2017 IEEE 25th International Conference on Program Comprehension (ICPC)

Bug Report Enrichment with Application of Automated Fixer Recommendation

Tao Zhang*†, Jiachi Chen†, He Jiang‡, Xiapu Luo†[a], Xin Xia§
*College of Computer Science and Technology, Harbin Engineering University, China
†Department of Computing, The Hong Kong Polytechnic University, Hong Kong
‡School of Software, Dalian University of Technology, China
§Department of Computer Science, University of British Columbia, Canada
cstzhang@hrbeu.edu.cn, chenjiachi317@gmail.com, csxluo@comp.polyu.edu.hk, jianghe@dlut.edu.cn, xxia02@cs.ubc.ca

***Abstract*—For large open source projects (*e.g.*, Eclipse, Mozilla), developers usually utilize bug reports to facilitate software maintenance tasks such as fixer assignment. However, there are a large portion of short reports in bug repositories. We find that 78.1% of bug reports only include less than 100 words in Eclipse and require bug fixers to spend more time on resolving them due to limited informative contents. To address this problem, in this paper, we propose a novel approach to enrich bug reports. Concretely, we design a sentence ranking algorithm based on a new textual similarity metric to select the proper contents for bug report enrichment. For the enriched bug reports, we conduct a user study to assess whether the additional sentences can provide further help to fixer assignment. Moreover, we assess whether the enriched versions can improve the performance of automated fixer recommendation. In particular, we perform three popular automated fixer recommendation approaches on the enriched bug reports of Eclipse, Mozilla, and GNU Compiler Collection (GCC). The experimental results show that enriched bug reports improve the average F-measure scores of the automated fixer recommendation approaches by up to 10% for DREX, 13.37% for DRETOM, and 8% for DevRec when top-10 bug fixers are recommended.**

I. Introduction

A bug repository is a collection of bug reports stored in a defined directory structure. In the development process of large open source projects, developers or users create and update their bug reports. Bug reports are important resources that help developers understand the bugs' details and fix them. To reduce the developers' workload, some recent studies utilize machine learning techniques to analyze the rich information in bug reports for performing several software maintenance tasks such as automated fixer recommendation [1]–[10]. For example, Anvik et al. [2] utilized a machine learning technique to predict a small number of developers suitable for fixing a new reported bug.

A statistical analysis (See Section II) on bug reports from three open source projects (*i.e.*, Eclipse [11], Mozilla [12] and GNU Compiler Collection (GCC) [13]) shows that short bug reports occupy a high proportion. For example, 32,198 (78.1%) bug reports collected from Eclipse only include less than 100 words. Unfortunately, these short bug reports may influence the fixing time. Based on our statistical results(Section II-B), the given bugs reported in short reports need more time to be resolved than those in long reports. For instance, in Eclipse, the average bug fixing time for bug reports with less than 100 words is 409.8 days, which is 121 days more than bug reports with 400-499 words.

The reason why the short reports delay the fixing time may lie in the lack of informative contents in bug reports. Hooimeijer and Weimer reported that the lack of resource often causes the delay of bug fixing [14]. According to the results of the questionnaire on bug report quality [15], Bettenburg et al. find that time delay is due to the absent information for bug reports. Thus, we analyze the *ingredients* of the bug reports in our data sets (See Section II). The results show that most of short bug reports only contain the textual contents written in natural language and lack the important ingredients (*i.e.*, `stack trace(s)`, `code example(s)`, and `patch(es)`) defined by Bettenburg et al. [15]. In addition, according to the results of the questionnaire survey shown in Section II, most of developers working on open source projects agree that lack of the ingredients will extend the time of fixer assignment so that the overall fixing time is prolonged. Therefore, it is promising to enrich bug reports for avoiding fixing-time delay.

In this paper, we propose a new approach to **enrich** bug reports. The concept "**enrich**" means adding more detailed information (*e.g.*, the reason why a bug appears) to a given bug report. When a new bug report comes, our approach automatically supplements its textual content by a list of sorted sentences, which are extracted from historical bug reports. Moreover, we leverage these enriched versions to conduct automated fixer recommendation in bug repositories. In detail, we select three popular approaches, including DREX (Developer Recommendation with k-nearest-neighbor search and EXpertise ranking) [3], DRETOM (Developer REcommendation based on Topic Models) [4], and DevRec [5], to assess the usefulness of the enriched bug reports. Note that these approaches utilize the different elements (*i.e.*, social network, topic model, hybrid model of social network and topic model) related to bug reports. DREX utilized K-Nearest-Neighbor search with bug similarity and the ranking of developers to recommend bug fixers. DRETOM captured developers' interest in bug resolving activities and expertise by using the topic models to build the ranking algorithm for fixer recommendation. DevRec combined bug reports analysis and developer based analysis to recommend a list of ranked

[a]Corresponding author

978-1-5386-0535-6/17 $31.00 © 2017 IEEE
DOI 10.1109/ICPC.2017.28

bug fixers. The experimental results on Eclipse, Mozilla, and GCC show that enriched bug reports can improve the average F-measure scores of the fixer recommendation approaches by up to 10% for DREX, 13.37% for DRETOM, and 8% for DevRec when we recommend top-10 bug fixers.

To help researchers and developers reproduce our work, we open all data sets, source code, and experimental results at https://github.com/BREnrich/Enriching-Bug-Report.

We summarize the major contributions as follows:

- To our best knowledge, this is the *first* study on enriching bug reports. We analyze the descriptions of all bug reports in our data sets, and the result reveals the reason why the short bug reports cause the delay of bug fixing, namely, the lack of some important ingredients such as `stack trace(s)`, `code example(s)`, and `patch(es)`.
- We propose a novel approach to enrich bug reports by adding the sentences in historical bug reports, which are strongly related to the given bug report. The user study shows that the enriched bug reports are useful to the manual fixer assignment.
- We conduct careful experiments to evaluate whether the enriched bug reports can effectively improve the performance of automated fixer recommendation. The results show that the enriched bug reports obviously increase the average F-measure scores of the automated fixer recommendation approaches.

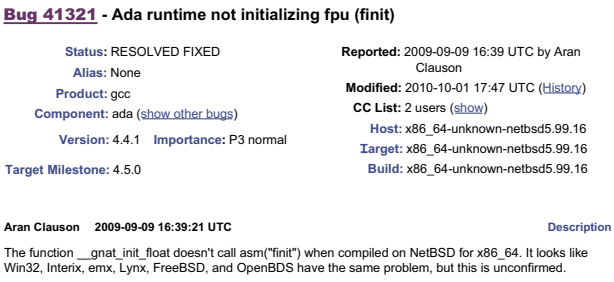
Roadmap. Section II introduces the background knowledge and motivation of our research. Section III details how to enrich the given bug reports. We detail the research questions for evaluation in Section IV, and then present the evaluation results of manual fixer assignment and automated fixer recommendation in Section V and Section VI, respectively. After introducing the threats to validity in Section VII and the related work in Section VIII, we summarize this paper and present future work in Section IX.

II. Background Knowledge and Motivation

Bug reports play a pivotal role when developers perform the activities of software maintenance. For example, developers need to read bug reports to understand how bugs occur. In this section, we introduce what is a bug report and show the motivation of our work.

A. What is a bug report?

Bug reports detail how the bugs occur, and they can help triagers assign the correct developers to fix the corresponding bugs. Fig. 1 shows an example of bug report #`41321` from GCC. Note that this bug report consists of non-textual information and free text. The non-textual information includes factors (*e.g.*, `component`, `product`); the main free-text contains the `summary` (or `title`) and `description`. The `summary` presents a high-level overview of the bug while the `description` is the main body of a bug report, which provides the detailed information of the given bug.


Bug 41321 - Ada runtime not initializing fpu (finit)

Status: RESOLVED FIXED
Alias: None
Product: gcc
Component: ada (show other bugs)
Version: 4.4.1 Importance: P3 normal
Target Milestone: 4.5.0

Reported: 2009-09-09 16:39 UTC by Aran Clauson
Modified: 2010-10-01 17:47 UTC (History)
CC List: 2 users (show)
Host: x86_64-unknown-netbsd5.99.16
Target: x86_64-unknown-netbsd5.99.16
Build: x86_64-unknown-netbsd5.99.16

Aran Clauson 2009-09-09 16:39:21 UTC Description

The function __gnat_init_float doesn't call asm("finit") when compiled on NetBSD for x86_64. It looks like Win32, Interix, emx, Lynx, FreeBSD, and OpenBDS have the same problem, but this is unconfirmed.

Fig. 1: An example of bug report in GCC

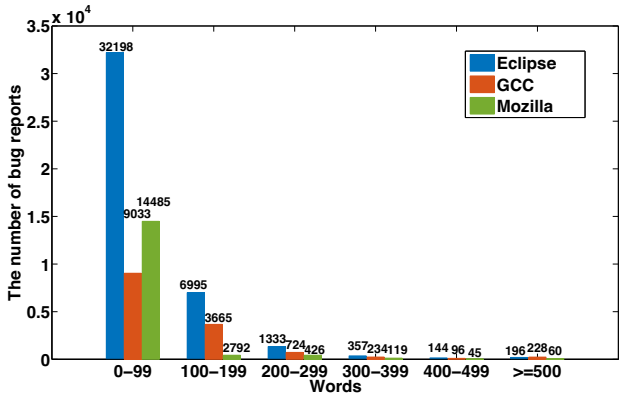


Fig. 2: The distribution of bug reports with their lengths (words)

B. Motivation

Bug reports vary in their quality of content. For instance, some bug reports do not describe enough information about the bug to be fixed. Unfortunately, we find that most of bug reports are short by investigating the length of bug reports from three open source projects, namely, Eclipse, Mozilla, and GCC. For each project, we only collect the bug reports of fixed bugs, which are denoted by 'resolved' or 'closed'. The details of the data sets are shown in Table I. The time span covers more than 10 years (*e.g.*, from 2001 to 2015 in Eclipse). Fig. 2 shows the distribution of all bug reports according to their lengths (*i.e.*, the number of words in the descriptions) in three projects.

Note that most bug reports only include less than 100 words. With the growth of the length, the number of bug reports sharply decreases. For example, there are only 60 bug reports with more than 500 words in Mozilla. This statistical result illustrates that developers or users tend to write short bug reports. In this case, we throw out a question: "*Are these short bug reports enough to fix bugs?*"

In order to look for the answer, we analyze the relationship between the fixing time and the length of bug reports. Fig. 3 shows the average fixing time for the bug reports in three open source bug repositories with different length. For the short bug reports, developers have to spend more time in fixing the corresponding bugs. For example, developers averagely spend 409.8 days in fixing the Eclipse bug reports which only contain less than 100 words. The lack of important information in bug reports may be a reason to the extended fixing time. Since bug fixers have to collect more detailed information from other channels (*e.g.*, duplicate bug reports) to fix the given bugs,

TABLE I: Data sets for fixed bug reports in three open source projects

Project	# bug reports	#sentences	#avg. sentences per report	Period
Eclipse	41,223	203,679	4.9	10/2001-12/2015
Mozilla	17,927	80,061	4.5	08/1999-01/2016
GCC	13,980	85,865	6.1	03/1999-12/2015

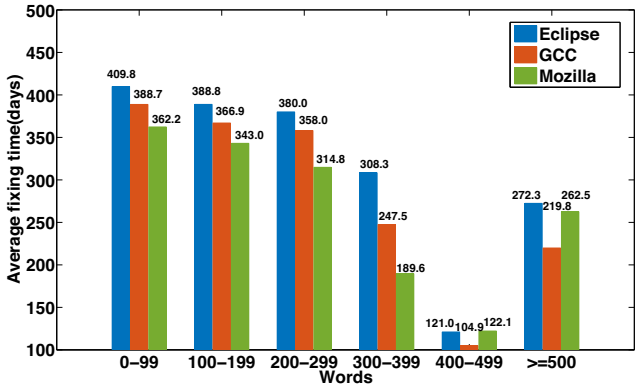


Fig. 3: The average fixing time for bug reports with different length

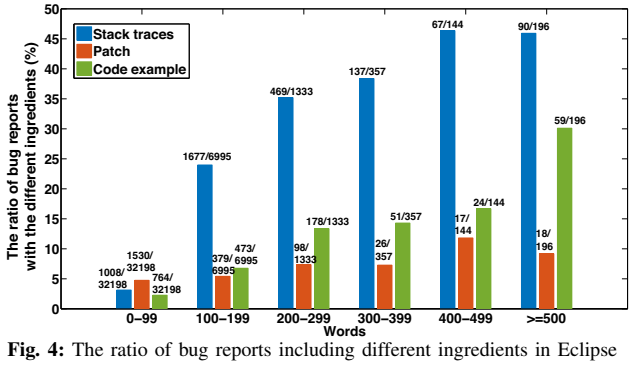


Fig. 4: The ratio of bug reports including different ingredients in Eclipse

automatically enriching bug reports can facilitate bug fixers. In Fig. 3, we find an interesting phenomenon that the fixing time increases when the bug reports include over 500 words. This finding explains that too much information may increase the developers' workload.

Fig. 3 raises another question for a short bug report: "*Why is it possible to prolong fixing time?*" According to the investigation of bug reports' quality [14], [15], the lack of resources can delay bug fixing. This result provides a clue to answer our question. Thus, we analyze the ingredients of bug reports. For the example shown in Fig. 1, bug report `#41321` contains 31 words, which is a relatively shorter bug report compared with the longer bug reports having more than 200 words[1]. We note that this bug report only contains the textual content written in natural language. Bettenburg et al. [15] summarize that a good bug report should include `stack trace(s)`, `code example(s)`, and `patch(es)` in the description part, which can speed up the bug-fixing process. Obviously, the bug report `#41321` does not contain these ingredients. It may be a major reason to cause a long fixing time (387 days) for this reported bug.

To further illustrate the reason, we use the methods in [15] to recognize the ingredients of all bug reports in our data sets and calculate the statistics. For `stack trace(s)`, we recognize them by adopting regular expressions which contain start lines and trace lines. For `code example(s)`, we can recognize declarations, conditional statements, and loops. For `patch(es)`, we also adopt regular expressions to identify which files to patch and which are the changes to make. Fig. 4 shows the ratio of bug reports that include the different ingredients (*i.e.*, `stack traces`, `patch`, and `code example`) in Eclipse. Generally, the longer bug reports include higher proportion of the above-mentioned ingredients than the shorter bug reports. For the bug reports that include more than 500 words, the proportions of `stack traces` and `patch` descend slightly. We find the similar trend in Mozilla and GCC as well, which suggests the conclusion: compared with the longer bug reports, most of the shorter bug reports (*e.g.*, the bug reports that include less than 100 words) do not contain enough important ingredients, and thus prolong the fixing time of the reported bugs. Otherwise, long bug reports containing enough ingredients decrease the fixing time. Therefore, it is possible to shorten the delay by enriching bug reports.

[1]We ask the top-10 active bug reporters who reported the largest number of bug reports from Eclipse, Mozilla, and GCC, respectively by using public mailing lists. 86.7% of the respondents response the mails and 76.7% of them think the bug reports that include more than 200 words should be treated as the longer reports.

TABLE II: The proportion of developers who think that lack of the ingredients can delay the time of fixer assignment

Task	ingredients		
	Stack trace	Code example	Patch
Fixer assignment	70.9%	**82.9%**	67.1%

Fixer assignment, which identifies a suitable person to fix the given bug, is a main task of bug triagers, who are responsible for managing bug reports [3]. Its delay will affect the process of bug fixing. We ask the top-10 active bug fixers who were assigned for fixing the largest number of bugs in Eclipse, Mozilla, and GCC, respectively via public mailing lists about the relationship between bug assignment and fixing time. 86.7% of respondents give us the responses and they all think that fixer assignment affects the time of bug fixing.

In order to demonstrate whether the bug reports that lack the ingredients delay fixer assignment, we conduct a questionnaire survey (https://www.surveymonkey.com/r/2HSDZJQ) and send it to the developers working on Eclipse, Mozilla, and GCC via mailing lists. We received 158 responses from triagers or developers with experience in fixer assignment. Table II summarizes the statistical results by analyzing these responses. Note that each developer can choose one or more than one ingredient.

The results show the proportion of developers who think that lack of the mentioned ingredient(s) can delay the time of fixer assignment. For example, 82.9% of developers think that the lack of `code example` is harmful to it. Thus, our new approach for automatically enriching bug reports can facilitate automated fixer recommendation.

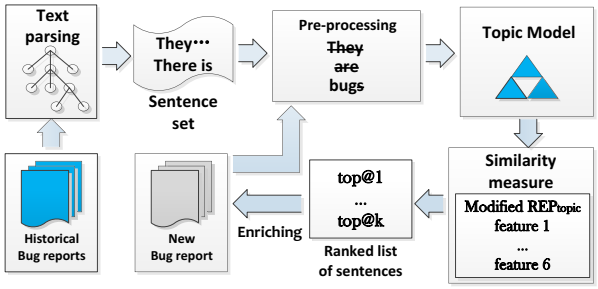


Fig. 5: The workflow of bug report enrichment

III. Methodology

This section details our new approach for bug report enrichment and how to use the enriched reports to facilitate automated fixer recommendation. Fig. 5 shows the workflow of our approach, which consists of the following steps: 1) parsing the descriptions of historical bug reports to obtain a sequence of sentences; 2) pre-processing all textual contents (summary and description) of a new bug report to be enriched and the sequence of sentences extracted from historical bug reports in our data sets described in Table I; 3) performing topic modeling to identify each sentence's topic; 4) computing the similarity between a new bug report and each candidate sentence from historical bug reports, and then ranking these sentences to find Top@K nearest neighbors (*i.e.*, additional sentences) for enriching the bug report. After that, we utilize the enriched bug reports to conduct automated fixer recommendation.

A. Text parsing for historical bug reports

To select a subset of existing sentences to enrich a bug report, we convert the descriptions of the historical bug reports into a sequence of sentences by using a widely used platform called Natural Language Toolkit (NLTK) [16]. According to the result of text parsing, we first filter out the interrogative words such as "what", "how", etc. and greeting words, for example "thank you", "congratulations", and then label each sentence via a unique `id` and record its source (*i.e.*, the bug report which includes the sentence) `id`. Finally, we output these sentences into the sentence set.

B. Pre-processing

Pre-processing is a necessary step to identify the topics of sentences via topic modeling and compute the similarity between a query (*i.e.*, a new bug report which will be enriched) and the candidate sentences. It is performed by utilizing the Nature Language Processing (NLP) techniques that include lexical processing, stemming, and stop word removal. The purpose of lexical processing is to chop a given bug report up into the words called tokens. Specifically, 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`. In addition, some tokens appearing in source code, such as keywords (*e.g.*, `int`, `class`, `public`, etc.), separators, and operators are removed. Stemming is used to reduce the derived words to their stem, base or root forms. For example, the words such as "carrying", "carried", and "carries" are changed to "carry". Stop word removal is introduced to eliminate meaningless words (*e.g.*, "to", "as", "are", etc.) for guaranteeing the effectiveness of similarity measure.

We adopt NLTK to pre-process the new bug reports and the sentences extracted from the historical bug reports.

C. Building a topic model via LDA

After parsing and pre-processing the descriptions of the historical bug reports, we utilize the topic modeling approach to find each sentence's topic. More precisely, we use the implementation of Latent Dirichlet Allocation (LDA) algorithm [17] in Stanford Topic Modeling Toolbox (TMT) [18] to process each sentence. The topics produced by TMT are used to measure the similarity between the new bug report and the candidate sentences.

In TMT, four parameters (N, R, α, β) need to be set. N and R denote the number of topics and the number of iterations, respectively. α and β are two hyper parameters, where α represents the association between the documents (*i.e.*, new bug reports and candidate sentences in our work) and the topic(s), and β indicates the association between a topic and related terms. For the parameter adjustment, we discuss the details in Section III-E.

For each sentence, TMT outputs its topic distribution and the corresponding probability value for each topic. We decide the topic of each sentence when its probability value reaches the highest.

D. Bug report enrichment via sentence ranking

To enrich a bug report, we retrieve the Top@K sentences that are strongly related to it from historical bug reports. In order to complete the retrieval task, we propose a new textual similarity metric $REP^{\#}_{topic}$, which is motivated by two metrics including REP_{topic} in [9] and REP in [19], to quantify the similarities between the bug report and the candidate sentences extracted from historical bug reports. Since both REP and REP_{topic} are used to compute the similarity between bug reports, they are not suitable for bug report enrichment, because an enriched bug report is generated by a list of ranked sentences rather than a similar bug report. $REP^{\#}_{topic}$ is based on REP_{topic} that combines five features, including the `component`, the `product`, the `topic`, the textual similarities based on textual contents represented by bags of `unigrams` and `bigrams`. More precisely, we modify three features ($feature_1$, $feature_2$, and $feature_5$) in REP_{topic}, and add a new feature ($feature_6$) to implement bug report enrichment. All features in $REP^{\#}_{topic}$ are described in Table III. Here, q (*i.e.*, query) is the bug report to be enriched; $q.n_{topic}$ shows the number of sentences in q which share the same topic with the candidate sentence s while $q.N$ represents the total number of sentences in q. In addition, textual contents stand for the contents of bug reports' summary and description or the candidate sentences after pre-processing.

For $REP^{\#}_{topic}$ based on REP_{topic}, we detail what features are modified (or added) and why make the changes as follows:

(1) Change $\rightarrow$ $feature_1$ **and** $feature_2$:

TABLE III: Features in $REP^{\#}_{topic}$

Feature	Content	Description	Range
$feature_1$	Textual similarity	Similarity between textual contents represented by bags of `unigrams` (*i.e.*, words).	[0,∞)
$feature_2$	Textual similarity	Similarity between textual contents represented by bags of `bigram` (*i.e.*, two words that appear consecutively).	[0,∞)
$feature_3$	Product	Product that is affected by the bug.	0 OR 1
$feature_4$	Component	Component that is affected by the bug.	0 OR 1
$feature_5$	Topic	Topics of textual contents via topic modelling.	0 OR $q.n_{topic}/q.N$
$feature_6$	Ingredient	A stack, a code example, or a patch existed in textual contents.	0 OR 1

Content: We adopt BM25 defined in formula (2) to compute the textual similarities between q and s instead of BM25F which is a variant of BM25.

Motivation: The candidate sentences are unstructured documents, thus BM25 is more suitable than BM25F which performs better for processing structured documents.

(2) Change $\rightarrow feature_5$

Content: We use $\frac{q.n_{topic}}{q.N}$ instead of 1 when the topic of s is the same as one of the topics of q.

Motivation: The given bug report q may belong to multiple topics due to the multiple sentences included in it, thus we consider the ratio of the sentences which belong to the same topic with s to the overall sentences in q to represent $feature_5$.

(3) Add $\rightarrow feature_6$

Content: We add a new feature, *i.e.*, $feature_6$. When the candidate sentence s is `stack traces`, `code example`, or `patch`, the value is 1; otherwise, the value is 0.

Motivation: According to the investigation results, the bug reports which lack the ingredients delay the fixing time. Thus, we introduce this feature for giving more weight to the candidate sentences that belong to one of three ingredients.

The performance comparison results shown in Section VI-D demonstrate that these *changes* are effective for bug report enrichment.

We present how to get the values of these features via Equation (1).

$$
\begin{aligned}
feature_1(q,s) &= BM25(q,s)\ //of\ unigrams\\
feature_2(q,s) &= BM25(q,s)\ //of\ bigrams\\
feature_3(q,s) &= \begin{cases} 1 & \text{if q.product=s.product}\\ 0 & \text{otherwise}\end{cases}\\
feature_4(q,s) &= \begin{cases} 1 & \text{if q.component=s.component}\\ 0 & \text{otherwise}\end{cases}\\
feature_5(q,s) &= \begin{cases} \frac{q.n_{topic}}{q.N} & \text{if q.topic=s.topic}\\ 0 & \text{otherwise}\end{cases}\\
feature_6(q,s) &= \begin{cases} 1 & \text{if s}\in S_{stack\,trace} \text{ or } S_{code} \text{ or } S_{patch}\\ 0 & \text{otherwise}\end{cases}
\end{aligned}
\tag{1}
$$

where $S_{stack\,trace}$, S_{code}, and S_{patch} represent the sentences belonging to `stack traces`, `code example`, and `patch`, respectively. $feature_1$ and $feature_2$ stand for the textual similarities via BM25 [20], [21] between q and s which are represented by bags of `unigrams` and `bigrams`, respectively. The textual similarity metric BM25 is presented as follow:

$$
\begin{aligned}
BM25(q,s) =& \sum_{t\in q\cap s} IDF(t) \times \frac{TF(t,s)(k_1+1)}{TF(t,s)+k_1(1-b+b\times\frac{l_s}{\bar{l}})}\\
&\times \frac{(k_3+1)TF(t,q)}{k_3+TF(t,q)}
\end{aligned}
\tag{2}
$$

Here, t is the shared term occurring in both q and s; $IDF(t)$ is the the inverse document frequency represented by $log(\frac{N-n_t+0.5}{n_t+0.5})$, where N is the total number of candidate sentences, n_t denotes the number of sentences containing the term t, and 0.5 is added for avoiding the situation n_t=0; $TF(t,s)$ denotes the term frequency of term t in s while $TF(t,q)$ represents the term frequency of term t in q; l_s denotes the length of sentence s in words; $\bar{l}$ refers to the average length over all the sentences in the collection. The free parameters k_1, b, and k_3 serve controlling the weight of term frequency $TF(t,s)$, the normalized document length, and the weight of term frequency $TF(t,q)$, respectively.

According to above-mentioned features, we can get $REP^{\#}_{topic}$ as follows:

$$
REP^{\#}_{topic}(q,s) = \sum_{i=1}^{6} \omega_i \times feature_i \tag{3}
$$

We note that $REP^{\#}_{topic}$ is a linear combination of six features, where ω_i is the weight for the ith feature $feature_i$ defined by formula (1). We describe how to adjust all parameters of $REP^{\#}_{topic}$ in Section III-E.

By using $REP^{\#}_{topic}$, we get a ranked list of candidate sentences. Then, we supplement Top@K sentences to the given bug report so that an enriched version is generated.

E. Parameter adjustment

As mentioned in Section III-D, we should adjust the parameters in $REP^{\#}_{topic}$. We present how to adjust these parameters in the following paragraphs.

For the feature *topic* which is produced by topic modelling, we adjust the parameters used in TMT, which is a topic modeling tool introduced in Section III-C. We set the parameters, including R, α, and β, to their default values (*i.e.*, 100, 0.01, 0.01) respectively, and adjust the number of topics N from 10 to 100, with a step-size of 10.

The similar measure $REP^{\#}_{topic}$ defined in formula (3) has 12 free parameters in total. For $feature_1$ and $feature_2$, we compute textual similarities of q and s by using *BM25*. Computing each of two features needs 3 free parameters (*i.e.*, k_1, b, and k_3). In addition, in formula (3), there are 6 weight factors for the corresponding 6 features. Thus, $REP^{\#}_{topic}$ requires $(2\times 3+6)=12$ parameters to be set.

Table IV shows the parameters and their inital values of $REP^{\#}_{topic}$ in columns 1 and 2, respectively. We follow the

TABLE IV: Parameters in $REP^{\#}_{topic}$

Parameter	Init.	Selected parameter values per project		
		Eclipse	Mozilla	GCC
N: the number of topics	10	20	20	20
ω_1: weight of $feature_1$	0.9	0.021	0.007	0.200
ω_2: weight of $feature_2$	0.2	0.430	0.002	0.150
ω_3: weight of $feature_3$	2	2.200	3.810	0.250
ω_4: weight of $feature_4$	0	1.540	2.540	0.170
ω_5: weight of $feature_5$	0	0.660	0.003	0.150
ω_6: weight of $feature_6$	0	1.320	0.830	0.220
$b^{unigram}$: b in $feature_1$	1	0.840	0.800	1.880
$k_1^{unigram}$: k_1 in $feature_1$	2	2.000	2.000	2.000
$k_3^{unigram}$: k_3 in $feature_1$	0	0.090	0.122	0.030
b^{bigram}: b in $feature_2$	1	1.110	0.800	2.390
k_1^{bigram}: k_1 in $feature_2$	2	2.000	2.000	2.000
k_3^{bigram}: k_3 in $feature_2$	0	0.380	0.437	0.110

same parameter tuning method (*i.e.*, gradient descent algorithm) used in the previous studies [19], [22] to verify the parameter values in $REP^{\#}_{topic}$. Specifically, when the number of topics is initialized (*e.g.*, N=10), we start to adjust all 12 parameters in $REP^{\#}_{topic}$ using gradient descent algorithm.

Given each of these parameters x, we initialize it with a default value. For $feature_{1-4}$, we follow the same initial values recommended by Sun et al. [19]; for $feature_{5-6}$, we set their weight vectors ω_5 and ω_6 to 0 as their initial values. Then, we run the iterative adjustment of the value of x so that the value of the RankNet Cost function (RNC) [23], [24] reaches the *minimum*. RNC is defined by $RNC(I) = log(1+e^Y)$ where I denotes a training instance (q, s_{rel}, s_{irr}), and Y is presented as $Y = sim(s_{irr}, q) - sim(s_{rel}, q)$. Here, s_{irr} is an irrelevant sentence with a query q (*i.e.*, a new bug report) while s_{rel} means a relevant sentence with q. Tian et al. [22] treated the duplicate bug reports and the non-duplicate bug reports as the relevant documents and irrelevant documents, respectively. We also adopt the similar way to compute RNC. In detail, the irrelevant sentences are extracted from the non-duplicate bug reports while the relevant sentences come from the duplicate bug reports.

At the process of the iterative adjustment, we first fix the values of b, k_1 and k_3 in $feature_{1-2}$ to their initial values, and then tune the unfixed parameters ω_1 to ω_6 to find their best values. Next, we fix the best values of ω_1 to ω_6, and then tune the unfixed parameters b, k_1 and k_3 to find the most appropriate values. For the details of the iterative adjustment using gradient descent, please refer to the previous study [19]. We list the parameter values selected for performing $REP^{\#}_{topic}$ in the columns 3-5 of Table IV.

IV. Research questions for evaluation

We evaluate the enriched bug reports from two aspects. First, we evaluate their usefulness for manual fixer assignment (*i.e.*, RQ1). Second, we assess their effectiveness and features' influence on automated fixer recommendation (*i.e.*, RQ2-RQ3). Moreover, we compare the performance of $REP^{\#}_{topic}$ and REP_{topic} when they are used in the fixer recommendation approaches (*i.e.*, RQ4). We answer RQ1 in Section V and RQ2-4 in Section VI, respectively.

- **RQ1-Human evaluation:** ***Can the enriched bug reports provide further help to fixer assignment?***
  Details: To evaluate whether the enriched bug reports can provide further help to triagers when they assign bugs to fixers, we invite the real bug triagers for Eclipse, Mozilla, and GCC to evaluate whether the additional sentences are related to the corresponding original bug reports and can improve the efficiency of fixer assignment.
- **RQ2-Effectiveness evaluation:** ***Can the enriched bug reports improve the performance of the automated fixer recommendation in bug repositories?***
  Details: To evaluate whether the enriched bug reports can improve the performance of automated fixer recommendation, we perform three approaches of automated fixer recommendation based on the original bug reports and the enriched version. By comparing the results, we verify whether our enrichment method can facilitate automated fixer recommendation.
- **RQ3-Feature influence:** ***Which feature(s) in $REP^{\#}_{topic}$ has(have) the greatest impact on bug report enrichment for improving the automated fixer recommendation?***
  Details: There are six features in $REP^{\#}_{topic}$, thus it would be interesting to know which feature(s) has(have) the greatest impact on bug report enrichment. We perform $REP^{\#}_{topic}$ to implement bug report enrichment by removing each feature one by one, respectively, then we compare the performance and verify which feature(s) has(have) the greatest impact.
- **RQ4-Performance comparison:** ***Does $REP^{\#}_{topic}$ outperform REP_{topic} in terms of enriching bug reports for improving the automated fixer recommendation?***
  Details: Since $REP^{\#}_{topic}$ is based on REP_{topic}, we compare their effectiveness on bug report enrichment by using them in bug report enrichment. Then, we compare the performance of automated fixer recommendation using different metrics (i.e., $REP^{\#}_{topic}$ or REP_{topic}).

V. Enriched bug report for fixer assignment

This section details the evaluation of enriched bug report's usefulness for manual fixer assignment.

A. Experiment setup

We collected the bug reports from the bug repositories of Eclipse, Mozilla, and GCC, and the details of our data sets are shown in Table I. On average, each bug report includes 4.9, 4.5, and 6.1 sentences for three projects, respectively. For each bug report, we extracted the sentences from the descriptions of the historical bug reports to generate its enriched version. To analyze and evaluate the enriched bug reports, we employ a folding-based training and validation approach [15] to conduct the experiment for each project because Bettenburg et al. indicate that this method can achieve higher prediction accuracy when it is used to training data. In detail, we divide all bug reports into 11 equally-sized folds in chronological order. We execute 10 rounds evaluation with these 11 folds. In the i^{th} round, the first i folds are formed as a training set (historical bug reports) and the $(i+1)^{th}$ fold is used as a test set. When all 10 rounds are finished, we can get the enriched versions of all bug reports in the test sets.

<bug_id>41321<bug_id>
TOP-1:
Ada runtime missing floating point and error handler initialization on OpenBSD
TOP-2:
"Ada runtime (gcc trunk, r131220) is missing floating point and error handler initialization on OpenBSD."
TOP-3:
../../gcc/gcc/ada/init.c: In function '__gnat_error_handler':
../../gcc/gcc/ada/init.c:444:56: error: unused parameter 'ucontext'
make[3]: *** [ada/init.o] Error 1
TOP-4:
However, gcc occasionally
evaluates hexadecimal (and possibly decimal) floating-point constants
incorrectly starting with version 3.4:

```
das@VARK: cat bar.c
#include stdio.h
int main(int argc, char *argv[]) {
    volatile long double d = 0x3.243f6a8885a31p0L;
    printf("%La\n", d);
    d = 2.0 * d / 2.0;
    printf("%La\n", d);
    return (0);
}
```

TOP-5:
soft float patch
--- gcc-3.3-branch/gcc/config/arm/elf.h 2004-03-30 12:43:45.000000000 -0800
+++ gcc-3.3.3-fg/gcc/config/arm/elf.h 2004-04-07 21:42:05.000000000 -0700
••• •••
TOP-30:
[Ada] Compiler assertion in iterator

Fig. 6: An enhanced version of bug report `41321`

B. A case study of enriched bug report

We show an enriched version of bug report #`41321` in Fig. 6. Note that we only indicate the additional sentences, and the original summary and description of bug report #`41321` are shown in Fig. 1.

Note that in this case, the 1st and the 2nd sentences further explain the initialization error of Ada runtime; the 3rd sentence indicates where the bug is; the 4th sentence provides a `code example` (Note that we treat a whole piece of code as one sentence); and the 5th sentence denotes a `patch`. Obviously, these sentences are related to the original bug report which does not contain the additional sentences. For the 30th sentence, note that there is no evidence to demonstrate whether this sentence is related to the original bug report. Therefore, by adopting our approach, the important details and related ingredients in higher ranked sentences are added into original bug reports like #`41321` so that they are enriched.

C. Answer to RQ1: Human evaluation

To evaluate whether the enriched bug reports can help developers improve the efficiency of fixer assignment, we randomly selected 300 pairs of bug reports from Eclipse, Mozilla, and GCC (100 pairs per project) as the evaluation objects. Each pair includes a bug report and its enriched version with the Top@30 additional sentences. Then we sent each pair of bug reports to the corresponding real triager by the email (*i.e.*, totally 300 emails) to conduct human evaluation. Bug triagers are responsible for fixer assignment, thus only they can verify whether the enriched bug reports can facilitate this task. In detail, we first investigate the change history to verify who is responsible for assigning the fixer, and then invite the triager to participate in our survey. We ask each triager to answer the two questions (*i.e.*, *Q1* and *Q2*) for the Top@30 additional sentences of each bug report assigned by her/him. To save respondents' time, we divide the additional sentences into 6 groups, and each group contains 5 sentences shown in the second row of Table V. Thus, respondents need only check each group to answer the following questions.

TABLE V: Human evaluation for enriched bug reports

Question	The average score (standard deviation) for Top@30 additional sentences					
	T1-5	T6-10	T11-15	T16-20	T21-25	T26-30
Q1	3.99(1.11)	3.88(1.14)	3.65(1.08)	3.06(1.21)	2.66(1.11)	2.46(0.92)
Q2	4.27(0.78)	4.00(1.10)	3.78(1.19)	3.10(1.32)	2.82(1.07)	2.52(0.90)

- *Q1: Are the additional sentences related to the original bug report?*
- *Q2: Can the additional sentences help you improve the efficiency of fixer assignment?*

In order to facilitate the analysis for the investigation results, we request the triagers to choose the score s from 1 to 5 as the answer for each question. The closer s is to 1, the more likely the answer tends to be negative. Conversely, the closer s is to 5, the more likely the answer tends to be positive. As a result, we received 216 responses (*i.e.*, 72% of response rate) from 48 triagers[2] in the three projects. We list the statistical results in Table V.

In this table, we show the average scores provided by the respondents and the corresponding standard deviation. Note that for both of the two questions, the average scores for the Top@1-5 additional sentences are the highest while they are lowest for the Top@26-30 additional sentences. In addition, the data is not much more spread out because all standard deviations are not high. By analyzing the results, we know that the triagers think the additional sentences ranked ahead are related to the original bug report and may be useful for fixer assignment. We perform the experiments (See Section VI) to further demonstrate whether the enriched bug reports have the actual usefulness in practise.

According to the results of human evaluation, we obtain the following answer to RQ1:

> **Answer to RQ1**: The additional sentences ranked ahead in the enriched bug reports are related to the original bug reports. They may provide more help for fixer assignment.

VI. Enriched Bug Report for Fixer Recommendation

Automated fixer recommendation is one of triagers' major tasks, which influences the fixing time according to our investigation result described in Section II. This section details the evaluation of whether the enriched bug report can facilitate automated fixer recommendation.

A. Experiment setup

To evaluate the effectiveness of enrichment on automated fixer recommendation, we also employ the folding-based training and validation approach [15] to conduct the following experiments for bug reports collected from the same data sets

[2]Note that some triagers are responsible for processing more than one bug reports.

with RQ1 and their enriched versions. We perform this task on each round and calculate the average results of all the 10 rounds. Therefore, for all the experiments presented in this section, all the results are average output. We will not emphasize it in the following subsections.

We adopt Precision ($\frac{TP}{TP+FP}$), Recall ($\frac{TP}{TP+FN}$), and F-measure ($2 \times \frac{Precision \times Recall}{Precision+Recall}$) [25] for performance evaluation, where TP (*i.e.*, True Positive instances) denotes the number of instances (*i.e.*, 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 the approach.

B. Answer to RQ2: Effectiveness evaluation for automated fixer recommendation

The purpose of fixer recommendation is to assign an appropriate developer to fix the given bug. To reduce the triagers' workload, automated approaches have been proposed.

In this paper, we utilize the enriched bug reports to re-implement three automated fixer recommendation approaches in our data sets. **DREX**[3] [3] adopted social network metrics (*e.g.*, out-degree) to rank the candidate developers. **DRETOM** [4] utilized topic model to recommend potentioal developers. **DevRec** [5] combined social network and topic model to perform automated fixer recommendation. These approaches used the different factors related to bug reports to recommend appropriate bug fixers, thus we select them to verify whether the enriched bug reports are effective for multiple factors adopted by automated algorithms on fixer recommendation.

The evaluation results when we recommend top-10 fixers are shown in Table VI. Due to limited space, we present Precision@10 scores and Recall@10 scores at https://github.com/BREnrich/Enriching-Bug-Report, and only show F-measures in Table VI. Note that the values in parentheses indicate the differences of F-measures between enriched bug reports and their original versions. Note that the enriched bug reports can further improve the performance of automated fixer recommendation approaches. For example, in Eclipse, using the enriched versions of the bug reports via adding Top@20 sentences improves 13.37% of F-measure scores when adopting DRETOM. Overall, the greatest improvements for three projects come to 10%, 13.37%, and 8% when using DREX, DRETOM, and DevRec, respectively. It is worth noting that the performance improvement decreases when the number of added sentences exceeds a threshold. For instance, in GCC, when performing DREX, if adding 10 sentences (more than 5 sentences), the improvement of F-measure decreases to 3.49%. Thus, supplementing unconcerned sentences may degrade the performance of fixer recommendation.

To further demonstrate whether enriching bug reports can significantly improve the performance of automated fixer recommendation approaches, we perform a t-test [26] in the R environment [27] due to the normal distribution of the data, *i.e.*, the normality values calculated by Shapiro-Wilk normality test [28] are more than 0.05. In detail, we define the null hypothesis: *Enriching bug reports shows no noteworthy difference against original bug reports by utilizing the approaches for recommending bug fixers.* Then, we introduce the F-measure values of three projects for each approach shown in Table VI as the input data when performing t-test. The result is also stored at https://github.com/BREnrich/Enriching-Bug-Report. For DREX, the p-values are lower than 0.05 when adding the Top@1-Top@5 sentences. For DRETOM, when supplementing the Top@5 and Top@10 sentences, the p-values are 0.01361 and 0.04927, respectively, which are lower than 0.05. For DevRec, the p-values are lower than 0.05 when adding the Top@10-25 sentences. In these situations, we reject the above null hypothesis. Thus, enriching bug reports can provide the *significant* improvement for the performance of automated fixer recommendation approaches when we supplement the *appropriate* number of sentences. If developers use our approach to execute automated software maintenance tasks, we suggest that they can decide the number of additional sentences when the enriched bug reports can reach the highest efficiency.

TABLE VI: Performance comparison on automated fixer recommendation approaches when supplementing different ratio of sentences

Project	Top@K	F-measure for different approaches (%)		
		DREX	DRETOM	DevRec
Eclipse	Original	20.75	21.19	28.86
	Top@1	24.42(3.67)	26.84(5.65)	29.52(0.66)
	Top@5	25.57(4.82)	30.92(9.73)	29.93(1.07)
	Top@10	26.21(5.46)	33.42(12.23)	32.46(3.60)
	Top@15	26.49(5.74)	33.79(12.60)	34.25(5.39)
	Top@20	26.68(5.93)	**34.56(13.37)**	35.68(6.82)
	Top@25	**26.72(5.97)**	32.56(11.37)	**36.61(7.75)**
	Top@30	23.64(2.89)	30.00(8.81)	34.61(5.75)
Mozilla	Original	22.59	22.53	33.18
	Top@1	26.84(4.25)	24.43(1.90)	34.00(0.82)
	Top@5	31.65(9.06)	29.07(6.54)	36.95(3.77)
	Top@10	32.35(9.76)	30.45(7.92)	38.39(5.21)
	Top@15	**32.59(10.00)**	**31.76(9.23)**	39.07(5.89)
	Top@20	32.57(9.98)	31.12(8.59)	**39.15(5.97)**
	Top@25	32.55(9.96)	30.66(8.13)	39.53(5.35)
	Top@30	32.38(9.79)	29.79(7.26)	39.31(6.13)
GCC	Original	21.53	32.02	28.31
	Top@1	24.63(3.10)	36.60(4.58)	30.67(2.36)
	Top@5	**27.44(5.91)**	**39.62(7.60)**	**36.31(8.00)**
	Top@10	25.02(3.49)	37.51(5.49)	35.03(6.72)
	Top@15	24.97(3.44)	37.05(5.03)	34.75(6.44)
	Top@20	24.61(3.08)	36.68(4.66)	34.23(5.92)
	Top@25	24.50(2.97)	36.42(4.40)	34.06(5.75)
	Top@30	24.12(2.59)	36.28(4.26)	33.93(5.62)

By analyzing the evaluation results of automated fixer recommendation using the enriched bug reports, we can answer RQ2 as follow:

> **Answer to RQ2**: The proposed enrichment approach for bug reports can be used to improve the performance of automated fixer recommendation in bug repositories. However, adding the low-ranked sentences may influence the performance.

C. Answer to RQ3: Feature influence for performance

$REP^{\#}_{topic}$ defined in formula (3) involves six features. In order to verify the influence of each feature for the performance of automated fixer recommendation approaches, we conduct the experiment to compare the F-measure values by using the enriched bug reports when we remove each feature one by one. If the value is the minimum, the corresponding feature has the

[3]We only adopt Out-degree to implement DREX because it performed the best among all social network metrics.

greatest impact. Table VII shows the comparison results. The data on the first row show the F-measure values using original bug reports, the data on the second row show the F-measure values using enriched bug reports via $REP^{\#}_{topic}$, and the data on the third to the eighth rows show the F-measure values when we set the weight of each feature to 0 in $REP^{\#}_{topic}$ to enrich bug reports, respectively. Specifically, we show the best values in the second row, and we keep the same parameter values (except the weight of the removed feature is set to 0) in $REP^{\#}_{topic}$ to get the data on the third to the eighth rows.

In Table VII, when we remove $feature_3$ (*i.e.*, `Product`) in $REP^{\#}_{topic}$ to perform automated fixer recommendation approaches on Eclipse, the F-measure scores decrease to 22.33%, 28.95%, and 33.12% for DREX, DRETOM, and DevRec, respectively. In this case, we note that using enriched bug reports still improves the performance of three approaches using original bug reports, but the decrease is the highest. This finding shows that $feature_3$ has the greatest impact among all features. By the same token, we find that which features have the greatest impact to enrich bug reports for implementing the automated fixer recommendation approaches on Mozilla and GCC. Thus, we can answer RQ3 as follow:

> **Answer to RQ3**: In Eclipse, $feature_3$ in $REP^{\#}_{topic}$ has the greatest impact to enrich bug reports for implementing the three automated fixer recommendation approaches such as DREX, DRETOM, and DevRec. In Mozilla, $feature_3$, $feature_4$ (*i.e.*, `Component`), and $feature_1$ (*i.e.*, `Textual similarity`) have the greatest impact when we implement DREX, DRETOM, and DevRec, respectively. In GCC, $feature_1$ has the greatest impact to perform DRETOM and DevRec, and $feature_3$ has the greatest impact to implement DREX.

By analyzing the feature influence, we find an interesting phenomenon. As a popular information retrieval model, topic modelling is used to produce the topics as $feature_5$ (*i.e.*, `Topic`) in $REP^{\#}_{topic}$ for improving the performance of bug report enrichment. However, it does not show the most significant influence. Therefore, the performance improvement caused by our approach is not only due to topic modelling, but also because of the features' combination in $REP^{\#}_{topic}$.

We note that feature influence is affected by different data sets and different approaches. We think the major reason is due to the different data distributions in data sets and the different weight of factors in approaches. It would be an interesting topic in the future work.

D. Answer to RQ4: Performance comparison between $REP^{\#}_{topic}$ and REP_{topic}

Since $REP^{\#}_{topic}$ is based on REP_{topic} (Section III-D) [9], we compare the performance of automated fixer recommendation approaches using different similarity metrics. Specifically, we use the same parameter values as REP_{topic} in $REP^{\#}_{topic}$ and only consider the description as the sole field in BM25 used in $feature_1$ and $feature_2$. In the last row of Table VII, we show the comparison result. We note that the performance of using $REP^{\#}_{topic}$ is better than that of using REP_{topic}. For instance, when we use REP_{topic} to generate enriched bug reports in Mozilla, the F-measure values reach up to 29.74% for DREX, 29.49% for DRETOM, and 34.62% for DevRec, which are lower than the performance using $REP^{\#}_{topic}$. Thus, we can answer RQ4 as follow:

> **Answer to RQ4**: The performance of automated fixer recommendation approaches using $REP^{\#}_{topic}$ to enrich bug reports is better than using REP_{topic}.

VII. Threats to validity

A. Generality

In this paper, we only collect the bug reports from three open source projects to perform the experiments. We are not sure that the enriched bug reports can also improve the performance of automated fixer recommendation in commercial projects, because the scale, the size, and the textual contents of bug reports in commercial projects are different from open source projects. However, if the bug repositories of the commercial projects meet some conditions (*e.g.*, a large number of historical bug reports) of large-scale open source projects, our approach can also be applied to them.

In addition, we only select three approaches (*i.e.*, DREX, DRETOM, and DevRec) of fixer recommendation to verify the effectiveness of bug report enrichment. These approaches utilize the different recommendation models, therefore we adopt them to verify whether the enrichment approach is effective for a variety of approaches on fixer recommendation. However, we also should implement more fixer recommendation models and more automated software maintenance tasks such as bug localization. We plan to achieve the further verification in the extended work.

B. Parameter adjustment

Our approach needs tuning 12 parameters to achieve the best performance. We have adopted the parameter adjustment methods proposed in Sun et al. [19] and Tian et al. [22] for $REP^{\#}_{topic}$ and the algorithms used for verifying the best values used to perform bug report enrichment. In future work, we will develop an effective tuning method.

C. Readability of enriched bug reports

In human evaluation, we only focus on whether the additional sentences are related to original bug reports and whether they can provide further help when using them to assign bug fixers. Obviously, the high-level readability is also an important factor because a good logical bug report can instruct inexperienced developers to understand how the bug happens. In future work, we will develop a sorting algorithm to arrange the order for each candidate sentence so that we can get enriched bug reports with high readability.

VIII. Related work

A. Automated summarization for bug reports

As an early-stage work, Rastkar et al. [29] found that existing conversation-based generators can produce more accurate summaries of given bug reports than random generators. By comparing with EC (Email classifier), EMC (Email-Meeting

TABLE VII: Performance (F-measure %) comparison using Original Bug Reports (OBR) and Enriched Bug Reports (EBR) with different variants of $REP^{\#}_{topic}$

Variants	F-measure (%) of automated fixer recommendation approaches with the different variants of $REP^{\#}_{topic}$								
	Eclipse			Mozilla			GCC		
	DREX	DRETOM	DevRec	DREX	DRETOM	DevRec	DREX	DRETOM	DevRec
OBR	20.75	21.19	28.86	22.59	22.53	33.18	21.53	32.02	28.31
EBR+$REP^{\#}_{topic}$	26.72	34.56	36.31	32.59	31.76	39.53	27.44	39.62	36.31
EBR+$REP^{\#}_{topic}(\omega_1 = 0)$	26.09	31.44	33.47	30.00	28.62	**34.19**	25.24	**30.94**	**32.29**
EBR+$REP^{\#}_{topic}(\omega_2 = 0)$	25.55	31.56	33.52	30.89	30.02	35.54	25.52	34.62	32.42
EBR+$REP^{\#}_{topic}(\omega_3 = 0)$	**22.33**	**28.95**	**33.12**	**28.00**	28.58	39.22	**21.82**	32.25	32.49
EBR+$REP^{\#}_{topic}(\omega_4 = 0)$	24.74	30.99	33.45	30.41	**28.46**	38.15	25.82	36.13	34.22
EBR+$REP^{\#}_{topic}(\omega_5 = 0)$	26.12	32.85	35.40	30.79	31.53	39.46	25.61	37.19	34.34
EBR+$REP^{\#}_{topic}(\omega_6 = 0)$	26.03	31.79	33.39	30.51	30.92	36.10	25.92	36.99	32.82
EBR+REP_{topic}	26.48	29.02	33.96	29.74	29.49	34.62	23.03	37.07	32.65

classifier) and BRC (Bug Report classifier), they found that BRC performed the best among them. Different from supervised learning approaches [29] , Mani et al. [30] utilized four unsupervised approaches such as Centroid, Maximum Marginal Relevance, Grasshopper and Diverse Rank to summarize the given bug reports. Rastkar et al. [31] extended their previous work in. Except for demonstrating BRC performed better than EC and EMC, they utilized these generated summaries to execute the duplicate bug reports detection task.

The goal of our study is different from automated summarization for bug report. Even if we use sentence extraction to perform this work, the sentences ranking algorithm based on $REP^{\#}_{topic}$ is different from previous supervised learning approaches (*e.g.*, BRC) and unsupervised learning methods (*e.g.*, Diverse Rank). We do not need to annotate the sentences in historical bug reports, and it is not necessary to adopt a noise reducer. The proposed approach is expected to reduce the cost of the manual annotation and help developers to execute the automated software maintenance tasks effectively.

B. Automated fixer recommendation

Čubranić and Murphy used Naïve Bayes to train a classifier to recommend appropriate bug fixer [1]. Anvik et al. [2] used SVM to provide the best appropriate developers to a human triager for executing a bug fixing task. Wu et al. [3] proposed a developer recommendation approach called Developer Recommendation with K-Nearest-Neighbor Search and Expertise Ranking (DREX). Xie et al. [4] proposed DRETOM to model the topics for grouping the bug reports which share the same topic(s) and analyzed the developers' interests and experiences on the bug reports belonging to the corresponding topic(s) in the past fixing records so that it can work well for recommending the appropriate fixers. Xia et al. [5] proposed DevRec algorithm which performed bug report-based and developer-based analysis to recommend the bug fixers. In this paper, we utilized enriched bug reports to execute DREX, DRETOM, and DevRec, and demonstrate the enrichment can improve their performance.

Expect bug reports, a number of studies [6]–[8], [10] used other information sources (*e.g.*, commits and source code comments) or specialized models to recommend appropriate developers. In this work, we only consider bug reports as our research object.

Although $REP^{\#}_{topic}$ is based on REP_{topic} (Section III-D) [9], there are obvious differences between this work and the study in [9]. First, the purpose is to rank candidate sentences to enrich bug reports by using the modified REP_{topic} rather than automated fixer recommendation; second, due the different purpose, we modify the features ($feature_1$,$feature_2$, and $feature_5$) and add the new feature $feature_6$ in REP_{topic} to generate $REP^{\#}_{topic}$.

C. Bug report management

Except for automated bug report summarization and fixer recommendation, some researchers devote continuously to bug report management. Existing works [14], [15] aimed to improve the quality of bug reports; some studies [19], [32]–[34] focused on detecting duplicate bug reports, and the study [35] is proposed to find high-impact bug reports; a number of studies [22], [36]–[39] have been proposed to reassign the fields (*e.g.*, severity, reopen) of bug reports; and a number of other studies [20], [40], [41] locate source code relevant to a bug report.

IX. Conclusion and Future Work

In this paper, we investigate the necessity of bug report enrichment by analyzing the reason why short bug reports can delay the fixing time. Then, we proposed a novel approach to enrich bug reports for providing developers more rich information, especially for the important ingredients. The result of human evaluation indicates that the enriched bug reports can improve the efficiency of fixer assignment. The evaluation results of using the enriched bug reports in automated fixer recommendation show that they can provide more information and therefore improve the accuracy of automated fixer recommendation.

In future work, we will consider other factors such as comments, history log, and commit message to further enrich bug reports. Moreover, we plan to enrich other software artifacts (*e.g.*, emails and source code files) so that these artifacts can help developers perform more software maintenance tasks.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China under Grant 61602258 and Grant 61202396, in part by Hong Kong GRF (No. PolyU 5389/13E, 152279/16E), in part by the HKPolyU Research Grants (No. G-UA3X, G-YBJX), in part by the Shenzhen City Science and Technology R&D Fund under Grant JCYJ20150630115257892, and in part by the China Postdoctoral Science Foundation under Grant 2015M582663.

REFERENCES

[1] D. Čubranić and G. C. Murphy, "Automatic bug triage using text categorization," in *SEKE'04*, 2004, pp. 92–97.

[2] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *ICSE'06*, 2006, pp. 361–370.

[3] W. Wu, W. Zhang, Y. Yang, and Q. Wang, "Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking," in *APSEC '11*, 2011, pp. 389–396.

[4] X. Xie, W. Zhang, Y. Yang, and Q. Wang, "Dretom: Developer recommendation based on topic models for bug resolution," in *PROMISE '12*, 2012, pp. 19–28.

[5] X. Xia, D. Lo, X. Wang, and B. Zhou, "Dual analysis for recommending developers to resolve bugs," *Journal of Software: Evolution and Process*, vol. 27, no. 3, pp. 195–220, 2015.

[6] M. Linares-Vásquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk, "Triaging incoming change requests: Bug or commit history, or code authorship?" in *ICSM'12*, 2012, pp. 451–460.

[7] K. Huzefa, G. Malcom, P. Denys, and H. Maen, "Assigning change requests to software developers," *Journal of Software Evolution & Process*, vol. 24, no. 1, pp. 3–33, 2012.

[8] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, "Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation," in *MSR'13*, 2013, pp. 2–11.

[9] T. Zhang, J. Chen, G. Yang, B. Lee, and X. Luo, "Towards more accurate severity prediction and fixer recommendation of software bugs," *Journal of Systems and Software*, vol. 117, pp. 166–184, 2016.

[10] X. Xia, D. Lo, Y. Ding, J. M. Al-Kofahi, T. N. Nguyen, and X. Wang, "Improving automated bug triaging with specialized topic model," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 272–297, 2017.

[11] "Eclipse bug tracking system," https://bugs.eclipse.org/bugs.

[12] "Mozilla bug tracking system," https://bugzilla.mozilla.org.

[13] "Gcc bug tracking system," https://gcc.gnu.org/bugzilla.

[14] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *ASE '07*, 2007, pp. 34–43.

[15] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *FSE '08*, 2008, pp. 308–318.

[16] "Natural language toolkit," https://www.nltk.org.

[17] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[18] "Stanford topic modeling toolbox," http://nlp.stanford.edu/software/tmt/tmt-0.4.

[19] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *ASE '11*, 2011, pp. 253–262.

[20] R. Saha, M. Lease, S. Khurshid, and D. Perry, "Improving bug localization using structured information retrieval," in *ASE'13*, 2013, pp. 345–355.

[21] C.-Z. Yang, H.-H. Du, S.-S. Wu, and X. Chen, "Duplication detection for software bug reports based on bm25 term weighting," in *TAAI'12*, 2012, pp. 33–38.

[22] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *WCRE '12*, 2012, pp. 215–224.

[23] M. Taylor, H. Zaragoza, N. Craswell, S. Robertson, and C. Burges, "Optimisation methods for ranking functions with multiple parameters," in *CIKM '06*, 2006, pp. 585–593.

[24] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, "Learning to rank using gradient descent," in *ICML '05*, 2005, pp. 89–96.

[25] C. Goutte and E. Gaussier, "A probabilistic interpretation of precision, recall and f-score, with implication for evaluation," in *ECIR '05*, 2005, pp. 345–359.

[26] C. A. Boneau, "The effects of violations of assumptions underlying the t test." *Psychological bulletin*, vol. 57, no. 1, pp. 49–64, 1960.

[27] R. C. Team, "R: A language and environment for statistical computing. r foundation for statistical computing, vienna, austria," 2014.

[28] L. M. Surhone, M. T. Timpledon, S. F. Marseken, S. (statistics), and M. Wilk, *ShapiroWilk Test*. Betascript Publishing, 2010.

[29] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: A case study of bug reports," in *ICSE'10*, 2010, pp. 505–514.

[30] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey, "Ausum: Approach for unsupervised bug report summarization," in *FSE'12*, 2012, pp. 1–11.

[31] S. Rastkar, G. C. Murphy, and G. Murray, "Automatic summarization of bug reports," *IEEE Transactions on Software Engineering*, vol. 40, no. 4, pp. 366–380, 2014.

[32] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *ICSE'07*, 2007, pp. 499–510.

[33] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *ICSE'10*, 2010, pp. 45–54.

[34] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun, "Combining word embedding with information retrieval to recommend similar bug reports," in *ISSRE'16*, 2016, pp. 127–137.

[35] X.-L. Yang, D. Lo, X. Xia, Q. Huang, and J.-L. Sun, "High-impact bug report identification with imbalanced learning strategies," *Journal of Computer Science and Technology*, vol. 32, no. 1, pp. 181–198, 2017.

[36] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *MSR '10*, 2010, pp. 1–10.

[37] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *CSMR '11*, 2011, pp. 249–258.

[38] X. Xia, D. Lo, E. Shihab, and X. Wang, "Automated bug report field reassignment and refinement prediction," *IEEE Transactions on Reliability*, vol. 65, no. 3, pp. 1094–1113, 2016.

[39] X. Xia, D. Lo, E. Shihab, X. Wang, and B. Zhou, "Automatic, high accuracy prediction of reopened bugs," *Automated Software Engineering*, vol. 22, no. 1, pp. 75–109, 2015.

[40] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *ICSE'12*, 2012, pp. 14–24.

[41] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE Transactions on Software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.