2014 IEEE 38th Annual International Computers, Software and Applications Conference

Towards Semi-automatic Bug Triage and Severity Prediction Based on Topic Model and Multi-Feature of Bug Reports

Geunseok Yang
Department of Computer Science
University of Seoul, Seoul, Korea
ypats87@uos.ac.kr

Tao Zhang
Department of Computer Science
University of Seoul, Seoul, Korea
kerryking @uos.ac.kr

Byungjeong Lee*
Department of Computer Science
University of Seoul, Seoul, Korea
bjlee @uos.ac.kr

***Abstract*—Bug fixing is an essential activity in the software maintenance, because most of the software systems have unavoidable defects. When new bugs are submitted, triagers have to find and assign appropriate developers to fix the bugs. However, if the bugs are at first assigned to inappropriate developers, they may later have to be reassigned to other developers. That increases the time and cost for fixing bugs. Therefore, finding appropriate developers becomes a key to bug resolution. When triagers assign a new bug report, it is necessary to decide how quickly the bug report should be addressed. Thus, the bug severity is an important factor in bug fixing. In this paper, we propose a novel method for the bug triage and bug severity prediction. First, we extract topic(s) from historical bug reports in the bug repository and find bug reports related to each topic. When a new bug report arrives, we decide the topic(s) to which the report belongs. Then we utilize multi-feature to identify corresponding reports that have the same multi-feature (e.g., component, product, priority and severity) with the new bug report. Thus, given a new bug report, we are able to recommend the most appropriate developer to fix each bug and predict its severity. To evaluate our approach, we not only measured the effectiveness of our study by using about 30,000 golden bug reports extracted from three open source projects (Eclipse, Mozilla, and Netbeans), but also compared some related studies. The results show that our approach is likely to effectively recommend the appropriate developer to fix the given bug and predict its severity.**

Keywords—bug triage; severity prediction; topic model; multi-feature; corrective software maintenance

I. INTRODUCTION

Recently, software systems have been getting increasingly more complex. Most of the systems inevitably have defects that include errors that can eventually lead to software failure, so bug fixing is an important and strenuous phase in the software maintenance.

Open source software projects and huge business software projects generally adopt the bug tracking system (e.g., Bugzilla) [1] to investigate errors in lines of code for fixing the bugs and to manage project milestones. When a bug report that was written by a reporter (e.g., common user or developer) arrives in the bug repository, a triager (e.g., senior developer or project manager) assigns the bug to a developer for fixing by using the bug tracking system. However, many bug reports about 37% to 44% might be reassigned to another developer for fixing them [2]. Also, bug assignment is manually executed, which is a time-consuming task so that bug lifetime and developers' workloads are increasing. In addition, the bug severity is a feature that helps decide how quickly it should be fixed. In other words, triagers decide which bugs should be fixed first according to severity. High-severity represents critical errors (e.g., "blocker," "critical," "major"), and low-severity denotes bugs categorized as "minor" or "trivial." They may assign the high-severity bugs as well as the low-severity bugs to developers. Reasons for these problems include the following.

- A large number of bug reports submitted every day increases the triagers' workloads. This inevitably leads to incorrect bug report assignment.
- Due to lack of knowledge about the given bug report, a triager might assign the wrong developer to fix it or be confused as to its severity, so fixing time is extended.

To solve these problems, in this paper, we propose a novel method for semi-automatic bug triage and severity prediction by introducing topic model and multi-feature. First, from the bug repository, we extract the topics of all bug reports to establish a base of topics from them. When a new bug comes, we compare the new bug report against the base topics to select which topic(s) the bug report belongs to. Then we extract both candidate developers and bug reports related to topic(s). Finally, we utilize multi-feature of the bug report to rank the appropriate developers and predict the severity of the given bug. This paper presents the following contributions.

- To our knowledge, we first try to combine topic model and multi-feature to find corresponding reports having the strongest relevance (i.e., the same topic and the same multi-feature) with the new bug report. Using this new technique, we believe it can improve the accuracy of bug triage and severity prediction.
- For the bug triage, we not only build a social network to analyze developers' behaviors in the bug

* Corresponding Author

0730-3157/14 $31.00 © 2014 IEEE
DOI 10.1109/COMPSAC.2014.16

fixing process, but also capture the number of assignments and the number of attachments as factors to improve the accuracy of bug triage.

- By comparing related studies in the evaluation experiment, we showed that our proposed approach can effectively recommend developers and predict the severity of bugs. In addition, we determine which feature is the most important in this process.

The paper is organized as follows. Section II introduces background knowledge and the motivations for our study. Section III discusses related works and differences in our research. In Section IV, we describe how to recommend appropriate developers and to predict the severity of a given bug using the proposed approach. Section V outlines the experiment and the results. We discuss the results and present some threats in Section VI. Section VII concludes this paper and introduces future work.

II. BACKGROUND KNOWLEDGE AND MOTIVATION

A. Summary of Bug Report

A bug report is freeform textual content. Generally, it includes the defaults appearing in the source code files. The report consists of predefined fields such as "Bug ID," "Title," "Component," and so on. Figure 1 shows the detailed information of an Eclipse JDT bug report[1] with "Bug ID" number 36465.

Figure 1. Eclipse JDT bug report (# 36465)

We note that this bug report was submitted by Jason Sholl. The current status of the report is "CLOSED," and "FIXED," and the report is also classified as Eclipse product "JDT" and component "Core." In addition, the bug was assigned to a developer named Philipe Mulet to be fixed. The bottom of the report presents detailed information, which includes the bug description, the attachment (e.g., patch, test case, etc.) and some remarks posted by commenters. It is worth noting that the commits do not appear in this bug report. Actually, the commits are stored in the history log of the bug repository. If developers change the code lines, they should send message(s) to declare the code changes. We think this feature can enhance verification of developers' experience in bug fixing.

In bug reports, the important meta-fields are "Product," "Component," "Priority," and "Severity." We adopt them as multi-feature. We note that the "Product" is the first category in developing customers' requirements such as "JDT" and "PDE" in Eclipse. "Component" denotes a sub-category belonging to a specific product, such as "Debug" and "UI" in Eclipse. Both "Priority" and "Severity" are scale effect factors as to whether the given bug report is urgent or not. Also, "Priority" denotes a fixing priority that includes 5 levels (P1 to P5), where P1 represents the highest priority and P5 denotes the lowest. For instance, bug report 36465 has the highest priority P1, which means this bug report needs to be fixed at the earliest opportunity. "Severity" stands for the degree of severity for the bug. In a general way, it includes 7 degrees (blocker, critical, major, normal, minor, trivial, and enhancement). The enhancement severity is not bug which requires for new functions. However, in our study, we include it in severity prediction. Among these degrees, blocker, critical, and major denote significant problems in the program, such as crashes, loss of data, or a severe memory leak; normal, minor, trivial, and enhancement represent minor errors, like a cosmetic problem or a minor loss of function. Among them, blocker means that the bug is the most severe type, and trivial indicates that the bug is the least severe type. For example, bug report 36465 was marked "blocker," which is a most severe fault. In our work, we focus on recommending the most appropriate developer to fix a new bug, predicting the severity of it by utilizing topic model and multi-feature.

B. Life Cycle of Bug Fixing

Generally, current status (such as "UNCONFIRMED," "RESOLVED," "CLOSED," etc.) is shown in the status field of the bug report. When a bug report is submitted, the status of the bug report is "UNCONFIRMED." Then a triager confirms its presence and changes the status to "NEW." Next, the triager assigns the bug report to a developer for fixing, and the status changes to "ASSIGNED." The developer tries to fix this bug according to the summary (or title) and description, and during this time, the status might be marked as "RESOLVED," "FIXED," or perhaps "DUPLICATE," etc. Finally, the triager verifies whether the bug is fixed, and if the bug fixing task has been completed, the status is marked "CLOSED." However, if the bug is not fixed completely, it may be reopened by the triager to be reassigned to another developer, or the original assignee might reopen the bug report to try and fix it again. In this situation, the status changes to "REOPENED."

C. Topic Model

Most of the developers contribute to their specific interest in bug fixing. In other words, developers are not involved in

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=36465

all bug reports [3]. The bug reports that describe similar contents should be classified as the same topic. Therefore, topic model can help us to find historical bug reports that have the same topic with a new bug report. In this paper, we adopt Latent Dirichlet Allocation (LDA) [4] to extract the topics and corresponding topic terms. The Stanford Topic Modeling Toolbox (TMT) is used to implement LDA. A brief introduction of LDA and TMT is as follows:

1) Latent Dirichlet Allocation

As a topic model, LDA used in our study models the documents in the same category used by some of the given topics. Each topic is characterized by a distribution of words that frequently co-occur in the documents.

In our work, we utilize LDA to collect historical bug reports belonging to the same topic as a new bug report. For a given pre-defined topic number (e.g., 30) and a set of historical bug reports, where each bug report is a sequence of terms, LDA extracts the topics from these reports. After the topics are extracted, the topic model can be used to determine which topic the new bug report belongs to. We detail this process in Section IV Subsection B.

2) Stanford Topic Modeling Toolbox

TMT brings topic modeling tools to perform analysis on data sets. It helps us to train an LDA model to create the categories of the bug reports by producing the topics.

In TMT, there are four parameters (N, R, α and β) that need to be set. N stands for the number of topics; R denotes the number of iterations; and α and β are association factors. The higher the value of α, the higher the probability of a bug report being associated with multiple topics; the higher the value of β, the higher the probability of a topic being associated with multiple terms.

D. Social Network

A social network is built by a set of social actors (e.g., developers) and the corresponding ties (or edges) between these actors. In our research, these ties reflect the relationship between the developers participated in the bug fixing process [5]. Comments and commits can visualize this relationship in the social network. Comments are posted by commenters who suggest ideas to solve given bugs and who write their own self-memos; commits are posted by the assignees (i.e., bug fixers) who indicate any changes to the source code and the change history of the bug reports. By analyzing the activities among the social network, we can verify the developers' experience in bug fixing.

E. Preprocessing

Preprocessing is a basic step of LDA and a textual similarity measure between bug reports. Preprocessing that includes tokenization, stemming, and stop-word removal is well known as a natural language processing (NLP) technique. Tokenization is used to parse a character stream into a sequence of word tokens; stemming is responsible for transforming terms to their base forms; stop words are insignificant words. For example, there is a new bug report that says, "*I tried to fix, but I have no idea about this problem.*" Through tokenization, the sentence is divided into words. Among these words, 'tried' is changed to 'try' after stemming. And removing the stop words eliminates 'I', 'to', 'but', etc. As a result, we are left with "try," "fix," "no," "idea," and "problem."

F. Textual Similarity Measure

Before executing bug triage and severity prediction, we need to compute the textual similarity between bug reports. In this subsection, we introduce how to implement this process.

1) Smoothed Unigram Model

In order to measure the textual similarity between bug reports, we utilize smoothed Unigram Model (UM) [6] to transform the bug reports into probability vectors. Different from the traditional Vector Space Model (VSM) [7] which represents the bug report as numeric vectors, smoothed UM transforms the bug report into probability vectors. As evidence, Rao and Kak [6] demonstrated that smoothed UM performed better than VSM when executing bug localization. Even if bug localization is different from bug triage, both of them also need to transform bug reports to vectors. Therefore, the finding from Rao and Kak becomes a reason why we choose smoothed UM instead of VSM.

Definition 1: Smoothed Unigram Model

$$P_{suni}(\omega|\overrightarrow{\omega_k}) = (1-\mu)\frac{\omega_k(n)}{\sum_{n=1}^{|R_k|}\omega_k(n)} + \mu\frac{\sum_{l=1}^{K}\omega_l(n)}{\sum_{l=1(l\neq k)}^{K}\sum_{n=1}^{|R_l|}\omega_l(n)}$$

where

- ω is a term, and $\overrightarrow{\omega_k}$ is a weight vector of report k.
- $|R_k|$ is the number of words in report k.
- $\omega_k(n)$ is the value for the occurrence frequency of the n^{th} vocabulary term in report k.
- K denotes the total number of reports, and μ is the different weight of two parts in this equation.

2) KL Divergence

We utilize KL divergence [8] to measure the textual similarity between a query (a new bug report) and historical reports in the bug repository. Definition 2 shows how to implement KL divergence based on smoothed UM.

Definition 2: Similarity Measure via KL Divergence

$$\mathrm{sim}(\overrightarrow{\omega_q},\overrightarrow{\omega_k}) = -\mathrm{KL}\left(P_{suni}(\omega|\overrightarrow{\omega_q}), P_{suni}(\omega|\overrightarrow{\omega_k})\right)$$

$$= -\sum_{i}^{i=|\omega_i|} P_{suni}(\omega_i|\overrightarrow{\omega_q}) * \log\frac{P_{suni}(\omega_i|\overrightarrow{\omega_q})}{P_{suni}(\omega_i|\overrightarrow{\omega_k})}$$

where

- $P_{suni}(\omega|\overrightarrow{\omega_q})$ stands for the probability of term ω appearing in query q.
- $P_{suni}(\omega|\overrightarrow{\omega_k})$ represents the probability of term ω appearing in bug report k.

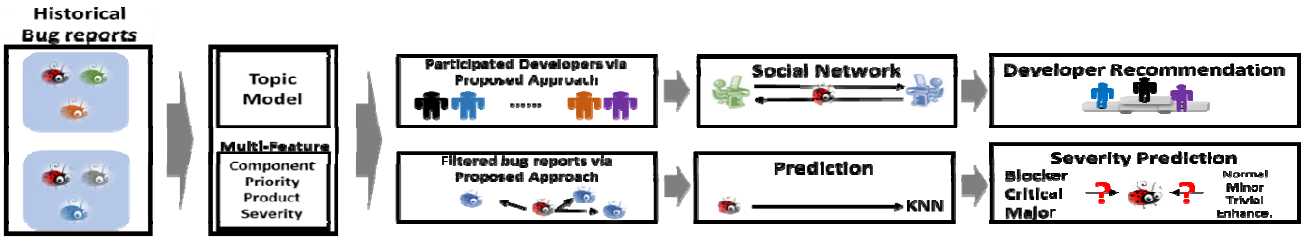

Figure 2. Overview of our approach

G. K-Nearest Neighbor Algorithm

KNN [9] is a non-parametric lazy learning algorithm. It can find the top *k* objects that are similar to a given instance. In this study, when a new bug report arrives, KNN is adopted to search the top *k* similar historical bug reports having the same topic and multi-feature with the given bug report. KL divergence shown in Definition 2 is used to measure the textual similarity between the bug reports. By considering the severity labels of the *k* nearest neighbors, we can predict which severity status the new bug belongs to.

H. Motivation

A topic model can help us to find the historical bug reports that have the same topic(s) with the new bug report; however, the accuracy of bug triage and severity prediction need to be improved.

Therefore, we propose multi-feature (e.g., product, component, severity and priority) to implement this goal. We believe that bug reports having the same multi-feature with the new report are very similar to it. Ranking the candidate developers extracted from these bug reports can enhance the effectiveness of bug triage. Moreover, by adopting the features coming from these bug reports as the input of KNN, we can accurately predict the severity of the given bug. In a word, combing topic model and multi-feature is a good way to effectively implement bug triage and severity prediction.

III. RELATED WORK

A. Semi-automatic Bug Triage

In recent years, some research has utilized various techniques (e.g., social network, topic model) to enhance the effectiveness of bug triage.

Wu et al. proposed an approach called DREX, which used KNN to find historical bug reports similar to a new given bug, and introduced social network metrics (such as in-degree and out-degree) to recommend the correct developers for fixing the given bug [10]. Xuan et al. modeled developer prioritization in Eclipse and Mozilla bug reports based on the social networking technique [11]. By analyzing communication among developers in the comments, they ranked the developers based on their ability to accomplish three tasks, one of them being bug triage. The results of the experiment showed that developer prioritization was helpful in improving the performance of bug triage. Park et al. modeled "developer profiles" to indicate the estimated costs for fixing different types of bugs [12]. LDA was used to verify the bug types and quantify each value of the developer's profile as the average time to fix the bugs related to each type. By considering the probability and cost to fix the given bug, the proposed method can decide on the most appropriate fixer. Naguib et al. created an activity profile for each user from the history of all her activities (review, resolve and etc.) [13]. They think the activity profile can influence and contribute to the identification of fixers and fixer rankings. Experiment results showed this approach outperforms the LDA-SVM–based assignee recommendation technique. Xie et al. proposed an approach called DRETOM to model developers' interest in, and expertise on, bug-resolving activities based on topic models that are built from historical bug reports [14]. Experimental results on Eclipse JDT and Mozilla Firefox showed that DRETOM can achieve high recall up to 82% and 50% with top-5 and top-7 developers, respectively.

Our work is different from these previous studies. Even if we also utilize LDA to verify which topic a new bug report belongs to, we adopt multi-feature (e.g., product, component, priority and severity) to find the corresponding bug reports that have the same values of these features with the new bug report. To our knowledge, this is a first work to combine topic model and multi-feature of bug reports to pick the bug reports which have a strong relevance with the given bug.

B. Severity Prediction

Previous works usually adopted machine learning algorithms to predict the severity of the given bugs. Menzies and Marcus designed and built a tool named SEVERIS, which is based on a rule-learning technique to effectively predict the severity of bugs [15]. Lamkanfi et al. utilized a Naïve Bayes classifier to predict whether a new bug report belongs to a "non-severe" or "severe" category [16]. In a later work, Lamkanfi et al. compared the effectiveness of severity prediction using four different machine learning algorithms such as Naïve Bayes, Naïve Bayes multinomial, KNN and SVM [9]. The evaluation results showed that Naïve Bayes multinomial performed better than the other algorithms. Tian et al. first measured the similarity of different bug reports using BM25 textual similarity function, and then introduced KNN to decide the appropriate severity label for a new bug report [17]. Bhattacharya et al. constructed graphs that captured software structure at two different levels, including the product level and developer collaboration level [18]. They identified a set of graph metrics that capture interesting properties of these graphs. By analyzing the evaluation results on 11 open source programs (e.g., Firefox, Eclipse, MySQL and etc.), they found that the

graph metrics can be used to effectively predict the severity of given bugs.

Different from the aforementioned research, we utilize topic model and multi-feature to capture the historical bug reports that have strong relevance (i.e., the same topic and the same multi-feature) to a new bug report. Moreover, although we also use KNN to predict the severity of the bug, KL divergence, which is different from cosine similarity and BM25 similarity measures, is used to measure the texture similarity as the input to KNN. We believe that these differences can improve the accuracy of severity prediction.

IV. METHODOLOGY

A. Overview

Figure 2 illustrates an overview of our approach. First, we take each new bug report into coreNLP to perform preprocessing to get word tokens. Then, we use these word tokens to find similar topic(s) related to the new bug report. We extract the participating developers from the historical bug reports according to the corresponding topic(s). By introducing multi-feature that includes "severity," we filter developers from the bug reports who have different multi-feature from the new bug report so that a list of candidate developers is produced. Finally, we analyze the candidate developers' behavior (e.g., commenting activity and commit activity) via the social network so that we can recommend the appropriate developers. Next, in order to predict the severity of the bugs, we also utilize multi-feature that excludes "severity." Then, we filter the corresponding bug reports and produce the potential bug reports. We compute the similarity between the new bug report and historical bug reports via KL divergence, and take these similarity measures into KNN so that we can predict the severity of the new bugs.

B. Topic Modeling

Before building a topic model, we bring the predefined fields of the bug report (e.g., "summary," and "description") into Stanford coreNLP, which returns word tokens. Then we concatenate the word tokens from each bug report as features and plug these features into Stanford TMT to produce the terms for each topic. Table I presents extracted terms from TMT for each topic in Eclipse.

TABLE I. EXTRACTED ECLIPSE TOPIC(S) AFTER TMT

	Topic-1	Topic-2	Topic-3	Topic-4
Rank 1	time	context	import	readme
Rank 2	set	text	export	api
Rank 3	run	help	jar	datatool
Rank 4	execution	menu	initial	release
Rank 5	debug	dialog	message	note
Rank 6	memory	list	format	package
Rank 7	zero	open	handle	specific
Rank 8	incorrect	search	performance	entry
Rank 9	throw	icon	correctly	point
Rank10	analysis	display	schema	isery

When a new bug report arrives, we select the matching topic(s). Specifically, we count the frequency of the topic terms that appear in the new bug report. If the frequency is the highest, we consider the given bug report as belonging to the topic(s). The bug reports belonging to the same topic(s) with the new bug report are extracted to execute bug triage and severity prediction.

C. Semi-automatic Bug Triage

In order to recommend appropriate developers for fixing the given bugs, we utilize topic model and multi-feature (described in Section II Subsection A) to implement this goal. Figure 3 shows the framework for developer recommendation.

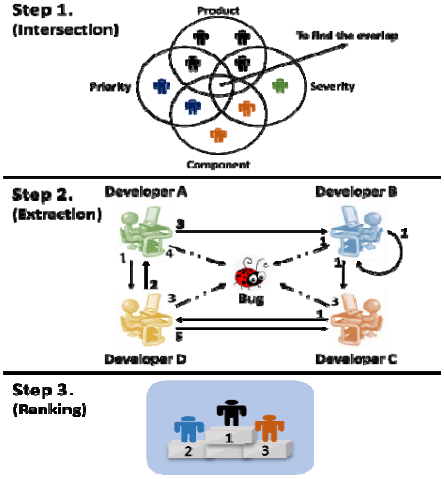

Figure 3. Semi-automatic bug triage

First, we verify the topic(s) the new bug report belongs to. Then, we not only extract all participating developers, including assignees and commenters, from the historical bug reports on the same topic with the given bug report, but also obtain the corresponding developers extracted from the bug reports that have the same multi-feature (e.g., "Product," "Component," "Priority," and "Severity"). The developers' intersection process is shown as follows:

$$D = D_{product} \cap D_{component} \cap D_{priority} \cap D_{severity}$$

We take D as a set of candidate developers who have contributed to the bug reports. These bug reports have the same topic, product, component, priority, and severity. Next, to analyze the candidate developers' behavior, we collect developers' activities (e.g., comment and commit activity) by parsing XML files from Eclipse, Mozilla and Netbeans repositories. Moreover, we adopt the number of assignments and attachments as factors to enhance bug triage. Finally, we compute the candidate developers' ranking score as seen in Definition 3.

Definition 3: Developers' Ranking Score (DRScore)

$$\text{AScore(Dev)} = \frac{Dev_{assign}/Dev_{attach} + 1}{\sum_{n=1}^{30}(Dev_{assign}/Dev_{attach} + 1)}$$

$$\text{SScore(Dev)} = \frac{Dev_{commit} + Dev_{comment}}{\sum_{n=1}^{30}(Dev_{commit} + Dev_{comment})}$$

$$\mathrm{DRScore(Dev)} = \gamma * \mathrm{AScore(Dev)} + (1-\gamma) * \mathrm{SScore(Dev)}$$

where

- $Dev_{assignments}$ is the number of assignments to developer *Dev* for fixing bugs.
- $Dev_{attachments}$ denotes the number of attachments (e.g., uploaded patch files, etc.) in the bug reports assigned to *Dev*.
- Dev_{commit} stands for the number of commits (e.g., changes to the histories of the bug reports) sent by *Dev*.
- $Dev_{comment}$ represents the number of comments (e.g., suggestions and opinions about the given bugs) posted by *Dev*.
- n is the number of topics.

In this definition, except for adopting the number of commits and comments posted by the candidate developers in the social network, we also consider the number of assignments and attachments to further identify the developers' experience in bug fixing. Thinking in terms of empirical analysis, the higher the number of assignments, the more the developer's experience. We found that the number of commits can also help us verify the developer's experience, in the same way as assignment count. Hooimeijer and Weimer [19] demonstrated that the comment count is a positive coefficient, which means assigned bugs received more user attention, so that the bug has a high probability of being fixed. In addition, they also found that attachment count is a negative coefficient because the presence of attachments means increased costs for fixing bugs. Based on these conclusions, we assign Dev_{assign}, Dev_{commit}, and $Dev_{comment}$ as the numerator and Dev_{attach} as the denominator. By computing DRScore for each candidate developer, we can determine the most appropriate developer.

D. Severity Prediction

In the same manner, to predict the severity of bugs, we also utilize topic model and multi-feature. Figure 4 presents the workflow for severity prediction.

As shown in Step 1, at first, we decide which topic(s) the new bug report belongs to and collect the historical bug reports in the same topic. These reported bugs should have the same multi-feature (e.g., "Product," "Component," and "Priority"). We take the intersection of these bug report sets as follows:

$$B = B_{product} \cap B_{component} \cap B_{priority}$$

We take these filtered reports (B) as final extracted reports. In Step 2, we compute the textual similarity between B and the new bug report using smoothed UM and KL divergence. Finally, by importing these similarity measures, we utilize KNN executed in R language [20] to predict the severity of the new bug as being either "Blocker," "Critical," "Major," "Normal," "Minor," "Trivial," or "Enhancement." Specifically, KNN is utilized to find the most similar bug reports to the given bug report. By analyzing the severity status of these reports, we can easily predict the severity of the new bug.

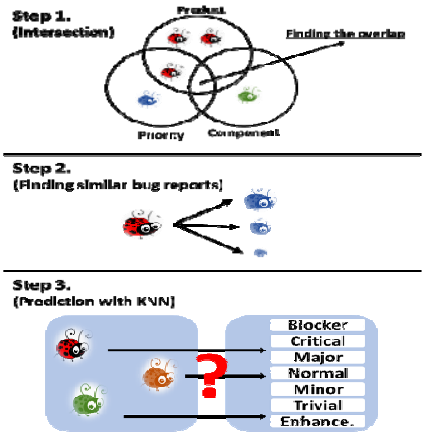

Figure 4. Semi-automatic severity prediction

E. Case Study

In this subsection, we present an example of our approach executed for developer recommendation. Due to the limited the space, we do not show the case study of severity prediction. When a new bug arrives, first, we preprocess the description of the given new bug report: "*The module does not work. I attach the patch file. This file will add an exception error in the subsystem.*" As a result, we get terms such as "module," "work," "attach," "patch," "file," "add," "exception," "error," and "subsystem." Then we determine that this bug report belongs to topic-1 due to the highest term frequency (e.g., three times in Topic-1, one time in Topic-2 and two times in Topic-3). Next, in Topic-1, "Yang" as an assignee (bug fixer) was assigned five and seven times to fix the bug reports that belong to Topic-1 and Topic-2, respectively. Also, he uploaded patches 17 times and three times to the bug reports belonging to Topic-1 and Topic-2, respectively. He posted comments 12 times and three times to bug reports belonging to Topic-1 and Topic-2, respectively. And he sent commits three times related to the bug reports for Topic-1 and five times for Topic-2. As a result, we get the Activity Score (*AScore*) of 0.49 ((5 / (17+1)) / ((5+7) / (17+3+1))) and the Social Score (*SScore*) of 0.65 ((3+12) / ((3+5) + (12+3))). Therefore, the ranking score of "Yang" is 0.62 (0.2*0.49 + 0.8*0.65) when the weight vector γ is set to 0.2.

V. EXPERIMENT

A. Experiment Setup

We collect the data set (1,646,218 bug reports) from three projects in the different repositories by parsing XML files. Then, to implement the experiment for evaluating our work, we manually select the golden bug reports (about 30,000) which provide much more the information. The bug reports were labeled "RESOLVED," "CLOSED," or "FIXED." Table II shows the details of our data set. In Table II, "Type" denotes whether the data set is for construction model or test. "Size" stands for the number of bug reports in each type. "Period" means the submission period of the corresponding bug reports.

To implement the task of bug triage, we extracted all developers, including assignees and commenters who participated in the bug reports for each topic. Then, we filtered related developers who came from bug reports that have different multi-feature. In this process, we exclude the severity, which is one of multi-feature, in Netbeans Java, because severity feature is not supported by the Netbeans repository.

TABLE II. OUR DATA SET

Projects	Type	Size	Period
Eclipse	Construction Model	9,000	'01/10/10~'11/11/03
	Test	1,000	'11/11/03~'14/01/07
Mozilla	Construction Model	8,987	'00/06/19~'11/08/20
	Test	999	'11/08/22~'14/01/16
Netbeans (Java)	Construction Model	8,940	'99/03/30~'12/06/11
	Test	993	'12/06/11~'13/12/03

Table III shows the number of products, components, and extracted developers in Eclipse, Mozilla, and Netbeans.

TABLE III. INFORMATION EXTRACTED FROM OUR DATA SET

Projects	# of products	# of components	# of developers
Eclipse	90	299	1,947
Mozilla	58	457	5,450
Netbeans	1	18	138

To our knowledge, for the bug triage, a related study to our work involves DRETOM, which recommends appropriate developers by using topic model. Apart from that, activity-based recommendation (ActivityRE) is an assignee recommendation system using activity profile–based topics. Out-Degree is a kind of social network metric in DREX, which performs developer recommendation. Thus, as a baseline for our work, we considered DRETOM [14], ActivityRE [13] and Out-Degree [10]. In the same manner, for the task of severity prediction, we considered Naïve Bayes [9] and single KNN [9] as a baseline to our approach.

To measure the effectiveness of the proposed method, we used Precision [21], Recall [21], F-measure [21] and MRR [22].

$$\text{Precision(q)} = \frac{TP}{TP+FP} \quad (4)$$

$$\text{Recall(q)} = \frac{TP}{TP+FN} \quad (5)$$

$$\text{F} - \text{Measure(q)} = 2 * \frac{Precision(q)*Recall(q)}{Precision(q)+Recall(q)} \quad (6)$$

$$\text{MRR(q)} = \frac{1}{N_q}\sum_{i=1}^{N_q}\frac{1}{R_i} \quad (7)$$

We note that q presents the new bug reports (e.g., test data set). In the case of bug triage, TP (true positive instances) is the number of candidate developers recommended correctly. FP (false positive instances) is the number of candidate developers recommended incorrectly. FN (false negative instances) is the number of actual developers, but who were not recommended. In the case of severity prediction, TP is the number of bugs whose severity status was predicted correctly. FP is the number of bugs whose severity status was predicted incorrectly. FN is the number of bug reports with the corresponding severity status which were not predicted by our approach. N_q is the number of total test data sets. R_i represents the matched rank number. For example, we produce the candidate developers (i.e., Yang, Zhang, and Lee) using our method. Then, we verify the developers with the actual assignee (i.e., Zhang) in the test data set. The ranked number is second candidate developer, so R_i is represented by 2. If the assignee (e.g., Zhang) is changed to another developer (e.g., Yang) to resolve the bug, we take this developer (e.g., Yang). In such a case, R_i is represented by 1.

To evaluate our approach, we tried to answer the following research questions (RQx).

RQ1: Are the appropriate developers successfully recommended in the proposed approach?

RQ2: Is the severity of the bugs successfully predicted in the proposed approach?

Our work aims to recommend appropriate developers and to predict the severity of bugs. The eventually goal is to make a tool that performs both recommendation of developers and prediction of severity. Both RQ1 and RQ2 are important questions. Before we compare our work and related studies, we need to identify the effectiveness of our tasks.

RQ3: Which approach can effectively perform the task of bug triage?

RQ4: Which approach can effectively perform the task of severity prediction?

In the same manner, to answer RQ3 and RQ4, we evaluated the performance of our work and previous studies to verify whether the proposed approach outperforms other bug triage and severity prediction methods. In these research questions, we use the statistical test [23][24] as well as the above evaluation metrics to identify which metrics are most effective.

RQ5: Which feature (e.g., product, etc.) is the most important impact factor in developer recommendation and prediction of severity?

For this question, we want to know which feature is the most important impact factor in the bug triage and severity prediction tasks.

B. Parameter Selection

Before implementing our approach, we need to verify a suitable parameter (γ). First, we manually checked 50 randomly selected bug reports from the test data sets in each project (Eclipse, Mozilla, and Netbeans Java). Then, we carried out developer recommendation. Next, we compared the candidate developer list against the actual developer (assignee) corresponding to the test data. If the actual assignee was changed (e.g., the bug report is resolved), we consider the changed developer as an actual assignee. Finally we compute the MRR values (formula 7). Table IV shows the results of MRR.

We note that the suitable value of the parameter in Netbeans Java is 0.2. On the other hand, the value in Eclipse and Mozilla is 0.3. We use these parameter settings to implement developer recommendation.

Except for the weight vector γ, we decide the value of the parameter μ to be 0.4 in smoothed UM (Definition 1) and N=30, R=1500, α=0.01 and β=0.01 in TMT (described in Section II Subsection C-2) and K=1 in KNN (e.g., the number of neighbors) due to the highest MRR values when selecting this value. We do not show the detailed process because of space limitations.

TABLE IV. RESULT OF MRR

γ	Eclipse	Mozilla	Netbeans Java
0.1	59.62	62.19	58.58
0.2	62.95	69.98	**67.45**
0.3	**75.61**	**74.84**	60.94
0.4	75.11	73.75	58.01
0.5	61.63	68.19	53.30
0.6	60.80	60.05	48.82
0.7	57.90	54.12	45.36
0.8	54.04	51.58	40.13
0.9	51.80	46.72	37.87

C. Result

1) Accuracy

We implemented the proposed approach with evaluation metrics that included precision, recall, and F-measure of our multi-tasks for the test data sets in each project (as shown in Table II). First, Figure 5 denotes how many developers were recommended correctly. The X axis (e.g., T) denotes the TOP x, and the Y axis stands for the result of the F-measure. The 'TOP x' column is rank matching for x (x is 1 to 10). For example in 'TOP 7', we deemed rank matching as 1 to 7.

Next, we implemented the evaluation metrics of our approach to predict the severity of the bugs, as shown in Figure 6. The results provide a balanced accuracy of F-measure. Both tasks provided more than the average F-measure of 70%.

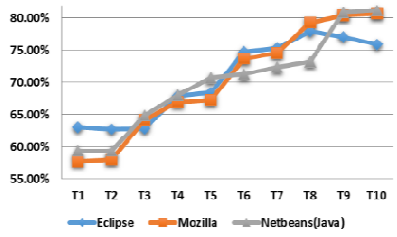

Figure 5. Accuracy of developer recommendation

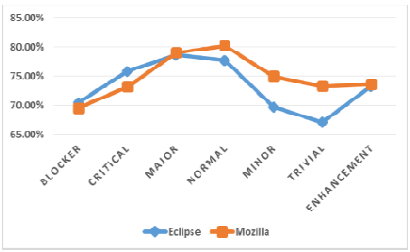

Figure 6. Accuracy of severity prediction

Answer to RQ1 and RQ2. Our approach is effective for the bug triage and severity prediction.

2) Effectiveness

To answer RQ3 and RQ4, in our tasks, we compared our method and our baselines, including DRETOM, ActivityRE and Out-Degree. In addition, Naïve Bayes and single KNN were compared with our approach in severity prediction. Figure 7 and Figure 8 present the comparison results with our baselines. Our approach outperforms other studies, namely DRETOM, ActivityRE, Out-Degree, Naïve Bayes, and single KNN.

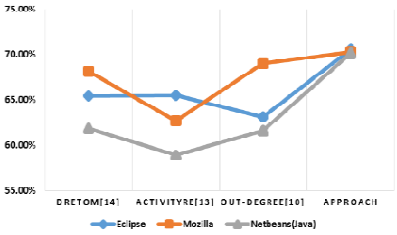

Figure 7. Comparison results in bug triage

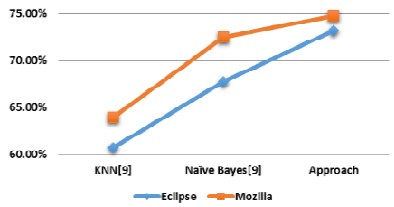

Figure 8. Comparison results in severity prediction

To give a supplement for answering RQ3 and RQ4, we formulated the following five null hypotheses.

$\boldsymbol{H1_0}$, $\boldsymbol{H2_0}$, $\boldsymbol{H3_0}$, $\boldsymbol{H4_0}$ and $\boldsymbol{H5_0}$: Our approach shows no acceptability difference against DRETOM, ActivityRE, Out-Degree, Naïve Bayes, and single KNN, respectively.

The corresponding alternative hypotheses are:

$\boldsymbol{H1_a}$, $\boldsymbol{H2_a}$, $\boldsymbol{H3_a}$, $\boldsymbol{H4_a}$ and $\boldsymbol{H5_a}$: Our approach proves more acceptable than DRETOM, ActivityRE, Out-Degree, Naïve Bayes, and single KNN, respectively.

We tested these hypotheses. At first, we took the F-measure of our implementation, which includes the proposed approach and our baselines. Next, we checked the test for normality [25] of each hypothesis. We not only used the Wilcoxon signed-rank test [23], but we also utilized a t-test [24] by using R language [20]. If the normality value of the statistical test was smaller than 0.05 (95% confidence interval), we used the former. Otherwise, we utilized the latter. Table V represents the result of the statistical test in Eclipse. (We also tested other projects, and the results show that we can accept all of the alternative hypotheses.)

We accept null hypothesis $H1_0$ if the result of the t-test (e.g., p-value) is larger than the significance threshold of 0.05. For example in H1, we verify whether or not there is acceptable difference between our approach and DRETOM. The result of the t-test is 0.008931, and we reject the null

hypothesis because the p-value is below the threshold of 0.05. Thus, there is acceptable difference between our approach and DRETOM in Eclipse. Also, we analyze the statistical test for other projects, such as Mozilla and Netbeans Java. The results show that there is acceptable difference between our approach and the others.

TABLE V. RESULT OF STATISTICAL TEST

null hypothesis	p-value	alternative hypothesis
$H1_0$	0.008931	$H1_a$: Accept
$H2_0$	0.001912	$H2_a$: Accept
$H3_0$	0.001734	$H3_a$: Accept
$H4_0$	0.02779	$H4_a$: Accept
$H5_0$	9.453E-05	$H5_a$: Accept

Answer to RQ3 and RQ4. The proposed approach outperforms other studies in two tasks.

3) Impact of Multi-Feature in Automactic Bug Triage

We verified which feature provides the most important impact in the bug triage with MRR. We also used the suitable parameter and the same data sets (as described in Section V Subsection B). Table VI shows the impact degree represented by MRR values of different features in Eclipse and Mozilla.

TABLE VI. RESULT OF THE IMPACT DEGREE OF DIFFERENT FEATURES

Multi-Feature	Eclipse	Mozilla
Product	52.81	53.01
Component	**61.33**	**62.85**
Priority	38.23	38.54
Severity	31.95	33.36
Product+Component	**70.29**	**71.72**
Priority+Severity	49.89	42.48
ALL	**73.38**	**72.59**

We note that a combination of all features is the most important for the bug triage. Also, a combination of "Product" and "Component" is ranked as second when executing developer recommendations. Of all single features, "Component" is the most important. Due to space limitations, we do not show the results of feature impact for severity prediction. We will do so in future work.

Answer to RQ5. The combination of all features plays the most important role in the bug triage.

VI. DISCUSSION

A. Performance Analysis

In our evaluation result, we improved bug triage performance and severity prediction compared to other studies. Some possible reasons are summarized as follows.

- **Semi-automatic Bug Triage**: The related studies used topic model, and utilized a social network for recommending developers. In our study, we not only utilized topic model, but also adopted multi-feature to filter developers to find the potential candidates for bug fixing. Then, we built a social network so that we recommended the appropriate developers. We believe that the combination of topic model and multi-feature can further enhance the effectiveness of bug triage.
- **Semi-automatic Severity Prediction**: In this paper, by extracting historical bug reports that have the same topic and multi-feature, the scope of candidate bug reports is reduced compared to other approaches, such as Naïve Bayes and single KNN. These well-chosen bug reports can perfect our approach for severity prediction.
- **Impact of the Multi-Feature**: We investigated which feature is the most important factor in bug triage. As a result, the "Component" feature is more important than other single features. Moreover, the combination of all features is the most important factor. Therefore, a combination of all features can improve the accuracy of our tasks.

B. Topic Mismaching

In our study, we adopt matched frequency of word tokens to decide the topic(s) which the given bug belongs to. However, the word tokens cannot be matched well, so it fails to find related topics. We investigated this phenomenon, and found the possible reason is bug report quality. Bug reports are written by humans (e.g., a common user or a developer), and is freeform textual content. Some bug reports may have self-memos or fake comments. However, although these reports are not real bug reports, they are classified into a bug category. In order to resolve this problem, we use Stanford coreNLP and the Stanford Topic Modeling Toolbox, which can support the filtering process by using counted terms in a document. In our approach, we use the default setting of the filtering function. Table VII denotes the filtered bug reports. We note that these filtered bug reports are not used in our experiment. Also, we need to look for strong evidence to handle these filtered bug reports.

TABLE VII. FILTERED BUG REPORTS

	Eclipse	Mozilla	Netbeans
# of bug reports	28 (0.28%)	32 (0.32%)	57 (0.57%)

C. Threats to Validity

- **External validity:** We collected experimental data sets (Table II) from three open source projects to evaluate our approach, but we are not sure that our approach is also effective in commercial projects. In the future, we will demonstrate whether our algorithm can be employed. Moreover, we analyzed the evaluation results when selecting the different weight vectors and explained why the proposed method outperformed other previous studies. However, some noise (e.g., fake bug reports) may have affected the results. We plan to reduce noise by checking whether the bug reports are fake or not in the future.
- **Internal validity:** When predicting the severity of the given bug, we compared our approach with other prediction algorithms, such as Naïve Bayes and single KNN. We need to introduce further methods

using machine learning algorithms (e.g., SVM) to certify the effectiveness of the proposed approach. In addition, we recommended the potential developers who can fix the given bugs. However, the developers who are "young" or "new" are not considered to be candidate developer. We focus on how to reduce the fixing cost, in other words, we find the appropriate developers so that we can reduce the developers' workload and the bugs' lifetime. This challenge is one of our future works.

VII. CONCLUSION

In this study, we utilized topic model and multi-feature to implement bug triage and bug severity prediction. This novel method can help us search historical bug reports for those that have strong relevance (e.g., the same topic and multi-feature) with the new bug report. This method first builds a social network among the candidate developers extracted from these bug reports, and verifies the developers' experience so that the most appropriate developer is recommended to fix the bugs. In this method, we adopt smoothed UM and KL divergence to measure the textual similarity between bug reports. We apply KNN to predict the severity status of the new bug report.

In order to demonstrate the effectiveness of the proposed approach, we experimented on the bug reports collected from three large-scale open source projects, including Eclipse, Mozilla, and Netbeans. The evaluation results and statistical analysis show that our approach can effectively execute the bug triage and bug severity prediction.

In the future, we will capture further factors and other machine learning algorithms to implement other tasks (e.g., bug localization/visualization) to effectively support bug fixing. In addition, we plan to implement this method in commercial projects.

ACKNOWLEDGMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(No.2012-0007149) and by Seoul Creative Human Development Program funded by Seoul Metropolitan Government(No.HM120006).

REFERENCES

[1] N. Jalbert and W. Weimer, "Automated Duplicate Detection for Bug Tracking Systems," Proc. of IEEE International Conference on Dependable Systems & Networks, 2008, pp. 52-61.

[2] G. Jeong, S. Kim and T. Zimmermann, "Improving Bug Triage with Bug Tossing Graphs," Proc. of Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2009, pp. 111-120.

[3] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes and P. Baldi, "Mining Eclipse Developer Contributions via Author-Topic Models," Proc of International Workshop on Mining Software Repositories, 2007.

[4] David M. Blei, Andrew Y. Ng and Michael I. Jordan, "Latent Dirichlet Allocation," Journal of Machine Learning Research, Vol. 3, pp. 993-1022, 2003.

[5] I. X. Chen, C. Z. Yang, T. K. Lu and H. Jaygarl, "Implicit Social Network Model for Predicting and Tracking the Location of Faults," Proc of International Computer Software and Applications Conference, 2008, pp. 136-143.

[6] S. Rao and A. Kak, "Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models," Proc of Working Conference on Mining Software Repositories, 2011, pp. 43-52.

[7] P. Castells, M. FERNÁNDEZ and D. Vallet, "An Adaptation of the Vector-Space Model for Ontology-Based Information Retrieval," IEEE Transactions on Knowledge and Data Engineering, Vol. 19, No. 2, pp. 261-272, 2007.

[8] T. Zhang and B. Lee, "A Hybrid Bug Triage Algorithm for Developer Recommendation," Proc. of Annual ACM Symposium on Applied Computing, 2013, pp. 1088-1094.

[9] A. Lamkanfi, S. Demeyer, Q. D. Soetens and T. Verdonck, "Comparing Mining Algorithms for Predicting the Severity of a Reported Bug," Proc. of European Conference on Software Maintenance and Reengineering, 2011, pp. 249-258.

[10] W. Wu, W. Zhang, Y. Yang, Q. Wang, "DREX: Developer Recommendation with K-Nearest-Neighbor Search and Expertise Ranking," Proc. of Asia-Pacific Software Engineering Conference, 2011, pp. 389-396.

[11] J. Xuan, H. Jiang, Z. Ren and W. Zou, "Developer Prioritization in Bug Repositories," Proc. of International Conference on Software, 2012, pp. 25-35.

[12] J. Park, M. Lee, J. Kim, S. Hwang and S. Kim, "COSTRIAGE: A Cost-Aware Triage Algorithm for Bug Reporting Systems," Proc. of AAAI Conference on Artificial Intelligence, 2011, pp. 139-144.

[13] H. Naguib, N. Narayan, B. Br¨ugge and D. Helal, "Bug Report Assignee Recommendation using Activity Profiles," Proc. of Working Conference on Mining Software Repositories, 2013, pp. 22-30.

[14] X. Xie, W. Zhang, Y. Yang and Q. Wang, "DRETOM: Developer Recommendation based on Topic Models for Bug Resolution," Proc. of International Conference on Predictive Models in Software Engineering, 2012, pp. 19-28.

[15] T. Menzies and A. Marcus, "Automated Severity Assessment of Software Defect Reports," Proc. of IEEE International Conference on Software Maintenance, 2008, pp. 346-355.

[16] A. Lamkanfi, S. Demeyer, E. Giger and B. Goethals, "Predicting the Severity of a Reported Bug," Proc. of IEEE Working Conference on Mining Software Repositories, 2010, pp. 1-10.

[17] Y. Tian, D. Lo and C. Sun, "Information Retrieval Based Nearest Neighbor Classification for Fine-Grained Bug Severity Prediction," Proc. of Working Conference on Reverse Engineering, 2012, pp. 215-224.

[18] P. Bhattacharya, M. Iliofotou, I. Neamtiu and M. Faloutsos, "Graph-Based Analysis and Prediction for Software Evolution," Proc. of International Conference on Software Engineering, 2012, pp. 419-429.

[19] P. Hooimeijer and W. Weimer, "Modeling bug report quality," Proc. of IEEE/ACM International Conference on Automated Software Engineering, 2007, pp. 34-43.

[20] R Core Team, "R: A Language and Environment for Statistical Computing," R Foundation for Statistical Computing, 2012.

[21] C. Goutte and E. Gaussier, "A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation," Lecture Notes in Computer Science, vol. 3408, pp. 345-359, 2005.

[22] J. Zhou, H. Zhang and D. Lo, "Where Should the Bugs Be Fixed? More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports," Proc. of International Conference on Software Engineering, 2012, pp. 14-24.

[23] F. Wilcoxon, "Individual comparisons by ranking methods," Biometrics Bulletin, Vol. 1, No. 6, pp. 80-83, 1945.

[24] The T-Test, Research Methods Knowledge Base, http://www.socialresearchmethods.net/kb/stat_t.php.

[25] Shapiro–Wilk test,WIKIPEDIA, http://en.wikipedia.org/wiki/Shapiro-Wilk_test.