference on Uncertainty in artificial intelligence 1999 Jul 30 (pp. 206-215). Morgan Kaufmann Publishers Inc..

[18] Zou W, Hu Y, Xuan J, Jiang H. Towards training set reduction for bug triage. InComputer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual 2011 Jul 18 (pp. 576-581). IEEE.

[19] Xin Xia, David Lo, Ying Ding, Jafar M. Al-Kofahi, Tien N. Nguyen, and Xinyu Wang. Improving Automated Bug Triaging with Specialized Topic Model. IEEE Transactions on Software Engineering (TSE), IEEE CS, 2016. in press

[20] Zhou P, Liu J, Yang Z, Zhou G. Scalable Tag Recommendation for Software Information Sites. InThe 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER) 2017 Feb 20.

[21] Xu, Z., Liu, Y., Mei, L., Hu, C. and Chen, L., 2015. Semantic based representing and organizing surveillance big data using video structural description technology. Journal of Systems and Software, 102, pp.217-225.

[22] Liu, J., Yu, X., Xu, Z., Choo, K.K.R., Hong, L. and Cui, X., 2016. A cloud‐based taxi trace mining framework for smart city. Software: Practice and Experience.