The Journal of Systems and Software 117 (2016) 166–184

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Towards more accurate severity prediction and fixer recommendation of software bugs☆

Tao Zhang[a,b], Jiachi Chen[b], Geunseok Yang[c], Byungjeong Lee[c], Xiapu Luo[b,*]

[a] School of Software, Nanjing University of Posts and Telecommunications, Nanjing 210-023, China
[b] Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China
[c] Department of Computer Science, University of Seoul, Seoul 130-743, South Korea

ARTICLE INFO

Article history:
Received 2 July 2015
Revised 20 February 2016
Accepted 23 February 2016
Available online 4 March 2016

Keywords:
Severity prediction
Fixer recommendation
Topic model

ABSTRACT

Due to the unavoidable bugs appearing in the most of the software systems, bug resolution has become one of the most important activities in software maintenance. For large-scale software programs, developers usually depend on bug reports to fix the given bugs. When a new bug is reported, a triager has to complete two important tasks that include severity identification and fixer assignment. The purpose of severity identification is to decide how quickly the bug report should be addressed while fixer assignment means that the new bug needs to be assigned to an appropriate developer for fixing. However, a large number of bug reports submitted every day increase triagers' workload, thus leading to the reduction in the accuracy of severity identification and fixer assignment. Therefore it is necessary to develop an automatic approach to perform severity prediction and fixer recommendation instead of manual work. This article proposes a more accurate approach to accomplish the goal. We firstly utilize modified *REP* algorithm (*i.e.*, REP_{topic}) and K-Nearest Neighbor (KNN) classification to search the historical bug reports that are similar to a new bug. Next, we extract their features (*e.g.*, assignees and similarity) to develop the severity prediction and fixer recommendation algorithms. Finally, by adopting the proposed algorithms, we achieve severity prediction and semi-automatic fixer recommendation on five popular open source projects, including GNU Compiler Collection (GCC), OpenOffice, Eclipse, NetBeans, and Mozilla. The results demonstrated that our method can improve the performance of severity prediction and fixer recommendation through comparison with the cutting-edge studies.

1. Introduction

Bug resolution is an important activity in software maintenance process. Recent years, due to the increased scale and complexity of software projects, a large number of bugs appear in the development process and hence bug resolution has become a difficult and challenging work (Xia et al., 2013). To effectively track and manage these bugs, open source software projects and many commercial projects adopt bug tracking systems (*e.g.*, Buzilla[1], JIRA[2], etc.) to maintain the huge information about the reported bugs. As an important component of bug tracking systems, bug repositories record a large number of bug reports, which are written by users or developers.

In software maintenance process, developers rely on bug reports stored in bug repositories to fix the given bugs. Once a new bug report is submitted, a triager who is responsible for managing bug reports can read this report to understand the details of the given bug, then verify whether the labelled severity level is correct or not. This process is called "**severity identification**". Severity levels include high-severity (*e.g.*, 'blocker', 'critical', 'major') that represents critical errors and low-severity (*e.g.*, 'minor', 'trivial') that denotes unimportant bugs (Lamkanfi et al., 2010). The following task of the triager is to assign the reported bug to an appropriate developer for executing bug resolution according to its severity level. This process is called "**fixer assignment**" (Servant and Jones, 2012) or bug assignment (Wu et al., 2011; Xia et al., 2015a). Severity identification and fixer assignment are two major tasks of triagers, whose success can affect the time of bug fixing (Yang et al., 2014). Specifically, there are two important issues in the execution process of these tasks as follows:

☆ A preliminary edition of this article was accepted by COMPSAC 2014 as a research full paper. This article extends and provides further experimental evidence of the proposed method.

* Corresponding author.
E-mail address: csxluo@comp.polyu.edu.hk (X. Luo).

[1] https://www.bugzilla.org/.
[2] https://www.atlassian.com/software/jira.

http://dx.doi.org/10.1016/j.jss.2016.02.034
0164-1212/

- **Triagers' workload**: Everyday, a large number of bug reports are submitted to bug repositories. For example, Mozilla bug repository receives an average of 135 new bug reports each day (Liu et al., 2013). Obviously, processing multitude of bug reports places a heavy burden on triagers.
- **Inaccurate severity identification and fixer assignment**: Manual severity identification and fixer assignment may lead to errors, especially on a vast number of bug reports. For example, a critical bug is labelled as a 'low-severity' bug and therefore the fixing time is extended (Menzies and Marcus, 2008). As another example, the triager may assign the improper developer to execute the task of bug fixing, thus leading to the bug reassignments. Jeong et al. (2009) has shown that the more the number of reassignments is, the lower the success probability of bug fixing is.

To resolve the above problems, existing studies tried to perform severity prediction (Menzies and Marcus, 2008; Lamkanfi et al., 2010, 2011; Tian et al., 2012; Yang et al., 2012) and semi-automatic fixer recommendation (Čubranić and Murphy, 2004; Anvik et al., 2006; Matter et al., 2009; Wu et al., 2011; Xuan et al., 2012; Park et al., 2011; Xie et al., 2012; Zhang and Lee, 2013; Naguib et al., 2013; Xuan et al., 2015; Xia et al., 2015b). Machine learning and information retrieval techniques were utilized to realize the goal. However, the major challenge is to find the close relationship between the new bug report (*i.e.*, query) and historical bug reports. In other words, the returned historical bug reports that are similar to the query and their features decide the accuracy of severity prediction and semi-automatic fixer recommendation. Topic modelling (Ramage et al., 2009) is a useful approach to cluster bug reports into corresponding categories. The bug reports in the same category share the same topic. By using topic model, we can find the topic(s) that each bug report belongs to. We introduce these topics as the additional feature of the *REP* algorithm, which is a similarity function proposed by Tian et al. (2012) to calculate the similarity between bug reports. Based on this enhanced *REP* (*i.e.*, REP_{topic}), we utilize K-Nearest Neighbor (KNN) to search the historical bug reports similar to the new bug.

In this work, we investigate which features of the similar historical bug reports can affect the accuracy of the severity prediction and fixer recommendation. Then, we use them, such as similarity between bug reports and developers' experience, to develop the more accurate methods for implementing severity prediction and semi-automatic fixer recommendation. To demonstrate the effectiveness of the proposed approach, we conduct experiments on five open source repositories, including GNU Compiler Collection (GCC), OpenOffice, Eclipse, NetBeans, and Mozilla. The results show that the proposed approach outperforms the cutting-edge approaches on severity prediction and semi-automatic fixer recommendation. Moreover, we also demonstrate that the proposed similarity measure REP_{topic} can improve the accuracy of our approach than other similarity metrics such as *REP* and *cosine similarity*.

To help researchers reproduce our work, we open all source code and datasets at https://github.com/ProgrammerCJC/SPFR.

We summarize the major contributions of our work as follows:

- By utilizing topic modelling, we find the topic(s) to which each bug report belongs. Then, we introduce these topics to enhance the similarity function *REP*, and adopt KNN to search the top-K historical bug reports that are similar to the new bug.
- Based on the features (*e.g.*, textural similarity and developers' experience) extracted from top-K nearest neighbours of the new bug report, we develop new algorithms to improve the accuracy of severity prediction and fixer recommendation.
- We conduct the experiments on five large-scale open source projects, including GCC, OpenOffice, Eclipse, NetBeans, and Mozilla. The results demonstrate that the proposed approach has better performance than the cutting-edge studies.

The remainder of the article is structured as follows: Section 2 introduces background knowledge and the motivations of our work. Section 3 details how to utilize the proposed approach to implement severity prediction and fixer recommendation. In Section 4, we show how to organize the experiments and indicate the experimental results. We discuss the performance of our approach and present some threats to validity in Section 5. Section 6 introduces the related works and shows the differences from our work. In Section 7, we conclude this paper and introduce the future work.

2. Background knowledge and motivation

In our work, we propose an approach to predict the severity levels and recommend the appropriate fixers based on the similar historical bug reports and their features. Thus, in this section, we introduce some background knowledges concerning bug reporting, two tasks in bug resolution, topic modelling, similarity function, and social network-based developers' relationship. Moreover, we present the motivation of our study.

2.1. Bug reporting

Bug reports are software artifacts that track the defects of software projects. Since they provide the detailed description information about the reported bugs, developers utilized these defect details to fix the corresponding bugs.

For example, Fig. 1 shows an Eclipse bug report-*Bug 463360* that contains all basic elements, such as `summary`, `description`, `comments`, `attachment`, `importance`, `reporter`, `assignee` (*i.e.*, fixer), and multiple features such as `component` and `product`. Among them, `summary` is a brief description of a bug; `description` shows the detailed information of the bug; `comments` indicate the free discussion about the reported bug; `attachment` includes one or more than one supporting materials such as patch and test cases; `importance` includes priority level (*e.g.*, P3) and severity level (*e.g.*, normal) of the reported bug; `reporter` is a developer or user who reported the bug; `assignee` is a developer who was assigned to fix the given bug; `component` indicates which component was affected by the bug; and `product` shows which product was influenced by the bug.

In our work, we introduce `summary`, `description`, `component`, `product`, and topics produced by topic modelling to calculate the similarities between the bug reports so that we can find the top-K nearest neighbours of the new bug to execute the severity prediction and fixer recommendation algorithms.

2.2. Two tasks in bug resolution

When a new bug is reported, the developers in the software development program work together for resolving the given bug. Fig. 2 shows its general life cycle in Bugzilla.

The initial state of the new bug report is "Unconfirmed". When the bug report is verified by a triager, the status is changed to "New". In this process, the triager verifies whether the labelled severity level is correct or not (*i.e.*, severity identification). Then the triager is responsible to assign the bug report to an appropriate assignee (*i.e.*, fixer). At this time, the state of the bug report is changed to "Assigned". If the assignee completes the bug-fixing task, the state is changed to "Resolved"; otherwise, the bug is marked as "New" and the report is re-triaged. This process is

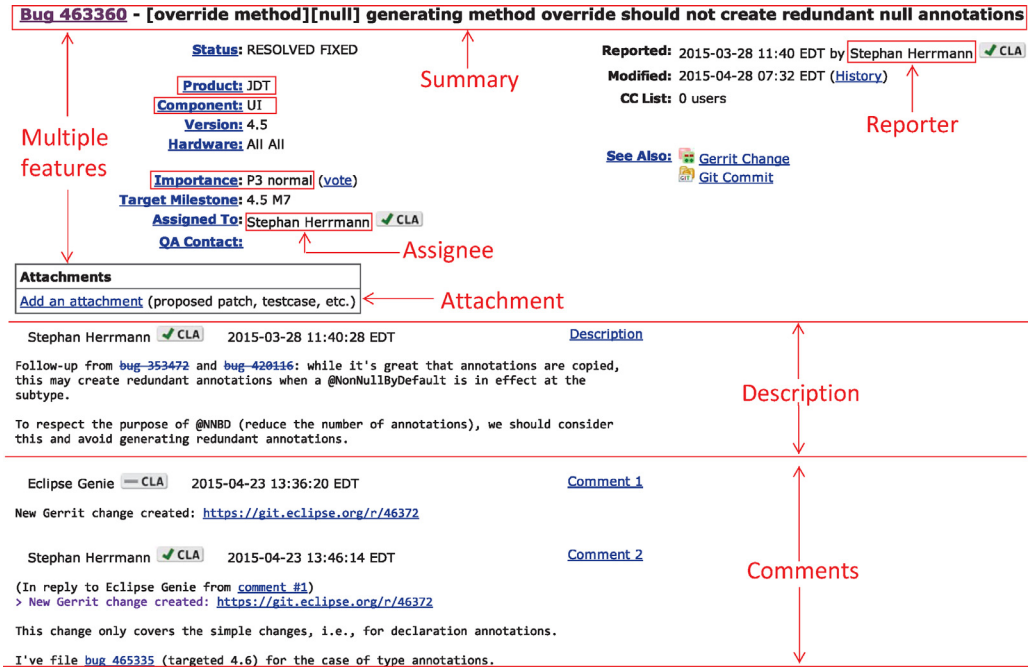

Fig. 1. An example of Eclipse bug report 463360.

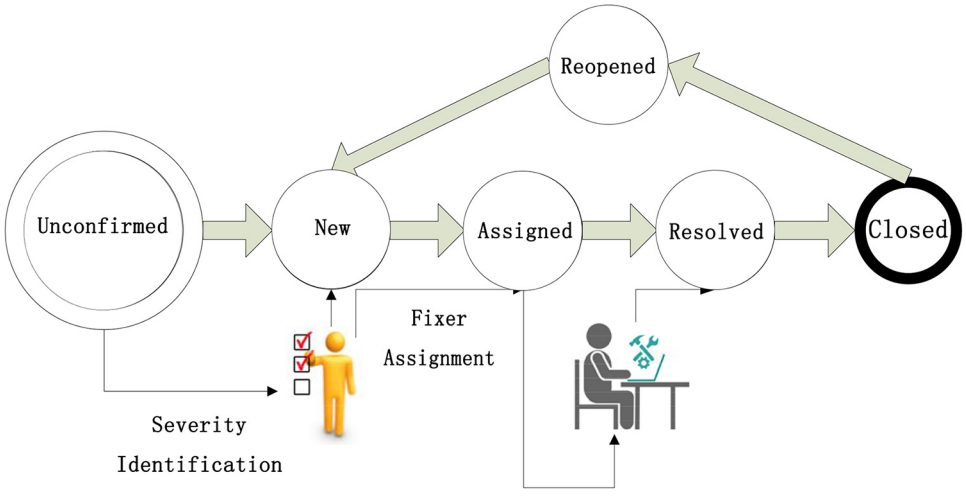

Fig. 2. The life cycle of bug resolution process in Bugzilla.

called bug reassignment. Once the bug report is fixed successfully, the task is finished and the state becomes "Closed". Afterwards, if a developer finds that the bug is not fixed in its entirety, this bug can be reopened. The bug-fixing task is re-executed in a step-wise manner through a cycle-regulated process as described above.

Since the severity identification and fixer assignment are two important tasks for triagers, in our work, we focus on developing a new approach to perform severity prediction and semi-automatic fixer recommendation.

2.3. Topic modelling

As the statistical models, topic models can discover the 'topics' from the collection of documents (Blei and Lafferty, 2007). Each topic includes the topic terms which appear in the documents, and each document may belong to one or more topics. For the bug reports which share the same topic, their textual contents are similar. Therefore, topics can be treated as a useful feature to verify the similarity between bug reports (Xie et al., 2012). In our study, we utilized Latent Dirichlet Allocation (LDA)(Chemudugunta and Steyvers, 2007) to extract the topic distribution of bug reports.

LDA is a general topic model. In LDA, each document is viewed as a mixture of various topics with different probabilities. Each topic is characterized by a distribution of words that frequently co-occur in the documents. Hence, LDA is able to find the bug reports with the same topic(s) that describe the similar contents. In our work, we use LDA to extract the topics with terms from historical bug reports so that we can get the bug reports which are similar to a coming bug.

The Stanford Topic Modelling Toolbox (TMT)[3] (Xie et al., 2012) brings topic modelling tools to perform the analysis on provided datasets. It can help us implement LDA so that we can get the topic distributions of given bug reports. In TMT, there are four parameters (N, R, α, β) that need to be set. N stands for the number of topics; R denotes the number of iterations; α and β are

[3] http://nlp.stanford.edu/software/tmt/tmt-0.4/.

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. We adopted TMT to extract the topics of bug reports as one of the input features of the similarity measure REP_{topic}, which is described in Section 3.

2.4. Similarity function: BM25F and its extension BM25F$_{ext}$

BM25F is a similarity function which is suitable for performing structured information retrieval, thus it can be used to measure the similarity between two bug reports because each bug report is a structured document which includes two textual fields such as `summary` and `description`. Similar to Vector Space Model (VSM) (Castells et al., 2007), BM25F is also represented as *TF∗IDF* model, but it presents the different form. In detail, IDF is the inverse document frequency defined as follows:

$$IDF(t) = log\frac{N}{n_t} \tag{1}$$

where N is the total number of documents, and n_t denotes the number of documents containing the term t.

A field-dependent normalized term frequency $TF_D(t, d)$ of a term t which is considered in each field of the document d is defined by the following formula:

$$TF_D(t,d) = \sum_{f=1}^{K} \frac{\omega_f \times o(d[f],t)}{1 - b_f + \frac{b_f \times l_{d[f]}}{\overline{l_f}}} \tag{2}$$

Here, ω_f is a field-dependent weight parameter. The large value of ω_f means higher importance of the corresponding field; $o(d[f], t)$ denotes the number of occurrences of term t in the field f of the document d; $l_{d[f]}$ is the size of the fth field of the document d; $\overline{l_f}$ means the average size of the fth field across all documents in D; and $b_f (0 \le b_f \le 1)$ is a parameter that determines the scaling by field length ($b_f = 1$ corresponds to full length normalization while $b_f = 0$ corresponds to term weight not being normalized by the length).

Based on *TF∗IDF* model described above, given a query q, the *BM25F* algorithm can be presented as follows:

$$BM25F(q,d) = \sum_{t \in q \cap d} IDF(t) \times \frac{TF_D(t,d)}{k_1 + TF_D(t,d)} \tag{3}$$

where t is the shared term occurring in both q and d, and $k_1 (k_1 \ge 0)$ is a parameter tuning the scale of $TF_D(t, d)$.

The above-mentioned *BM25F* algorithm can be utilized for short queries. However, in our work, each query is a new bug report with the long textual content (*i.e.*, the summary and the description). Therefore, we need to consider the term frequencies in queries. In this situation, we adopt another expression of *BM25F*, *i.e.*, $BM25_{ext}$ described in formula (4), as the similarity measure between the new bug report and the historical bug reports.

$$BM25F_{ext}(q,d) = \sum_{t \in q \cap d} IDF(t) \times \frac{TF_D(t,d)}{k_1 + TF_D(t,d)} \times \frac{(k_3+1)TF_Q(t,q)}{k_3 + TF_Q(t,q)} \tag{4}$$

Here, for each common term t appearing in document d and query q, its contribution to the overall $BM25F_{ext}$ contains two components: one is the product of *IDF* and TF_D inherited from *BM25F*; and the other is the local importance of term t in the query q, which is denoted as $\frac{(k_3+1)TF_Q(t,q)}{k_3+TF_Q(t,q)}$. $TF_Q(t, q)$ is calculated by $TF_Q(t,q) = \sum_{f=1}^{K} \omega_f \times o(q[f], t)$, where $o(q[f], t)$ denotes the number of occurrences of term t in the field f of the query q. Note that $TF_Q(t, q)$ is different from $TF_D(t, d)$, and we do not normalize it because we rank the historical bug reports based on their similarities to a single given query. k_3 is used to control the weight of the local importance of term t in q to the overall score. For example, if k_3 is set to 0, then the local importance of t in q contributes no weight so that $BM25F_{ext}$ becomes *BM25F*.

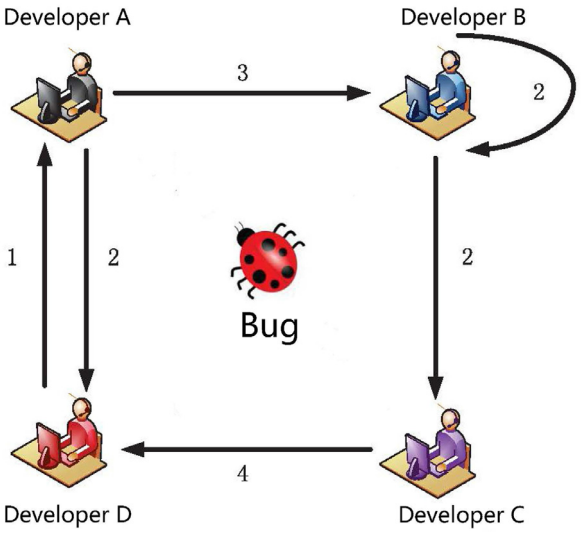

Fig. 3. An example of social network among developers participating bug fixing process.

In our work, we adopt the results of $BM25F_{ext}$ between bug reports as the input features of our similarity algorithm REP_{topic} described in Section 3 to measure the similarity between the new bug report and the historical bug reports.

2.5. Social network-based developers' relationship

Social network reflects a social structure made up of a set of social actors such as individuals and organizations (Kwak et al., 2010). By analysing the social network, we can know the relationship between the actors in a special activity. Based on the characteristics of social network, it can be adopted to analyse the developers' relationship in the bug fixing process. The analysis result can help to develop an accurate semi-automatic fixer recommendation algorithm. Fig. 3 shows an example of a social network among four developers who participate in the bug fixing process. In this figure, the lines represent the commenting activities while the numbers on the lines denote the number of comments. Comments are posted by commenters who participate the discussion to resolve the given bug. For example, the line with the number "2" from Developer A to Developer D means that Developer D comments the bug report(s) assigned to Developer A two times.

By investigating the commenting activities between developers in the bug fixing process, we can understand the developers' experience on fixing the historical bugs. Thus, in our work, social network-based developers' relationship is adopted to design the new method for recommending the appropriate bug fixers.

2.6. Motivation

In terms of the description in Section 1, we note that the large number of submitted bug reports increase developers' workload and lengthen the fixing time.

As an evidence, Table 1 shows the statistical result of average number of assigned bug reports per assignee and average fixing time per bug in five projects. Note that each assignee needs to fix a lot of bugs, especially for NetBeans, average number of assigned bug reports reaches up to 72.6 for each assignee. We cannot directly know the triagers' workload by parsing the XML files (*i.e.*,

Table 1
Statistical result of average workload and fixing time in our data set.

Project	# Bug reports	# Assignees	Average workload Per assignee (number)	Average fixing time Per bug (days)	Period
Eclipse	39,669	771	51.5	411.4	2001/10/10-2014/12/29
NetBeans	19,249	265	72.6	442.5	1999/02/11-2014/12/31
Mozilla	15,501	1022	15.2	244.9	1999/03/17-2014/12/31
OpenOffice	23,402	552	42.4	1129.0	2000/10/21-2014/12/31
GCC	13,301	256	52.0	292.1	1999/08/03-2014/12/01

Table 2
Distribution of the bug reports as the fixing time.

Distribution	Fixing time(year) per project				
Ratio	Eclipse	NetBeans	Mozilla	OpenOffice	GCC
<1%	≥8	≥8	≥6	≥12	≥5

bug reports) downloaded by Eclipse[4], NetBeans[5], Mozilla[6], OpenOffice[7], and GCC[8], however, we can infer that the triagers' workload is more than the assignees' because generally the number of triagers is much less than the number of assignees in open source projects. As a special case, in the early stage, the Eclipse Platform project only had a single bug triager to process all submitted bugs (Anvik et al., 2006). In addition, the fixing time is too long. Even for the shortest one, the average fixing time per bug still achieves to 244.9 days in Mozilla project. To avoid data bias, we omit the small-scale data (*i.e.*, less than 1% of the bug reports) which have the much longer fixing time (See Table 2) in the statistical process of the fixing time. Inaccurate severity identification and bug triage (*i.e.*, severity and fixer reassignment) may lead to longer fixing time. Xia et al. (2014) found that approximately 80% of bug reports have their fields (including severity and fixer field) reassigned. They also demonstrated that these bug reports whose fields get reassigned require more time to be fixed than those without field reassignment. In order to resolve this problem, it is necessary to develop automatic approaches to perform severity prediction and semi-automatic fixer recommendation.

Even though existing studies (Menzies and Marcus, 2008; Lamkanfi et al., 2010, 2011; Tian et al., 2012; Yang et al., 2012; Čubranić and Murphy, 2004; Anvik et al., 2006; Matter et al., 2009; Wu et al., 2011; Xuan et al., 2012; Park et al., 2011; Xie et al., 2012; Zhang and Lee, 2013; Naguib et al., 2013) proposed some approaches to perform severity prediction and semi-automatic fixer recommendation, it is still necessary to improve the accuracy of two tasks. Topic modelling can find the common topics between bug reports, thus using these topics can enhance the previous similarity algorithm-*REP* so that it can perfect the results of KNN classification. We believe that utilizing these bug reports can improve the performance of severity prediction and semi-automatic fixer recommendation. The experimental results shown in Section 4 demonstrate this conclusion.

This article is an extended version of our previous conference paper (Yang et al., 2014) published in COMPSAC 2014. Comparing with the previous study, we extend the contents from the following several aspects:

- We investigate and collect more evidences such as the average workload to enhance the motivation of our work.
- We propose a new similarity metric, *i.e.*, modified *REP* named as REP_{topic}, to compute the similarity between a new bug report and the historical bug reports.
- We propose new methods to conduct the severity prediction and fixer recommendation for improving the results.
- We add two new data sets, *i.e.*, GCC and OpenOffice, in our experiments. Thus, five open source projects are adopted to demonstrate the effectiveness of the proposed approach.
- We implement more previous studies such as (Tian et al., 2012; Xia et al., 2015b) as the baselines for demonstrating the effectiveness of the proposed approach.

3. Methodology for severity prediction and semi-automatic fixer recommendation

In this section, we present the details of our methodology for predicting the severity level of a new bug report and recommending the most appropriate developer to fix the given bug. We first provide a framework of the proposed method, and then we detail each step of the proposed approach.

3.1. Overview

To reduce the developers' workload and overcome the drawbacks of manual severity identification and fixer assignment, we propose an automated approach to perform these two tasks. Fig. 4 shows the framework of the proposed approach.

For our approach, we first conduct the pre-process, mainly includes tokenization, stop word removal, and stemming, to the bug reports collected from the open-source projects. Second, we compute the similarities between a new bug report and the historical bug reports by using the similarity measure REP_{topic}, which combines topics produced by topic modelling, product, component, the textual similarities (only consider the summary and description) between the bug reports by adopting $BM25F_{ext}$. Next, according to the similarities, we can find the top-K nearest neighbours of the new bug report. Finally, for severity prediction, we extract the similarities between the given bug report and these K nearest neighbours to develop the prediction algorithm for recommending the severity level to the new bug. For semi-automatic fixer recommendation, we extract the developers including assignees and commenters from the K nearest neighbours; then we develop a ranking algorithm by analysing the developers' behaviour on bug fixing and commenting activities to recommend the most appropriate bug fixer who has the highest ranking score.

In the following subsections, we introduce the details of each step described in the framework.

[4] https://bugs.eclipse.org/bugs/.
[5] https://netbeans.org/bugzilla/.
[6] https://bugzilla.mozilla.org/.
[7] https://bz.apache.org/ooo/.
[8] https://gcc.gnu.org/bugzilla/.

Table 3
A topic model for NetBeans bug reports when N is set to 30.

	Term@1		Term@2		Term@3		Term@4		Term@5	
Topic@1	Line	0.0285	Type	0.0135	Editor	0.0134	Comment	0.0120	Enter	0.0114
Topic@2	Module	0.0318	Cluster	0.0186	Web	0.0165	Plugin	0.0147	Install	0.0140
Topic@3	System	0.0618	Product	0.0604	Running	0.0598	Client	0.0334	Windows	0.0318
Topic@4	css	0.0270	Tag	0.0210	Error	0.0194	Color	0.0181	Value	0.0143
Topic@5	Module	0.0233	Classpath	0.0214	api	0.0199	Source	0.0185	David	0.0158
Topic@6	Folder	0.0257	Directory	0.0180	Root	0.0164	Path	0.0146	html5	0.0121
Topic@7	Report	0.0325	Exception	0.0314	Duplicates	0.0149	Reporter	0.0123	Exceptions	0.0104
Topic@8	Guest	0.0322	Server	0.0294	64-bit	0.0279	Windows	0.0207	Client	0.0202
Topic@9	Method	0.0277	Completion	0.0227	Public	0.0224	String	0.0156	Return	0.0152
Topic@10	Test	0.0335	Fix	0.0230	Patch	0.0175	Changes	0.0154	Trunk	0.0149

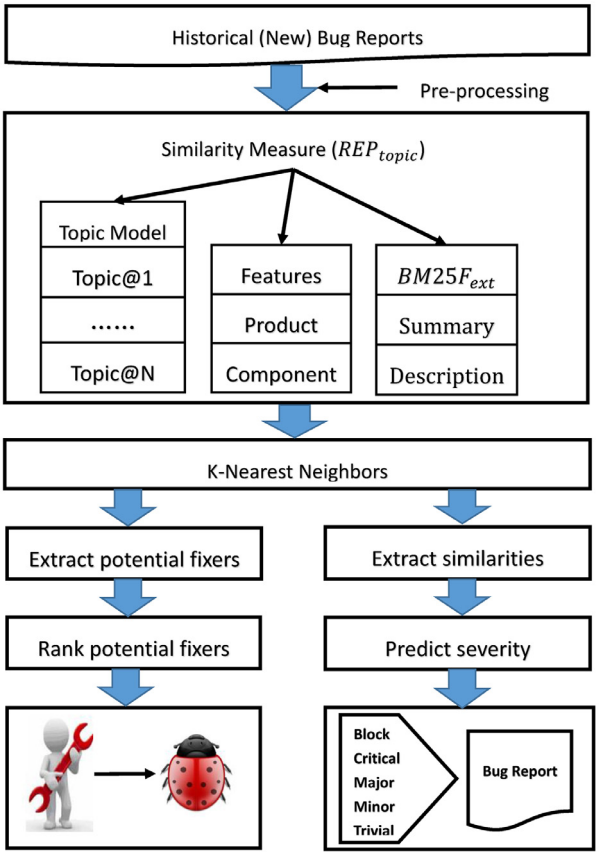

Fig. 4. Framework of severity prediction and semi-automatic bug triage.

3.2. Pre-processing

Once we get the collected bug reports, a pre-processing process is started. This process is implemented by utilizing Natural Language Processing (NLP) techniques, including tokenization, stop word removal, and stemming. For tokenization, each bug report is divided into a series of tokens. As a special case, the variables defined in a program are also split into a concatenation of words. For example, "fillColor" is divided into two words: "fill" and "Color". Next, a set of extraneous terms identified in a list of stop words (*e.g.*, "to", "as", "are", etc.) are filtered out to guarantee the effectiveness of textual similarity measure. Finally, stemming reduces a word to its root form. For example, the words such as "carrying", "carried", and "carries" are changed to "carry".

In our work, we utilize a leading platform called Natural Language Toolkit (NLTK)[9] for executing NLP techniques. This tool provides a lot of useful interfaces with a series of text processing libraries so that it can implement all needed NLP techniques such as tokenization, stop words removal, and stemming.

[9] http://www.nltk.org/.

3.3. Topic modelling

In our work, we adopt TMT (Xie et al., 2012) introduced in Section 2.1 to implement LDA for clustering historical and new bug reports. To utilize TMT, four parameters including N, R, α, β need to be adjusted. We adopt the default values (*i.e.*, 1500, 0.01, and 0.01) for R, α, and β, respectively, and adjust N from 10 to 100 for building the different topic models to perform severity prediction and semi-automatic bug triage. Table 3 lists the top-10 topics for NetBeans bug reports when setting the value of N to 30.

In this table, each topic is represented as topic terms with the probabilities that the terms belong to the corresponding topic. We just list top-5 terms with the highest probabilities in each topic due to the limited space. According to these terms appearing in the bug reports and their probabilities in each topic, we can get the distribution of each bug report in all produced topics by sum all terms' probabilities in each topic. In other words, we can know which topic(s) each bug report belongs to. In our approach, these topics are treated as one of the input features of the similarity measure REP_{topic}.

3.4. Retrieval for similar historical bug reports

To retrieve the historical bug reports which are similar to a new bug report, we need to compute the similarities between them. Tian et al. (2012) proposed *REP* algorithm combining the features, including the component, the product, the textual similarity of two bug reports based on summary an description which are represented by bags of unigrams and bigrams. In our work, we introduce the topics as the additional feature of *REP* to produce a enhanced version, *i.e.*, REP_{topic}, to search the similar historical bug reports with the given bug.

We present all features in the similarity measure REP_{topic} as follows:

$$feature_1(q, br) = BM25F_{ext}(q, br)\ //of\ unigrams \tag{5}$$

$$feature_2(q, br) = BM25F_{ext}(q, br)\ //of\ bigrams \tag{6}$$

$$feature_3(q, br) = \begin{cases} 1 & \text{if q.prod=br.prod} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

$$feature_4(q, br) = \begin{cases} 1 & \text{if q.comp=br.comp} \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

$$feature_5(q, br) = \begin{cases} 1 & \text{if q.topic=br.topic}\ //of\ any\ topic \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

As a special case, a bug report belongs to only one product and only one component, but may belong to more than one topics. In formula (9), we define that the value of $feature_5(q, br)$ is equal to 1, as long as one of the topics that the historical bug report br belongs to is the same as one of the query's topics.

The similarity measure REP_{topic} is a linear combination of five features, with the following formula where ω_i is the weight for the ith feature $feature_i$ defined by formula (5–9). We describe how to tune all parameters in REP_{topic} in Section 4.

$$REP_{topic}(q, br) = \sum_{i=1}^{5} \omega_i \times feature_i \tag{10}$$

By using REP_{topic} to compute the similarities between the new bug report and the historical bug reports, we can rank these historical reports so that the top-K nearest neighbours of the given bug report can be found. Then we utilize them to develop the severity prediction and semi-automatic fixer recommendation algorithms, which are presented as following subsections.

3.5. Severity prediction

When we get the top-K nearest neighbours of a new bug report by using REP_{topic}, we leverage the similarities between the K neighbours and the given bug report to predict the severity level of the new bug. Given a new bug report br_{new}, the probability of its severity level being l is presented as follows:

$$P(br_{new}|l) = \gamma_1 \frac{k_l}{K} + \gamma_2 \frac{\sum_{i=1}^{k_l} sim(br_{new}, br_i)}{\sum_{i=1}^{K} sim(br_{new}, br_i)} \tag{11}$$

where k_l is the number of the br_{new}'s nearest neighbours whose severity level is l while K is the total number of nearest neighbours of br_{new}. In addition, $sim(br_{new}, br_i)$ is the similarity between br_{new} and the nearest neighbor br_i; γ_1, $\gamma_2 \in [0, 1]$ represent the different contribution weights of the two parts to the overall prediction score.

We present our severity prediction approach in Algorithm 1. The parameter adjustment, experimental process and results are described in Section 4.

Algorithm 1 Severity prediction.

Input:
A new bug report br_{new};
The set of top-K nearest neighbours $br_{i \in K}$;
Output:
The most likely severity level of br_{new};
1: Search the number of br_{new}'s neighbours whose severity level is assigned to l_j (j=Block, Critical, Major, Minor, or Trivial).
2: Extract the similarities between br_{new} and its nearest neighbours;
3: Set the weight vectors γ_1 and γ_2 to compute the probability of br_{new}'s severity level being l_j: formula (11);
4: Continue to execute the process until the probabilities of all severity levels are computed.
5: **return** Severity level of br_{new}

3.6. Semi-automatic fixer recommendation

To realize the goal of semi-automatic fixer recommendation, we extract the developers including assignees and commenters from the top-K nearest neighbours of the new bug report as the candidates. In order to capture the candidates' behavior on their previous bug fixing activities, we build a social network described in Section 2.5 to collect the records on commenting activities so that we can quantize each candidate's behavior as following formula:

$$SocialScore(d) = \frac{nc \times od}{MAX_{1 \le i \le M}(nc_i \times od_i)} \tag{12}$$

where nc stands for the number of comments that the developer d posts while od (out-degree) represents the number of comments that the other developers post to the bug reports assigned to d. $MAX_{1 \le i \le M}(nc_i \times od_i)$ is used to normalize the social score via a maximum value among M candidate developers.

Moreover, we consider the number of fixed bug reports and that of reopened bug reports as factors to capture the candidates' experiences in previous bug fixing activities. We get the experience score of the candidate developer d as follows:

$$ExperienceScore(d) = \frac{n_{fix}/n_{reopen}}{MAX_{1 \le i \le M}(n_{fix_i}/n_{reopen_i})} \tag{13}$$

In formula (13), n_{fix} represents the number of bugs fixed by the developer d successfully while n_{reopen} stands for the number of bugs which have the reopened records among all historical bugs assigned to d. $MAX_{1 \le i \le M}(n_{fix_i}/n_{reopen_i})$ is used to normalize the experience score via a maximum value among M candidate developers.

In $SocialScore(d)$ and $ExperienceScore(d)$, we introduce several factors including nc, od, n_{fix}, and n_{reopen}. Hooimeijer and Weimer (2007) demonstrated that the number of comments is a positive coefficient, which means the assigned bug received more developers' attention. More suggestions let it get the high probability of being fixed. Thus we select nc and od as the positive factors in $SocialScore(d)$. For the strength of empirical analysis, the more the number of bugs fixed successfully, the more experience the developer has; by contrast, the increasing number of reopened bugs has a negative impact to the developer who was assigned to fix these bugs (Shihab et al., 2010). Thus, we assign d_{fix}, as the positive factor and d_{reopen} as the negative factor in $ExperienceScore(d)$. We combine $SocialScore(d)$ and $ExperienceScore(d)$ to compute the ranking score for each candidate bug fixer as follows:

$$FRScore(d) = \delta_1 \times SocialScore(d) + \delta_2 \times ExperienceScore(d) \tag{14}$$

where δ_1, $\delta_2 \in [0, 1]$ stand for the different contribution weights of $SocialScore(d)$ and $ExperienceScore(d)$ to the candidate's ranking score.

We summarize the semi-automatic fixer recommendation algorithm via Algorithm 2. In Section 4, we show the parameter tuning process and experimental results.

Algorithm 2 Semi-automatic fixer recommendation.

Input:
A new bug report br_{new};
The set of top-K nearest neighbours $br_{i \in K}$;
Output:
A list of developers ranked by ranking score, $d_1, d_2, \ldots d_n$;
Extracting the assignees and commenters from $br_{i \in K}$ as a set of candidate developers D;
2: Extracting the number of comments to compute $SocialScore(d_j \in D)$: formula (12)
Extracting the number of bugs fixed successfully and the number of reopened bugs assigned to $d_j \in D$ to compute $ExperienceScore(d_j \in D)$: formula (13)
4: Set the weight vectors δ_1 and δ_2 to compute $FRScore(d_j \in D)$: formula (14)
return $d_1, d_2, \ldots d_n$ based on descending order of ranking scores;

3.7. Research questions

Based on the top-K nearest neighbours of a new bug report via REP_{topic} and KNN, we implement severity prediction and semi-automatic fixer recommendation. To evaluate whether the proposed approach can effectively achieve the tasks, we answer the following research questions:

- **RQ1:** ***How effective is our approach to perform severity prediction and semi-automatic fixer recommendation when we choosing the different number of nearest neighbours of the new bug report?***
  By utilizing the proposed approach, we want to know its effectiveness for predicting the bug severity and recommending the bug fixer when changing the number of top-K nearest neighbours of the new bug report.
- **RQ2:** ***How much improvement could the new prediction algorithm gain over the cutting-edge studies such as INSPect*** (Tian et al., 2012) ***and Naive Bayes (NB) Multinomial*** (Lamkanfi et al., 2010, 2011)***?***
  Tian et al. proposed INSPect, which utilized *REP* to search top-K nearest neighbours of the new bug report and used their developed algorithm to predict the severity level (Tian et al., 2012). Lamkanfi et al. demonstrated that NB Multinomial performed better than the other three well-known machine learning algorithms, including NB, KNN, and Support Vector Machines (SVM) (Lamkanfi et al., 2011). To address this research question, we select INSPect and NB Multinomial as the baselines to measure the performance improvement of our approach. Answer to this research question would shed light to whether our approach can produce the better performance than existing state-of-the-art severity prediction algorithms.
- **RQ3:** ***How much improvement could the proposed fixer recommendation algorithm gain over the previous studies, including DRETOM*** (Xie et al., 2012)***, DREX*** (Wu et al., 2011)***, and DevRec*** (Xia et al., 2015b)***?***
  Xie et al. proposed DRETOM which utilized topic modelling to recommend the bug fixers while DREX adopted social network metrics (*e.g.*, out-degree) to implement the same task. DevRec introduced topics, component, and product to develop a fixer recommendation algorithm. In this research question, we want to evaluate the extent to which our approach outperforms these cutting-edge studies. To answer this question, we compare the performance of our approach with those of these algorithms to verify the performance improvement using our approach.
- **RQ4:** ***What is the performance of the*** REP_{topic}***?*** In our work, we develop an enhanced version of REP_{topic} by adding an additional feature, *i.e.*, topics. By using REP_{topic}, we find the top-K nearest neighbours of the new bug report to implement the severity prediction and fixer recommendation. We believe that REP_{topic} is an important part to improve the performance of the proposed approaches. Thus, answer to this research question can help us verify whether REP_{topic} can affect the performance of severity prediction and fixer recommendation.

4. Experiment and result evaluation

In this section, we introduce the experimental process and show the experimental results. Moreover, we compare the performance of our approaches and that of other previous studies.

4.1. Experiment setup

4.1.1. Data set

In order to demonstrate the effectiveness of the proposed approach, we carry out a series of experiments on five large-scale open source bug repositories, including GCC, OpenOffice, Eclipse, NetBeans, and Mozilla. We only collect the fixed bug reports which were denoted by "resolved" or "closed" before December 31, 2014 due to their strong stability and reliability. Note that we do not consider the reports whose severity label is `enhancement` because they technically do not represent real bug reports (Lamkanfi et al., 2010). The overview of our data sets is described in Table 1. We open all data sets at https://github.com/ProgrammerCJC/SPFR.

To expediently evaluate the results, we use the same method described in Xia et al. (2015b) for training-test set validation. First, the bug reports extracted from each bug repository are sorted in chronological order of creation time, and then divided into 11 non-overlapping frames of equal sizes. Table 4 shows the details of our data set. Second, we conduct the training using bug reports in frame 0, and test the bug reports in frame 1. Then, we train using bug reports in frame 0 and 1, and use the similar way to test the bug reports in frame 2. We continue this process until frame 10. In the final round, we train using bug reports in frame 0-9, and test using bug reports in frame 10. Finally, we use the average accuracy across the 10 round validation as the final result.

4.1.2. Evaluation methods

In order to measure the accuracy of our approach and compare the performance with other cutting-edge techniques, we adopt Precision ($\frac{TP}{TP+FP}$), Recall ($\frac{TP}{TP+FN}$), F-measure ($2 \times \frac{Precision \times Recall}{Precision+Recall}$) (Goutte and Gaussier, 2005), and MRR ($\frac{1}{M_q}\sum_{i=1}^{M_p}\frac{1}{R_i}$) (Zhou et al., 2012) to perform the evaluation, where *TP* (*i.e.*, True Positive instances) denotes the number of instances such as severity levels or bug fixers predicted correctly; *FP* (*i.e.*, False Positive instances) is the number of instances predicted incorrectly; *FN* (*i.e.*, False Negative instances) stands for the number of actual instances which are not predicted by our approach; M_q is the total number of queries(i.e., the bug reports in our test set); and R_i represents the rank of the correct result in the recommended list. Note that MRR is unfit for severity prediction because it is useful only for the evaluation on the ranked lists, thus, we only use Precision, Recall, and F-measure to evaluate the performance of severity prediction while we adopt all above-mentioned metrics to evaluate the results of semi-automatic fixer recommendation.

4.1.3. Exclusion criteria

Before we perform the proposed approach and other cutting-edge studies on the data sets describe in Tables 1–4, we select the effective and useful data to conduct the approaches.

For severity prediction, we focus on predicting five severity labels of the bug reports, namely `blocker`, `critical`, `major`, `minor`, and `trivial`. Following the previous work (Lamkanfi et al., 2010; 2011; Tian et al., 2012), we do not consider the severity label `normal`. Because the label `normal` is the default option for selecting the severity when reporting a bug and a lot of developers did not consciously assess the bug severity (Lamkanfi et al., 2010; 2011). In addition, the data imbalance may impact the prediction results. We show the distribution ratio of the bug reports with each severity label to all non-normal bug reports in Table 5. We find that the distributions of bug reports as per label in NetBeans and OpenOffice present higher imbalance than other three data sets. Specifically, in NetBeans, the bug reports labelled by `blocker` occupy 99.07% of total bug reports; in OpenOffice, the bug reports labelled by `trivial` possess 96.62% of total. The imbalanced data may affect the reliability of the results, and thus we do not perform the severity prediction in NetBeans and OpenOffice. Processing the imbalanced data is out of the scope of our work, but it will be examined in the future work.

For semi-automatic fixer recommendation, in each of the five data sets, we remove developers who appear less than 10 times,

Table 4
Details of our data set.

Project	# Bug reports	Average size of each frame	# Products	# Component	Period
Eclipse	39,669	3606	7	114	2001/10/10-2014/12/29
NetBeans	19,249	1750	33	148	1999/02/11-2014/12/31
Mozilla	15,501	1409	12	167	1999/03/17-2014/12/31
OpenOffice	23,402	2127	36	116	2000/10/21-2014/12/31
GCC	13,301	1209	2	27	1999/08/03-2014/12/01

Table 5
Distribution of the non-normal bug reports as the different severity labels.

Project	Distribution ratio as each non-normal severity label				
	Blocker	Critical	Major	Minor	Trivial
Eclipse	682(9.40%)	1,412(19.45%)	2,934(40.42%)	1,389(19.14%)	841(11.59%)
NetBeans	**13,812(99.07%)**	13(0.09%)	70(0.50%)	35(0.25%)	12(0.09%)
Mozilla	270(10.67%)	437(17.26%)	672(26.54%)	696(27.49%)	457(18.05%)
OpenOffice	51(0.23%)	158(0.70%)	518(2.30%)	33(0.15%)	**21,730(96.62%)**
GCC	221(8.21%)	1,656(61.52%)	257(9.55%)	479(17.79%)	79(2.93%)

Table 6
Parameters in REP_{topic}.

Parameter	Description	Init.	Selected parameter values per project				
			Eclipse	NetBeans	Mozilla	OpenOffice	GCC
N	The number of topics	10	30	30	30	30	30
ω_1	Weight of $feature_1$ (unigram)	0.9	1.264	1.226	1.159	1.163	1.175
ω_2	Weight of $feature_2$ (bigram)	0.2	0.413	0.313	0.034	0.013	0.124
ω_3	Weight of $feature_3$ (product)	2	2.285	2.775	2.198	2.285	2.322
ω_4	Weight of $feature_4$ (component)	0	0.232	0.535	0.041	0.032	0.039
ω_5	Weight of $feature_5$ (topics)	0	1.001	1.031	0.988	1.013	1.074
$\omega_{summ}^{unigram}$	Weight of `summary` in $feature_1$	3	3.128	3.814	3.014	2.980	2.994
$\omega_{desc}^{unigram}$	Weight of `description` in $feature_1$	1	1.287	1.481	0.764	0.287	0.233
$b_{summ}^{unigram}$	b of `summary` in $feature_1$	0.5	0.516	0.537	0.499	0.501	0.499
$b_{desc}^{unigram}$	b of `description` in $feature_1$	1	1.178	1.069	1.003	1.012	1.004
$k_1^{unigram}$	k_1 in $feature_1$	2	2.000	2.000	2.000	2.000	2.000
$k_3^{unigram}$	k_3 in $feature_1$	0	0.331	0.523	0.031	0.003	0.023
ω_{summ}^{bigram}	Weight of `summary` in $feature_2$	3	3.157	3.357	2.971	3.001	2.887
ω_{desc}^{bigram}	Weight of `description` in $feature_2$	1	1.194	1.084	1.003	1.004	1.212
b_{summ}^{bigram}	b of `summary` in $feature_2$	0.5	0.524	0.613	0.503	0.499	0.484
b_{desc}^{bigram}	b of `description` in $feature_2$	1	1.015	1.122	0.969	0.998	1.021
k_1^{bigram}	k_1 in $feature_2$	2	2.000	2.000	2.000	2.000	2.000
k_3^{bigram}	k_3 in $feature_2$	0	0.015	0.103	0.154	0.001	0.058

because they are not active and recommending these candidate fixers does not help much in bug resolution. Moreover, we delete the terms that appear less than 20 times according to the similar data filtering method described in Xia et al. (2015b).

4.2. Parameter tuning

Our approach involves parameter tuning. As the description in Section 3, our approach includes two phases: searching the historical bug reports that are similar to the new bug, and implementing two bug resolution tasks (namely severity prediction and semi-automatic fixer recommendation). In the first phase, we should adjust the parameters in the similarity measure REP_{topic} and KNN; in the second phase, we should tune the parameters used in the severity prediction and semi-automatic fixer recommendation algorithms. We describe how to adjust these parameters as following paragraphs.

In the first phase of our approach, we add the new feature *topics* to enhance the original *REP* (Tian et al., 2012) using topic modelling. Thus we should first adjust the parameters used in TMT, which is a topic modelling tool introduced in Section 3.3. We set the parameters, including R, α, and β, to their default values (*i.e.*, 1500, 0.01, 0.01) respectively, and adjust the number of topics N from 10 to 100, with an interval of 10.

The similar measure REP_{topic} defined in formula (10) has 17 free parameters in total. For $feature_1$ and $feature_2$, we compute textual similarities of q and br over two fields: `summary` and `description` by using *BM25F*. Computing each of two features needs $(2 + 2 \times 2) = 6$ free parameters. In addition, in formula (10), there are 5 weight factors for the corresponding 5 features. Thus, REP_{topic} requires $(2 \times 6 + 5) = 17$ parameters to be set.

Table 6 shows the parameters of REP_{topic} in column 1 and 2. We follow the same parameter tuning method (*i.e.*, gradient descent) used in Sun et al. (2011); Tian et al. (2012) to verify the values in REP_{topic}. Specifically, when the number of topics is initialized (*e.g.*, N=10), we start to adjust all 17 parameters in REP_{topic} using gradient descent. Given each of these parameters x, we initialize it with a default value recommended by Sun et al. (2011), which is described in the third column of Table 6. Then we run the iterative adjustment of the value of x so that the value of the RankNet cost function *RNC* (Taylor et al., 2006; Burges et al., 2005) reaches the minimum. *RNC* is defined by $RNC(I) = log(1 + e^Y)$ where I denotes a training instance, and Y is presented as $Y = sim(br_{irr}, q) - sim(br_{rel}, q)$. Here, br_{irr} is an irrelevant bug report with a query q (*i.e.*, a new bug report) while br_{rel} means a relevant bug report with q. In Tian et al. (2012), Tian et al. regard br_{rel} as the duplicate bug reports of the new bug report and br_{irr} as the non-duplicate bug reports. We also adopt the same way to compute *RNC*. For the

Table 7
Parameters in the severity prediction algorithm.

Severity label	Parameter	Selected parameter values per project		
		Eclipse	Mozilla	GCC
Blocker	γ_1	0.36	0.31	0.40
	γ_2	0.82	0.31	0.02
Critical	γ_1	0.46	0.90	0.04
	γ_2	0.48	0.90	0.58
Major	γ_1	0.61	0.71	0.72
	γ_2	0.17	0.72	0.74
Minor	γ_1	0.63	0.69	0.19
	γ_2	0.13	0.67	0.59
Trivial	γ_1	0.33	0.70	0.91
	γ_2	0.29	0.71	0.81

Table 8
Parameters in the semi-automatic fixer recommendation algorithm.

Parameter	Selected parameter values per project				
	Eclipse	NetBeans	Mozilla	OpenOffice	GCC
δ_1	0.12	0.32	0.11	0.78	0.54
δ_2	0.74	0.39	0.81	0.09	0.61

details of the iterative adjustment using gradient descent, please refer to (Sun et al., 2011). We list the parameter values selected for performing REP_{topic} in the columns 4–8 of Table 6.

In the second phase of our approach, we adjust the parameters appearing in the severity prediction and semi-automatic fixer recommendation algorithms. In each of the two algorithms, there are a pair of weight vectors need to be adjusted. For severity prediction algorithm described in formula (11) and Algorithm 1, the parameters γ_1 and γ_2 are used to adjust the weights of the different parts in formula (11); for semi-automatic fixer recommendation described in formula (14) and Algorithm 2, the parameters δ_1 and δ_2 are introduced to present the different weights of *SocialScore* and *ExperienceScore*, respectively, to the candidate's ranking score.

We adopt the similar adjustment method proposed by Xia et al. (2015b) because we also adopt the similar way to introduce the different weight factors to control the contributions of the various partial scores to the overall score. This method is a sample-based greedy method. Due to the large size of the bug reports in data sets, we randomly sample a small subset (10%) of the number of bug reports to produce the good parameter values. Then we use the small-scale data to compute each partial score of the severity prediction and semi-automatic fixer recommendation algorithms. We iterate the process of choosing the good values for all weight factors. For each iteration, we first randomly assign a value between 0 to 1 to a weight factor y (*e.g.*, γ_1 or δ_1). Next we fix the value of another weight factor (*e.g.*, γ_2 or δ_2) in the algorithm, and we increase y incrementally by 0.01 at a time and compute the F-measure values. Based on the F-measure values, we can get the best parameter values. For the details of the iterative adjustment using sample-based greedy method, please refer to (Xia et al., 2015b). We list the weight factor values selected for performing the severity prediction and semi-automatic fixer recommendation algorithms in Tables 7 and8, respectively.

The above-mentioned parameter values are used to perform the proposed approach, we show the experimental results as following subsections in order to give the answers of the research questions RQ1-RQ4.

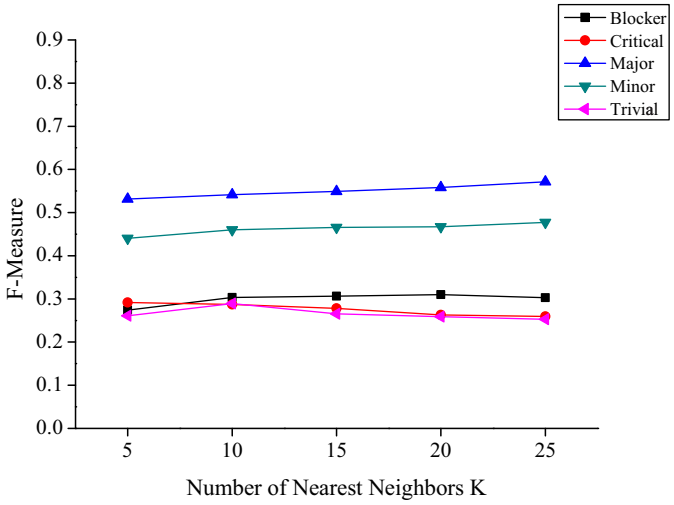

Fig. 5. Eclipse: varying K and its effectiveness on F-measure.

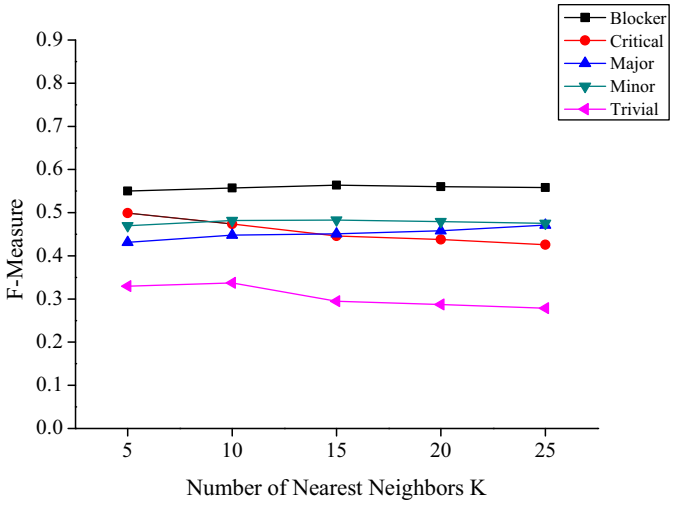

Fig. 6. Mozilla: varying K and its effectiveness on F-measure.

4.3. Answer to RQ1: effectiveness evaluation

To answer the research question RQ1, we perform the proposed approach when choosing the different number of top-K neighbours ($K = 5, 10, 15, 20, 25$) of each query (*i.e.*, new bug report).

First, we show the evaluation results of severity prediction for Eclipse, Mozilla, and GCC datasets in Figs. 5–7, respectively. When we increase *K*, we consider more nearest neighbours. We find an interesting result: the F-measure values do not always increase when increasing the more nearest neighbours.

From the figures, for Eclipse, the F-measure values of `major` and `minor` increase as we increase *K*. However, the F-measure value of `critical` decreases as we increase *K*. In addition, the F-measure values of `blocker` and `trivial` decrease when they reach up to the peak values as we increase *K*. For Mozilla, the F-measure value of `major` slightly increases as we increase *K*. However, the F-measure value of `critical` decreases as we increase *K*. Moreover, the F-measure values of `blocker`, `minor`, and `trivial` achieve to the peak values, then decrease as we increase *K*. For GCC, the F-measure values of `critical` and `major` slightly increase to a peak value, then slightly decrease as we increase *K*. The F-measure values of other three severity labels, including `blocker`, `minor`, and `trivial`, decrease as we increase *K*. The evaluation results of severity prediction indicate that the

Table 9
Effectiveness of semi-automatic fixer recommendation when varying K.

K	# Recommended fixers	F-measure per project				
		Eclipse (%)	NetBeans (%)	Mozilla (%)	OpenOffice (%)	GCC (%)
K = 5	5	**35.01**	**34.48**	**36.22**	**31.7**	**37.24**
	10	**35.42**	**33.78**	**36.93**	**31.47**	**35.42**
K = 10	5	28.54	31.77	30.08	25.14	32.66
	10	29.80	28	30.59	26.58	30.31
K = 15	5	24.57	30.51	26.65	21.03	27.94
	10	26.1	25.57	27.05	23.52	28.78
K = 20	5	20.56	29.27	24.7	18.68	25.34
	10	23.85	24.64	23.76	21.39	27.74
K = 25	5	19.07	28.31	23.71	17.25	23.57
	10	21.17	23.61	21.99	19.59	26.55

Table 10
Performance comparison among severity prediction algorithms when K=5 on P(Precision), R(Recall), and F(F-measure).

Project	Severity	Our approach			INSpect			NB Multinomial		
		P (%)	R (%)	F (%)	P (%)	R (%)	F (%)	P (%)	R (%)	F (%)
Eclipse	Blocker	30.63	24.80	**27.41**	1.84	33.37	3.50	23.76	17.10	19.89
	Critical	28.35	30.10	**29.19**	27.45	26.79	27.12	11.62	22.31	15.28
	Major	59.51	47.97	**53.12**	62.71	45.95	53.04	46.16	42.90	44.47
	Minor	43.23	44.88	**44.04**	46.28	39.64	42.71	32.81	23.21	27.19
	Trivial	18.93	41.87	**26.08**	4.27	47.51	7.83	28.42	17.03	21.30
Mozilla	Blocker	48.15	64.19	**55.02**	46.67	66.55	54.86	39.01	23.91	29.65
	Critical	47.27	52.84	**49.90**	45.82	44.72	45.26	24.09	31.52	27.31
	Major	46.51	40.21	**43.13**	41.90	37.86	39.77	58.57	31.43	40.91
	Minor	50.16	44.20	**46.99**	54.84	41.01	46.92	27.10	26.04	26.56
	Trivial	27.25	41.70	**32.96**	14.00	49.93	21.87	3.25	15.27	5.36
GCC	Blocker	8.00	54.67	13.96	10.41	38.45	**16.39**	12.67	13.07	12.87
	Critical	83.72	77.07	**80.25**	86.22	67.94	76.00	61.04	67.56	64.14
	Major	42.08	25.95	**32.10**	37.87	15.20	21.70	5.75	5.80	5.78
	Minor	29.77	47.53	**36.61**	11.72	50.98	19.06	30.11	17.31	21.98
	Trivial	25.71	16.53	**20.12**	0.56	10.00	1.05	1.25	0.28	0.45

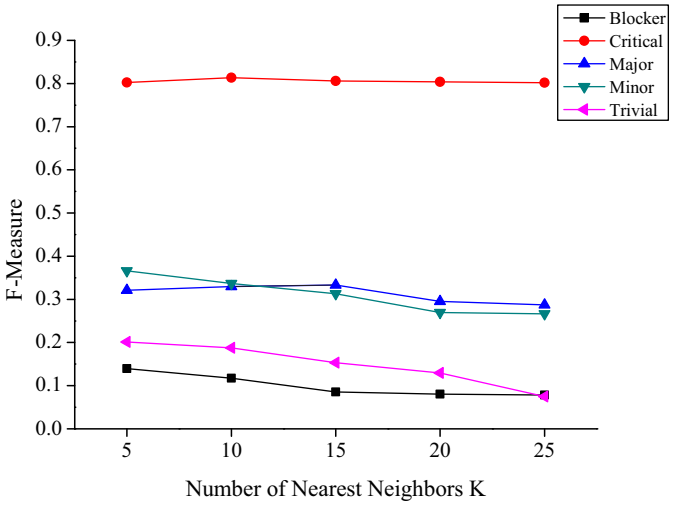

Fig. 7. GCC: varying K and its effectiveness on F-measure.

additional neighbours are not always similar to the target bug report. Thus, adding the irrelevant bug reports as the neighbours can reduce the accuracy of severity prediction in our work.

Next, we show the evaluation results of semi-automatic fixer recommendation for all five datasets in Table 9. In this table, the F-measure values decrease when we increase *K*. For example of Eclipse, when varying *K* from 5 to 25, the F-measure values decrease from 35.01% to 19.07% when recommending top-5 fixers, and decrease from 35.42% to 21.17% when recommending top-10 fixers. The phenomenon indicates that extracting the candidate fixers from irrelevant bug reports can affect the accuracy of the fixer recommendation. Therefore, we think that adding additional neighbours of the new bug report cannot give more help to the performance of semi-automatic fixer recommendation.

According to the evaluation results of severity prediction and semi-automatic fixer recommendation, we can answer RQ1 as follows:

Answer to RQ1: For severity prediction, adding the number of nearest neighbours of the new bug report can improve the prediction effectiveness of partial severity labels; for semi-automatic fixer recommendation, adding the number of nearest neighbours cannot improve the effectiveness of the recommendation.

4.4. Answer to RQ2: performance comparison among severity prediction methods

To answer the research questions RQ2, we compare the performance of our approach with previous cutting-edge studies. We select INSpect (Tian et al., 2012) and NB Multinomial (Lamkanfi et al., 2010; 2011) as the baselines to conduct the performance comparison. Table 10 shows the results of performance comparison when using the top-5 nearest neighbours.

We now analyse the comparison results shown in Table 10 as follows:

(1) For Eclipse, we can predict the `blocker`, `critical`, `major`, `minor`, and `trivial` severity labels by F-measure values of 27.41%, 29.19%, 53.12%, 44.04%, and 26.08%, respectively. The F-measure value is very good for `major` severity label but is poorest for `trivial` severity label.

By comparing with the F-measure values of INSPect, we find that the performance of our approach is much better than it when predicting the severity labels `blocker` and `trivial`. When predicting the severity labels `critical` and `minor`, our approach

can improve the F-measure values of 2.07% and 1.33%, respectively. Our approach presents the very close performance with INSPect when predicting `major`.

By comparing with the F-measure values of NB Multinomial, on the one hand, we note that our approach performs better than it when predicting all severity labels.

Thus for Eclipse, our approach performs better than INSPect and NB Multinomial.

(2) For Mozilla, we can predict the `blocker`, `critical`, `major`, `minor`, and `trivial` severity labels by F-measure values of 55.02%, 49.90%, 43.13%, 46.99%, and 32.96%, respectively. The F-measure value is very good for `blocker` severity label but is poorest for `trivial` severity label.

By comparing with the F-measure values of INSPect, we note that our approach performs better than it when predicting the severity labels `blocker`, `critical`, `major`, and `trivial`. Our approach presents the very close performance with INSPect when predicting `minor`.

By comparing with the F-measure values of NB Multinomial, our approach performs much better than it when predicting the severity labels `blocker`, `critical`, `minor`, and `trivial`. When predicting `major`, our approach presents slightly better performance than NB Multinomial.

Therefore, for Mozilla, our severity prediction method performs better than INSPect and NB Multinomial.

(3) For GCC, we can predict the `blocker`, `critical`, `major`, `minor`, and `trivial` severity labels by F-measure values of 13.96%, 80.25%, 32.10%, 36.61%, and 20.12%, respectively. The F-measure value is very good for `critical` severity label but is poorest for `blocker` severity label.

By comparing with the F-measure values of INSPect, we note that our approach performs better than it when predicting the severity labels `critical`, `major`, `minor`, and `trivial`. For the `blocker` label, our approach loses out to INSPect by 2.43%.

By comparing with the F-measure values of NB Multinomial, our approach performs much better than it when predicting the severity labels `critical`, `major`, `minor`, and `trivial`. When predicting `blocker`, our approach shows slightly better performance than NB Multinomial.

In general, for GCC, our severity prediction method performs better than INSPect except `blocker` label, and shows better performance than NB Multinomial.

In order to further evaluate the performance of INSPect, NB Multinomial, and our approach, we adopt the arithmetic mean of the F-measure values for all five severity labels to conduct the performance comparison. The method of arithmetic mean considers the different number of bug reports as each severity label, and thus it can help us get the convincing results. We show the comparison results of severity prediction for Eclipse, Mozilla, and GCC datasets in Figs. 8–10, respectively.

In these figures, the arithmetic mean F-measure values of our approach and INSpect change as we increase K, but NB Multinomial does not change because it does not adopt KNN to implement severity prediction. According to the comparison results, we can clearly know that our approach performs better than INSPect and NB Multinomial.

To further verify whether our approach performs significantly better than INSpect and NB Multinomial, we carry out a statistical test in the R environment (Team, 2014). First, we define the two null hypotheses as follows:

- $H1_0$: Our approach shows no noteworthy difference against INSpect;
- $H2_0$: Our approach shows no noteworthy difference against NB Multinomial

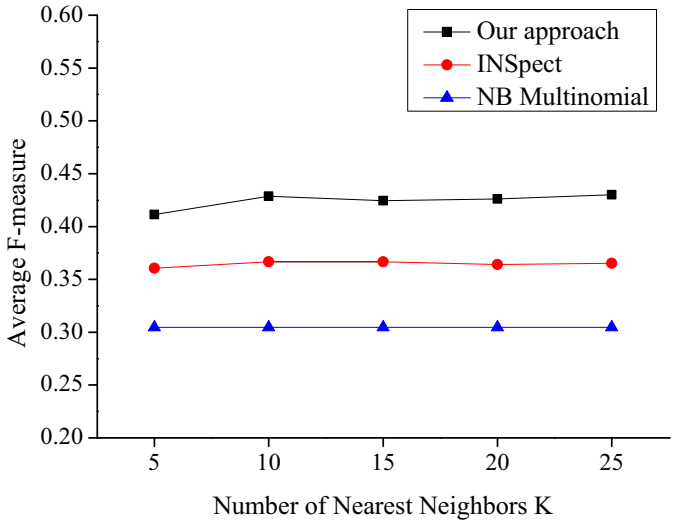

Fig. 8. Eclipse: Performance Comparison among severity prediction algorithms on different K neighbours.

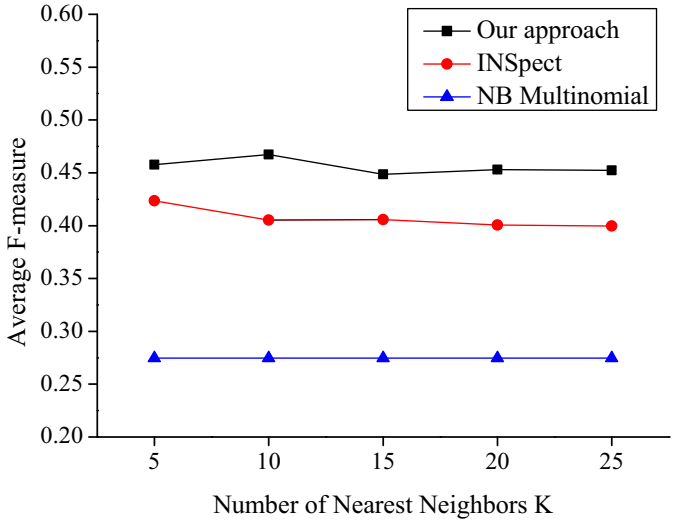

Fig. 9. Mozilla: Performance Comparison among severity prediction algorithms on different K neighbours.

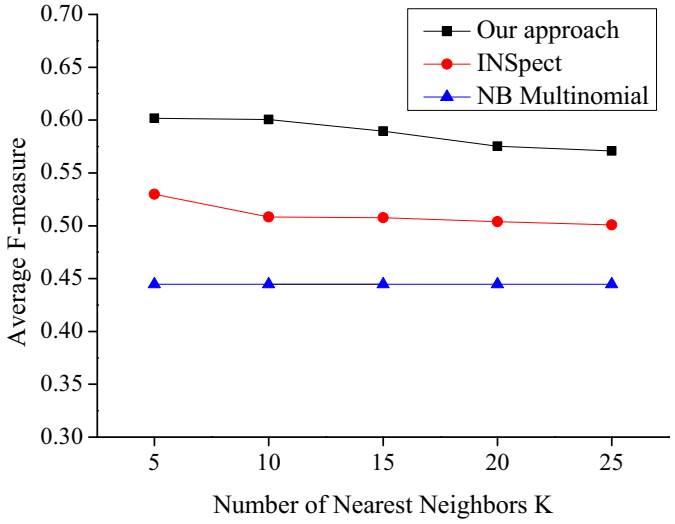

Fig. 10. GCC: Performance Comparison among severity prediction algorithms on different K neighbours.

Table 11
Results of the statistical tests for severity prediction.

Project	Null hypothesis	Normality value	Test type	p-value	Inverse hypothesis
Eclipse	$H1_0$	0.3663	t-test	1.668e-05	$H1_a$: Accept
	$H2_0$	0.1039	t-test	3.629e-06	$H2_a$: Accept
Mozilla	$H1_0$	0.8602	t-test	0.0005	$H1_a$: Accept
	$H2_0$	0.4617	t-test	6.14e-07	$H2_a$: Accept
GCC	$H1_0$	0.1378	t-test	5.528e-05	$H1_a$: Accept
	$H2_0$	0.3221	t-test	2.317e-05	$H2_a$: Accept

Table 12
Eclipse: performance comparison among semi-automatic fixer recommendation algorithms when K=5 on Precision, Recall, and F-measure.

Evaluation metrics	# Recommended fixers	Semi-automatic fixer recommendation methods			
		Our approach (%)	DRETOM (%)	DREX (%)	DevRec (%)
Precision	5	**25.63**	16.04	10.09	25.21
	10	**23.74**	12.58	12.89	15.73
Recall	5	**55.21**	26.62	19.87	45.8
	10	**69.71**	39.44	39.37	56.67
F-measure	5	**35.01**	20.02	13.38	32.51
	10	**35.42**	19.07	19.42	24.62

Table 13
NetBeans: performance comparison among semi-automatic fixer recommendation algorithms when K=5 on Precision, Recall, and F-measure.

Evaluation metrics	# Recommended fixers	Semi-automatic fixer recommendation methods			
		Our approach (%)	DRETOM (%)	DREX (%)	DevRec (%)
Precision	5	25.21	22.57	9.51	**27.97**
	10	**22.76**	15.55	12.83	17.03
Recall	5	**54.54**	32.77	16.6	41.71
	10	**65.47**	45.42	37.43	50.24
F-measure	5	**34.48**	26.73	12.09	33.49
	10	**33.78**	23.16	19.11	25.43

Table 14
Mozilla: Performance comparison among semi-automatic fixer recommendation algorithms when K=5 on Precision, Recall, and F-measure

Evaluation metrics	# Recommended fixers	Semi-automatic fixer recommendation methods			
		Our approach (%)	DRETOM (%)	DREX (%)	DevRec (%)
Precision	5	26.42	16.05	9.18	**27.8**
	10	**25.02**	12.58	10.20	18.23
Recall	5	**57.58**	26.62	12.84	40.71
	10	**70.46**	39.43	22.26	39.43
F-measure	5	**36.22**	20.02	10.71	33.03
	10	**36.93**	19.07	13.99	21.18

Furthermore, we present the corresponding inverse hypotheses as follows:

- $H1_a$: Our approach presents a significant difference with INSpect;
- $H2_a$: Our approach presents a significant difference with NB Multinomial

Then we adopt the arithmetic mean F-Measure values of the proposed approach and of the other two baselines as the input data when performing the statistical test. Specifically, if the normality value of the statistical test is smaller than 0.05, we use the Wilcoxon signed-rank test (W-test) (Wilcoxon, 1945) due to the non-normal distribution of the data; otherwise, we utilize the t-test (Boneau, 1960) due to the normal distribution of the data. Table 11 shows the results of the statistical tests.

We note that all normality values are more than 0.05, thus, we adopt t-test for all data. For the result, if the *p*-value is more than the significance level 0.05, we accept the corresponding null hypothesis. Otherwise, we accept the inverse hypothesis. This table shows that all p-values are less than 0.05, thus we accept all inverse hypotheses. In other words, our approach has a significant difference with INSpect and NB Multinomial.

According to the above-mentioned results of the performance comparison and statistical significance analysis, we can provide the answer to RQ2 as follows:

Answer to RQ2: Our approach on severity prediction outperforms INSPect and NB Multinomial.

4.5. Answer to RQ3: performance comparison among semi-automatic fixer recommendation methods

In order to answer RQ3, we select the cutting-edge studies, including DRETOM (Xie et al., 2012), DREX (Wu et al., 2011), and DevRec (Xia et al., 2015b), as the baselines to compare with the performance of our approach. Tables 12–16 show the comparison results among four semi-automatic fixer recommendation methods when using the top-5 nearest neighbours on Eclipse, NetBeans, Mozilla, OpenOffice, and GCC, respectively.

Table 15
OpenOffice: Performance comparison among semi-automatic fixer recommendation algorithms when K=5 on Precision, Recall, and F-measure.

Evaluation metrics	# Recommended fixers	Semi-automatic fixer recommendation methods			
		Our approach (%)	DRETOM (%)	DREX (%)	DevRec (%)
Precision	5	23.77	19.02	9.08	**25.65**
	10	**21.02**	13.23	11.47	17.47
Recall	5	**47.55**	36.68	16.92	38.71
	10	**62.54**	50.59	34.34	52.45
F-measure	5	**31.7**	25.06	11.82	30.86
	10	**31.47**	20.97	17.2	26.21

Table 16
GCC: Performance comparison among semi-automatic fixer recommendation algorithms when K=5 on Precision, Recall, and F-measure.

Evaluation metrics	# Recommended fixers	Semi-automatic fixer recommendation methods			
		Our approach (%)	DRETOM (%)	DREX (%)	DevRec (%)
Precision	5	30.76	29.04	12.56	**30.79**
	10	**25**	18.3	15.15	18.63
Recall	5	**47.18**	45.16	16.29	43.82
	10	**60.76**	56.97	30.96	52.24
F-measure	5	**37.24**	35.35	14.18	33.04
	10	**35.42**	27.7	20.34	27.47

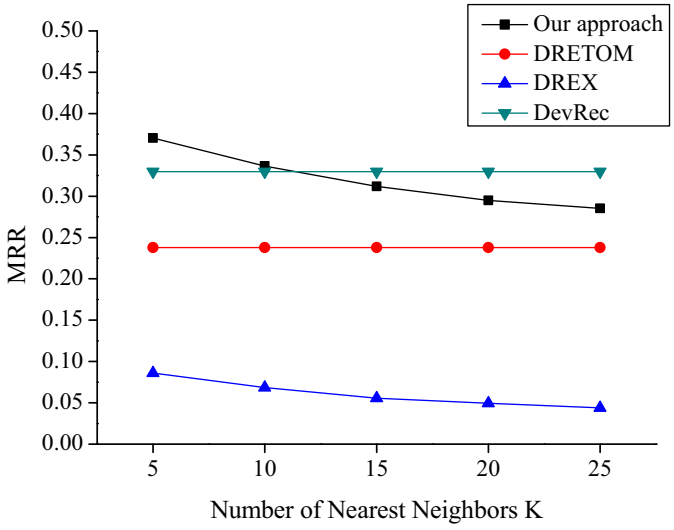

Fig. 11. Eclipse: Performance Comparison among semi-automatic fixer recommendation algorithms on MRR when varying K.

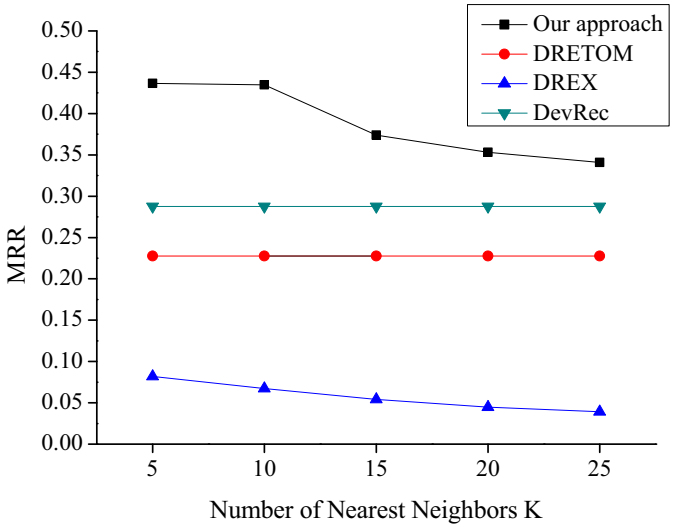

Fig. 12. NetBeans: Performance Comparison among semi-automatic fixer recommendation algorithms on MRR when varying K.

We now analyse the comparison results shown in these tables. For Eclipse, NetBeans, Mozilla, and OpenOffice datasets, our approach performs much better than DRETOM and DREX when recommending top-5 or top-10 fixers. In addition, when recommending top-5 fixers, the F-measure value of our approach is slightly better than DevRec. However, the improvement of the performance for DevRec increases when we recommend top-10 fixers. For GCC dataset, the F-measure value of our approach is better than DREX when recommending top-5 or top-10 fixers. Moreover, when recommending top-5 fixers, our approach performs slightly better than DRETOM and DevRec. However, the improvements of the performance for DREX and DevRec increase when we recommend top-10 fixers.

To further evaluate the performance of DRETOM, DREX, DevRec, and our approach, we utilize MRR to conduct the performance comparison. We show the results of the performance comparison for Eclipse, NetBeans, Mozilla, OpenOffice, and GCC in Figs. 11–15, respectively.

From these figures, the MRR values of DREX and our approach change as we increase K while the MRR values of DRETOM and DevRec do not change. Because DRETOM and DevRec do not adopt KNN to search the bug reports that are similar to the new bug report. For Eclipse, our approach performs better than DRETOM and DREX when varying K from 5 to 25. When K is set to 5 or 10, our approach presents the better performance than DevRec. However, if we continually increase K, DevRec performs better than our approach. For NetBeans and Mozilla, our approach performs better than DRETOM, DREX,and DevRec when varying K from 5 to 25. For OpenOffice, our approach presents the better performance than DREX when changing K from 5 to 25. When K is set to 5 or 10, our approach performs better than DRETOM and DevRec. If we increase K from 15 to 25, the MRR values of our approach are not better than them of DRETOM and DevRec. In this project, the MRR value (28.74%) of DRETOM is very close to the value (28.56%) of DevRec. For GCC, our approach performs better than DRETOM and DREX when varying K from 5 to 25. When we change K from 5

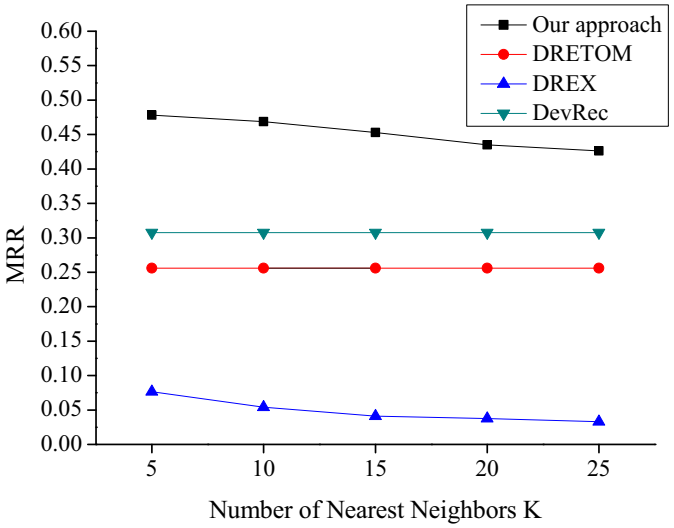

Fig. 13. Mozilla: Performance Comparison among semi-automatic fixer recommendation algorithms on MRR when varying K.

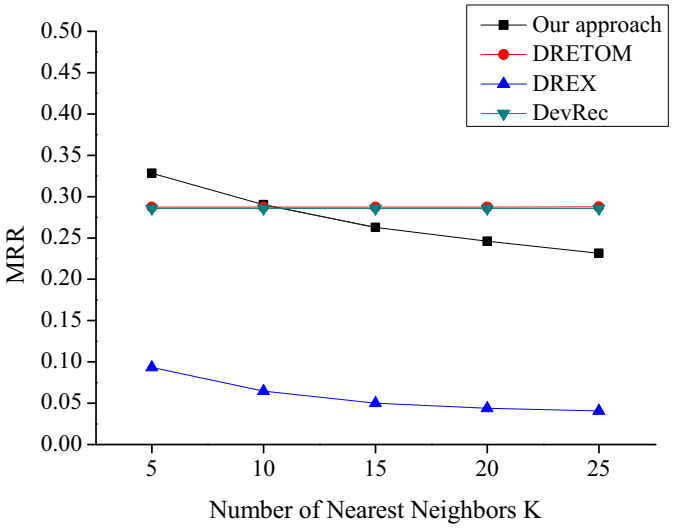

Fig. 14. OpenOffice: Performance Comparison among semi-automatic fixer recommendation algorithms on MRR when varying K.

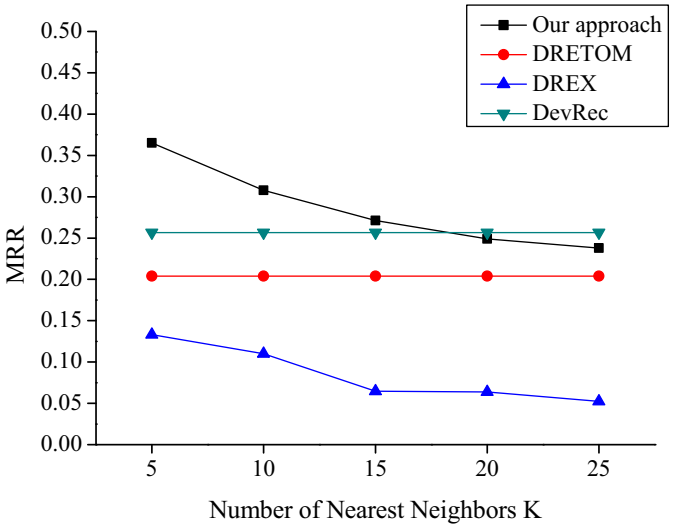

Fig. 15. GCC: Performance Comparison among semi-automatic fixer recommendation algorithms on MRR when varying K.

Table 17
Results of the statistical tests for semi-automatic fixer recommendation.

Project	Null hypothesis	Normality value	Test type	*p*-value	Inverse hypothesis
Eclipse	$H1'_0$	0.1889	t-test	0.0258	$H1'_a$: Accept
	$H2'_0$	0.3643	t-test	0.0203	$H2'_a$: Accept
	$H3'_0$	0.9012	t-test	0.0188	$H3'_a$: Accept
NetBeans	$H1'_0$	0.2636	t-test	0.0230	$H1'_a$: Accept
	$H2'_0$	0.6366	t-test	0.0334	$H2'_a$: Accept
	$H3'_0$	0.2468	t-test	0.0188	$H3'_a$: Accept
Mozilla	$H1'_0$	0.9258	t-test	0.0130	$H1'_a$: Accept
	$H2'_0$	0.6798	t-test	0.0260	$H2'_a$: Accept
	$H3'_0$	0.1517	t-test	0.0119	$H3'_a$: Accept
OpenOffice	$H1'_0$	0.7552	t-test	0.0156	$H1'_a$: Accept
	$H2'_0$	0.7230	t-test	0.0212	$H2'_a$: Accept
	$H3'_0$	0.1736	t-test	0.0293	$H3'_a$: Accept
GCC	$H1'_0$	0.3788	t-test	0.0476	$H1'_a$: Accept
	$H2'_0$	0.8786	t-test	0.0214	$H2'_a$: Accept
	$H3'_0$	0.8701	t-test	0.0028	$H3'_a$: Accept

to 15, our approach present the better performance than DevRec. However, if we increase K from 20 to 25, DevRec performs better than our approach.

Thus, if we choose the appropriate value of K, our approach performs better than DRETOM, DREX, and DevRec. Particularly, when the value of K is set to 5, our approach has the most significant improvement for the performance of fixer recommendation. To demonstrate it, we conduct a statistical test using R language. First, we define the three hypotheses as follows:

- $H1'_0$: Our approach shows no noteworthy difference against DRETOM;
- $H2'_0$: Our approach shows no noteworthy difference against DREX;
- $H3'_0$: Our approach shows no noteworthy difference against DevRec;

Furthermore, we present the corresponding inverse hypotheses as follows:

- $H1'_a$: Our approach shows no noteworthy difference against DRETOM;
- $H2'_a$: Our approach shows no noteworthy difference against DREX;
- $H3'_a$: Our approach shows no noteworthy difference against DevRec;

Then we adopt Precision, Recall, F-measure, and MRR values of the proposed approach and of the other three baselines when K is set to 5 as the input data to perform the statistical test. Table 17 shows the statistical results. All normality values are more than 0.05, therefore, we adopt t-test for all data. For the results, we note that all p-values are less than 0.05, thus we accept all inverse hypotheses. In other words, our approach has a significant difference with DRETOM, DREX, and DevRec when we choose the top-5 nearest neighbours to perform semi-automatic fixer recommendation.

According to the results of the performance comparison, we can give an answer to RQ3 as follows:

Answer to RQ3: Our approach for semi-automatic fixer recommendation outperforms the cutting-edge approaches such as DRETOM, DREX, and DevRec when we consider the appropriate number of the nearest neighbours of the new bug report.

4.6. Answer to RQ4: performance analysis

To answer the research question RQ4, we compare the performance using the different similarity metrics, including REP_{topic}, *REP*, and *cosine similarity* (Lazar et al., 2014), in our approach. Tables 18

Table 18
Performance comparison using the different similar metrics for severity prediction when K = 5.

Similarity metrics	Arithmetic mean values of F-measure per project		
	Eclipse (%)	Mozilla (%)	GCC (%)
REP_{topic}	**41.14**	**45.78**	**60.17**
REP	38.35	37.71	57.57
	(2.79)	(8.07)	(2.6)
Cosine	34.54	35.46	51.86
Similarity	(6.6)	(10.32)	(8.31)

Table 19
Performance comparison using the different similar metrics for semi-automatic fixer recommendation when recommending top-10 fixers and K = 5.

Similarity	F-measure per project				
metrics	Eclipse (%)	NetBeans (%)	Mozilla (%)	OpenOffice (%)	GCC (%)
REP_{topic}	**35.42**	**33.78**	**36.93**	**31.47**	**35.42**
REP	33.2	29.87	34.67	29.08	34.52
	(2.22)	(3.91)	(2.26)	(2.39)	(0.9)
cosine	28.04	24.98	18.89	22.04	29.82
similarity	(7.38)	(8.8)	(18.04)	(9.43)	(5.6)

and 19 shows the comparison results when $K = 5$ for the severity prediction algorithm and the semi-automatic fixer recommendation algorithm, respectively.

In Tables 18 and 19, the F-measure values in the parentheses mean the improvement for *REP* or *cosine similarity*. We note that using the enhanced version REP_{topic} can improve the performance of severity prediction and semi-automatic fixer recommendation using *REP* and *cosine similarity* in our approach. The major reason is that adding the feature *topics* can help us search the more accurate nearest neighbours of the new bug report so that it can improve the results of two tasks. Thus, we can give an answer to RQ4 as follows:

Answer to RQ4: Using REP_{topic} can improve the performance of severity prediction and semi-automatic fixer recommendation in our approach.

In addition to REP_{topic}, we propose the new severity prediction and semi-automatic algorithms. For severity prediction, the novel method considers the number of bug reports as the predicting severity label and the textual similarities between the nearest neighbours labelled by the severity label and the new bug report; for semi-automatic fixer recommendation, the proposed algorithm considers the candidate fixers' behaviours in the social network and their experience on fixing the historical bug reports. These introduced features can improve the accuracy of our approach for predicting the severity labels and recommending the appropriate fixers.

5. Threats to validity

In this section, we introduce some possible threats of our study. These threats include external threats and internal threats, which are presented as follows:

- **External threats:** We collect the fixed bug reports from five open source bug repositories for performing the experiments. However, we are not sure that the proposed approaches are also effective in other open source projects and commercial projects. Because we extract a lot of bug reports from the five lager-scale projects, the threat may be slight. For other projects, because the internal rules (*e.g.*, variations in development process) may be different from the five projects used in our experiments, we should extract the different features to develop a new approach to perform severity prediction and semi-automatic fixer recommendation.
- **Internal threats:** In our approach, we spent much effort in tuning a lot of parameters to achieve the best performance. Even though we adopt the parameter adjustment methods proposed in Sun et al. (2011) and Xia et al. (2015b) for REP_{topic} and our developed algorithms, respectively. However, a more effective tuning method needs to be developed. We will do it in the future.
  When analysing the distribution of the bug reports as each severity label, we find the imbalanced data appearing in NetBeans and OpenOffice. They can impact the results of the severity prediction, we will process these data in the future so that we can predict the severity labels in the projects which contain the imbalanced data.

6. Related work

In this section, we introduce some previous studies related to severity prediction and semi-automatic fixer recommendation. Moreover, we show some other tasks such as bug summarization, bug localization, and priority prediction.

6.1. Severity prediction

Previous studies usually tend to use machine learning algorithms to predict the severity level of a new submitted bug. As an early work, Menzies and Marcus (2008) proposed SEVERIS to assist triagers to assign severity levels to given bug reports. SEVERIS adopted text mining and rule-learning techniques to predict the severity levels. Lamkanfi et al. (2010) utilized text mining technique and NB classifier to predict whether the given bug is "Non-severe" or "Severe". In their follow-up work, Lamkanfi and his colleagues (Lamkanfi et al., 2011) compared four machine learning algorithms including NB, NB Multinomial, KNN, and SVM on coarse-grained severity prediction (*i.e.*, "Non-severe" or "Severe"). The experimental results showed that NB Multinomial performed best than others in Eclipse and GNOME. Tian et al. (2012) used information retrieval technique, in particular BM25-based textual similarity algorithm (*i.e., REP*), and KNN to predict the severity levels of each new bug report. This fine-grained severity prediction method showed the better performance than SEVERIS (Menzies and Marcus, 2008). Yang et al. (2012) adopted feature selection schemas such as information gain, chi-square, and correlation coefficient to select the best features as the input of Naive Bayes Multinomial. The evaluation results showed that the feature selection schemes can further improve the performance of severity prediction.

Our work is different from these previous studies. First, we propose an enhanced version of *REP, i.e.,* REP_{topic}, to extract the top-K nearest neighbours of a new bug report. Second, we develop a new prediction algorithm based on the textual similarities between these neighbours and the given bug report.

6.2. Semi-automatic fixer recommendation

As a premier work, Čubranić and Murphy (Čubranić and Murphy, 2004) utilized NB classifier to predict whether a candidate developer is able to fix a new bug. Anvik et al. (2006) adopted NB, SVM, and C4.5 to perform semi-automatic bug triage. The experimental results showed that SVM performed better than others in Firefox and Eclipse, and it also presented higher precision than NB in GCC open bug repository. Matter et al. (2009) modelled developers' expertises by comparing the vocabularies in source code and bug reports which are assigned to them.

Recent years, a series new techniques such as social network and topic model are also adopted for helping to improve the performance of semi-automatic bug triage. Wu et al. (2011) proposed DREX, which utilized KNN to search the historical bug reports that similar to the new bug and introduced social network metrics such as Out-Degree to rank the candidate developers for recommending the best one for fixing the given bug. Xuan et al. (2012) proposed a method to prioritize the developers via social network. Based on the prioritization result, they utilized NB and SVM to predict whether the candidate developer is the most appropriate fixer. Park et al. (2011) modelled "developer profiles" to indicate developers' estimated costs for fixing different types of bugs, which are extracted by apply LDA. The evaluation results showed that this cost-aware triage algorithm can optimize for both accuracy and cost for semi-automatic bug triage. Tamrawi et al. (2011) proposed Bugzie, which combines the fuzzy sets corresponding to the terms extracted from the new bug report and ranks the developers to find the most capable fixers. Xie et al. (2012) proposed DRETOM to model the topics for bug reports and calculate the probability of a developer being interested in and expertise on resolving the bugs so that the candidate developers were ranked according the probabilities. Experimental results on Eclipse JDT and Mozilla Firefox open bug repositories showed that DRETOM can achieve higher performance. As one of our previous work (Zhang and Lee, 2013), we proposed a hybrid bug triage algorithm to recommend the most appropriate bug fixer. The proposed approach combines experience model and probability model to implement semi-automatic bug triage. In detail, experience model captures the developers' experience on resolving historical bugs by extracting the features such as the number of fixed bug reports while probability model is built by analysing the social network between the candidate developers. Naguib et al. (2013) introduced activity profiles to rank all candidate developers. An activity profile captures each developer's activities (*i.e.*, review, assign, and resolve the corresponding bugs) in the bug fixing process so that it can influence and contribute to the ranking of suitable candidates. The evaluation result showed that the proposed method performed better than LDA-SVM-based developer recommendation technique. Xia et al. (2015b) proposed DevRec algorithm to fix the bug fixers, it performed two kinds of analysis, including bug reports based analysis and developer based analysis. DevRec extracted topics, component, product, developers, summary, and description as the features to perform semi-automatic fixer recommendation on five open source projects which are same as our datasets. The experimental results showed that DevRec performed better than DREX (Wu et al., 2011) and Bugzie (Tamrawi et al., 2011). Xuan et al. (2015) reduced the size of the data set to improve the quality of the bug reports, thus they can increase the accuracy of bug triage using machine learning techniques.

Different from these described previous studies, we utilized the similarity measure REP_{topic} and KNN to find the most similar bug reports (*i.e.*, top-K nearest neighbours) of a new bug report. Moreover, the new developed ranking algorithm combines the social network-based analysis and the experience-based analysis to improve the accuracy of semi-automatic fixer recommendation.

6.3. Other tasks in bug resolution process

Except for severity prediction and semi-automatic fixer recommendation, there are other tasks such as bug report summarization, bug localization, and priority prediction in bug resolution process.

The goal of bug report summarization is to automatically generate the summary of a given bug report. Rastkar et al. (2010) utilized three supervised classifiers including Email Classifier (EC), Email & Meeting Classifier (EMC), and Bug Report Classifier (BRC) to verify whether a sentence is a part of the extractive summary. Mani et al. (2012) adopted four unsupervised approaches, including Centroid, Maximum Marginal Relevance (MMR), Grasshopper and Diverse Rank (DivRank), to implement bug report summarization. The results showed that MMR, DivRank, and Grasshopper can achieve the same performance with the best of the supervised approach but reduced the running cost. Jiang et al. (2016) utilized byte-level N-grams to model the authorship characteristics of developers. Then, they employed the authorship characteristics to extract similar bug reports to facilitate the task of bug report summarization. Experiments validated the effectiveness of this new technique. Najam et al. (2016) used small-scale crowdsourcing based features to summarize the source code fragment. The experimental results showed that the proposed approach performed better than existing code fragment classifiers.

Bug localization aims to find the location of a new bug report so that it can reduce the fixers' workload. Lukins et al. (2008) utilized LDA to search the correct source code file where the new bug appears. In this work, the bug report is treated as a query and then the approach performs LDA to retrieve the corresponding source code file where the bug is. Rao and Kak (2011) compared the performance of different IR-models, including Unigram Model, Vector Space Model, Latent Semantic Analysis Model, LDA, and Cluster Based Document Model when performing the task of bug localization. Kim et al. (2013) proposed a two-phase recommendation model. In this model, they adopted Naive Bayes to filter out the uninformative bug reports and predict the buggy file for each submitted bug report. Saha et al. (2013) built AST to extract the program constructs of each source code file, and utilized Okapi BM25 to calculate the similarity between the given bug report and the constructs of each candidate buggy file. Zhou et al. (2012) proposed BugLocator to rank all files based on textual similarity between the new bug report and the source code using a revised Vector Space Model. Moreover, they also rank the relevant files by analysing the historical bug reports that similar to the given bug. Finally, by combining two ranks, BugLocator can return the correct source file to locate the given bug. Sisman and Kak (2012) used time decay in weighting the files in a probabilistic IR model to perform bug localization. Zamani et al. (2014) proposed a feature location approach using a new term-weighting technique that considers how recently a term has been used in the repositories. The empirical evaluation of the approach shows that it performs better than Vector Space Model-based approach.

Similar to severity, priority is an important feature of bug report. It indicates which bug should be fixed preferentially. Sharma et al. (2012) used SVM, NB, KNN, and Neural Network to predict the priority level of a new bug report. Tian et al. (2013) extracted multiple factors to train a discriminative model to verity which priority level that the given bug report belongs to.

7. Conclusion

In this paper, we propose a new approach to implement severity prediction and semi-automatic fixer recommendation instead of developers' manual work. Our approach extracts the top-K nearest neighbours of the new bug report by utilizing a similarity measure REP_{topic}, which is an enhanced version of *REP*. Next we adopt the K neighbours' features such as the participated developers and the textual similarities with the given bug report to implement severity prediction and semi-automatic fixer recommendation.

To demonstrate the effectiveness of our approach, we apply our approach to five popular open bug repositories, including GCC, OpenOffice, Eclipse, NetBeans, and Mozilla. The results show that the proposed approach outperforms the cutting-edge approaches. For severity prediction, our approach performs better than INSpect and NB multinomial; for semi-automatic fixer

recommendation, our approach presents the better performance than DRETOM, DREX, and DevRec. The evaluation results demonstrate that the similarity measure REP_{topic} can help to improve the performance of the two bug resolution tasks.

In the future, we will evaluate our approach on more bug reports extracted from other open and commercial software repositories. Moreover, we plan to develop a more accurate developer ranking algorithm by capturing other features such as the attachment to further improve the performance of semi-automatic fixer recommendation. At the same time, we also want to introduce new machine learning techniques such as deep learning to implement severity prediction.

Acknowledgement

This work is supported in part by the Hong Kong GRF (No. PolyU5389/13E), the Hong Kong ITF(No. UIM/285), the National Natural Science Foundation of China (No. 61202396), the HKPolyU Research Grant (G-UA3X), and Shenzhen City Science and Technology R&D Fund (No. JCYJ20150630115257892).

References

Anvik, J., Hiew, L., Murphy, G.C., 2006. Who should fix this bug? In: Proceedings of the 28th International Conference on Software Engineering, ICSE'06, pp. 361–370.

Blei, D.M., Lafferty, J.D., 2007. A correlated topic model of science. Ann. Appl. Stat. 17–35.

Boneau, C.A., 1960. The effects of violations of assumptions underlying the T test. Psychological Bull. 57 (1), 49–64.

Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., Hullender, G., 2005. Learning to rank using gradient descent. In: Proceedings of the 22nd International Conference on Machine Learning, ICML'05, pp. 89–96.

Castells, P., Fernandez, M., Vallet, D., 2007. An adaptation of the vector-space model for ontology-based information retrieval. IEEE Trans. Knowl. Data Eng. 19 (2), 261–272.

Chemudugunta, C., Steyvers, P.S.M., 2007. Modeling general and specific aspects of documents with a probabilistic topic model. In: Proceedings of the 20th Annual Conference on Neural Information Processing Systems, NIPS'06, Vol. 19, p. 241.

Čubranić, D., Murphy, G.C., 2004. Automatic bug triage using text categorization. In: Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering, SEKE'04, pp. 92–97.

Goutte, C., Gaussier, E., 2005. A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In: Proceedings of the 27th European Conference on IR Research, ECIR'05, pp. 345–359.

Hooimeijer, P., Weimer, W., 2007. Modeling bug report quality. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, ASE'07, pp. 34–43.

Kim, D., Tao, Y., Kim, S., Zeller, A., 2013. Where should we fix this bug? A two-phase recommendation model. IEEE Trans. Softw. Eng. 39 (11), 1597–1610.

Kwak, H., Lee, C., Park, H., Moon, S., 2010. What is twitter, a social network or a news media? In: Proceedings of the 19th International Conference on World Wide Web, WWW'10, pp. 591–600.

Jeong, G.G., Kim, S., Zimmermann, T., 2009. Improving bug triage with bug tossing graphs. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE'09, pp. 111–120.

Jiang, H., Zhang, J., Ma, H., Nazar, N., Ren, Z., 2016. Mining authorship characteristics in bug repositories, SCIENCE CHINA Information Sciences, accepted.

Lamkanfi, A., Demeyer, S., Giger, E., Goethals, B., 2010. Predicting the severity of a reported bug. In: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories, MSR'10, pp. 1–10.

Lamkanfi, A., Demeyer, S., Soetens, Q.D., Verdonck, T., 2011. Comparing mining algorithms for predicting the severity of a reported bug. In: Proceedings of the 15th European Conference on Software Maintenance and Reengineering, CSMR'11, pp. 249–258.

Lazar, A., Ritchey, S., Sharif, B., 2014. Improving the accuracy of duplicate bug report detection using textual similarity measures. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR'14, pp. 308–311.

Liu, C., Yang, J., Tan, L., Hafiz, M., 2013. R2fix: Automatically generating bug fixes from bug reports. In: Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation, ICST'13, pp. 282–291.

Lukins, S., Kraft, N., Etzkorn, L., 2008. Source code retrieval for bug localization using latent Dirichlet allocation. In: Proceedings of the 15th Working Conference on Reverse Engineering, WCRE'08, pp. 155–164.

Mani, S., Catherine, R., Sinha, V.S., Dubey, A., 2012. Ausum: Approach for unsupervised bug report summarization. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE'12, pp. 1–11.

Matter, D., Kuhn, A., Nierstrasz, O., 2009. Assigning bug reports using a vocabulary-based expertise model of developers. In: Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories, MSR'09, pp. 131–140.

Menzies, T., Marcus, A., 2008. Automated severity assessment of software defect reports. In: Proceedings of the 24th IEEE International Conference on Software Maintenance, ICSM'08, pp. 346–355.

Naguib, H., Narayan, N., Brugge, B., Helal, D., 2013. Bug report assignee recommendation using activity profiles. In: Proceedings of the 10th IEEE Working Conference on Mining Software Repositories, MSR'13, pp. 22–30.

Najam, N., Jiang, H., Gao, G., Zhang, T., Li, X., Ren, Z., 2016. Source code fragment summarization with small-scale crowdsourcing based features, Online http://link.springer.com/article/10.1007/s11704-015-4409-2.

Park, J.-W., Lee, M.-W., Kim, J., won Hwang, S., Kim, S., 2011. Costriage: A cost-aware triage algorithm for bug reporting systems. In: Proceedings of the 25th AAAI Conference on Artificial Intelligence, AAAI'11, pp. 139–144.

Ramage, D., Hall, D., Nallapati, R., Manning, C.D., 2009. Labeled lda: a supervised topic model for credit attribution in multi-labeled corpora. In: Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing, EMNLP'09, pp. 248–256.

Rao, S., Kak, A., 2011. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR'11, pp. 43–52.

Rastkar, S., Murphy, G.C., Murray, G., 2010. Summarizing software artifacts: A case study of bug reports. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1. New York, NY, USA, pp. 505–514.

Saha, R., Lease, M., Khurshid, S., Perry, D., 2013. Improving bug localization using structured information retrieval. In: Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering, ASE'13, pp. 345–355.

Servant, F., Jones, J.A., 2012. Whosefault: automatic developer-to-fault assignment through fault localization. In: Proceedings of the 34th International Conference on Software Engineering, ICSE'12, pp. 36–46.

Sharma, M., Bedi, P., Chaturvedi, K., Singh, V., 2012. Predicting the priority of a reported bug using machine learning techniques and cross project validation. In: Proceedings of the 12th International Conference on Intelligent Systems Design and Applications, ISDA'12, pp. 539–545.

Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W.M., Ohira, M., Adams, B., Hassan, A.E., Matsumoto, K.I., 2010. Predicting re-opened bugs: a case study on the eclipse project. In: Proceedings of the 17th Working Conference on Reverse Engineering, WCRE'10, pp. 249–258x.

Sisman, B., Kak, A.C., 2012. Incorporating version histories in information retrieval based bug localization. In: Proceedings of IEEE Working Conference on Mining Software Repositories, MSR'12, pp. 50–59.

Sun, C., Lo, D., Khoo, S.-C., Jiang, J., 2011. Towards more accurate retrieval of duplicate bug reports. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, ASE'11, pp. 253–262.

Tamrawi, A., Nguyen, T.T., Al-Kofahi, J.M., Nguyen, T.N., 2011. Fuzzy set and cache-based approach for bug triaging. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE'11, pp. 365–375.

Taylor, M., Zaragoza, H., Craswell, N., Robertson, S., Burges, C., 2006. Optimisation methods for ranking functions with multiple parameters. In: Proceedings of the 15th ACM International Conference on Information and Knowledge Management, CIKM'06, pp. 585–593.

Team, R. C., 2014. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, vienna, austria.

Tian, Y., Lo, D., Sun, C., 2012. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In: Proceedings of the 19th Working Conference on Reverse Engineering, WCRE'12, pp. 215–224.

Tian, Y., Lo, D., Sun, C., 2013. Drone: Predicting priority of reported bugs by multi-factor analysis. In: Proceedings of the 29th IEEE International Conference on Software Maintenance, ICSM'13, pp. 200–209.

Wilcoxon, F., 1945. Individual comparisons by ranking methods. Biom. Bull. 80–83.

Wu, W., Zhang, W., Yang, Y., Wang, Q., 2011. Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking. In: Proceedings of the 18th Asia Pacific Software Engineering Conference, APSEC'11, pp. 389–396.

Xia, X., Lo, D., Shihab, E., Wang, X., 2015a. Automated bug report field reassignment and refinement prediction. IEEE Trans. Reliab. 1–20.

Xia, X., Lo, D., Wang, X., Zhou, B., 2013. Accurate developer recommendation for bug resolution. In: Proceedings of the 20th Working Conference on Reverse Engineering, WCRE'12, pp. 72–81.

Xia, X., Lo, D., Wang, X., Zhou, B., 2015b. Dual analysis for recommending developers to resolve bugs. J. Softw. Evol. Process 27 (3), 195220.

Xia, X., Lo, D., Wen, M., Shihab, E., Zhou, B., 2014. An empirical study of bug report field reassignment. In: Proceedings of the 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering, CSMR-WCRE'14, pp. 174–183.

Xie, X., Zhang, W., Yang, Y., Wang, Q., 2012. Dretom: Developer recommendation based on topic models for bug resolution. In: Proceedings of the 8th International Conference on Predictive Models in Software Engineering, PROMISE'12, pp. 19–28.

Xuan, J., Jiang, H., Ren, Z., Zou, W., 2012. Developer prioritization in bug repositories. In: Proceedings of the 34th International Conference on Software Engineering, ICSE'12, pp. 25–35.

Xuan, J., Jiang, H., Hu, Y., Ren, Z., Zou, W., Luo, Z., Wu, X., 2015. Towards effective bug triage with software data reduction techniques. IEEE Trans. Knowl. Data Eng. 27 (1), 264–280.

Yang, C.-Z., Hou, C.-C., Kao, W.-C., Chen, I.-X., 2012. An empirical study on improving severity prediction of defect reports using feature selection. In: Proceedings of the 19th Asia-Pacific Software Engineering Conference, APSEC'12, pp. 240–249.
Yang, G., Zhang, T., Lee, B., 2014. Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports. In: Proceedings of the IEEE 38th Annual Computer Software and Applications Conference, COMPSAC'14, pp. 97–106.
Zamani, S., Lee, S.P., Shokripour, R., Anvik, J., 2014. A noun-based approach to feature location using time-aware term-weighting. Inf. Softw. Technol. 56 (8), 991–1011.
Zhang, T., Lee, B., 2013. A hybrid bug triage algorithm for developer recommendation. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC'13, pp. 1088–1094.
Zhou, J., Zhang, H., Lo, D., 2012. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the 34th International Conference on Software Engineering, ICSE'12, pp. 14–24.

Tao Zhang, received the B.S. and M.E. degrees in Automation and Software Engineering from Northeastern University, China, in 2005 and 2008, respectively. He got the Ph.D. degree in Computer Science from University of Seoul, South Korea in Feb., 2013. He was a postdoctoral fellow at the Department of Computing, Hong Kong Polytechnic University from November 2014 to November 2015. Currently, he is an assistant professor at the School of Software, Nanjing University of Posts and Telecommunications. His research interest includes data mining, software maintenance and natural language processing. Dr. Zhang is a member of IEEE, ACM, and IEICE.

Jiachi Chen, received the B.S. degree in Institute of Service Engineering, HangZhou Normal University, China in 2016. Currently, he is a master student in Department of Computing, Hong Kong Polytechnic University. He got Sliver Award in 12th Zhejiang Undergraduate ACM Program Design Competition. His research interest includes data mining and program analysis.

Geunseok Yang, received the B.S. degree in Computer Science from Korea University of Technology and Education, South Korea in 2013. Currently, he is a master student at the Department of Computer Science, University of Seoul, South Korea. His research interest includes mining software repositories and big data.

Byungjeong Lee, received the B.S., M.S., Ph.D. degrees in Computer Science from Seoul National University in 1990, 1998, and 2002, respectively. He was a researcher of Hyundai Electronics, Corp. from 1990 to 1998. Currently, he is a professor of the Department of Computer Science at the University of Seoul, South Korea. His research areas include software engineering and web science.

Xiapu Luo, received his B.S. in Communication Engineering and M.S. in Communications and Information Systems from Wuhan University. He obtained his Ph.D. degree in Computer Science from the Hong Kong Polytechnic University, under the supervision of Prof. Rocky K.C. Chang. After that, Daniel spent two years at the Georgia Institute of Technology as a post-doctoral research fellow advised by Prof. Wenke Lee. Currently, he is a research assistant professor at the Department of Computing, Hong Kong Polytechnic University. His research interests include Software Analysis, Android Security and Privacy, Network and System Security, Information Privacy, Internet Measurement, Cloud Computing, and Mobile Networks.