Recommending Frequently Encountered Bugs

Yun Zhang*, David Lo§, Xin Xia‡, Jing Jiang†, Jianling Sun*
*College of Computer Science and Technology, Zhejiang University, China
§School of Information Systems, Singapore Management University, Singapore
‡Faculty of Information Technology, Monash University, Melbourne, Australia
†State Key Laboratory of Software Development Environment, Beihang University, Beijing, China

ABSTRACT

Developers introduce bugs during software development which reduce software reliability. Many of these bugs are commonly occurring and have been experienced by many other developers. Informing developers, especially novice ones, about commonly occurring bugs in a domain of interest (e.g., Java), can help developers comprehend program and avoid similar bugs in the future. Unfortunately, information about commonly occurring bugs are not readily available. To address this need, we propose a novel approach named RFEB which recommends frequently encountered bugs (FEBugs) that may affect many other developers. RFEB analyzes Stack Overflow which is the largest software engineering-specific Q&A communities. Among the plenty of questions posted in Stack Overflow, many of them provide the descriptions and solutions of different kinds of bugs. Unfortunately, the search engine that comes with Stack Overflow is not able to identify FEBugs well. To address the limitation of the search engine of Stack Overflow, we propose RFEB which is an integrated and iterative approach that considers both relevance and popularity of Stack Overflow questions to identify FEBugs. To evaluate the performance of RFEB, we perform experiments on a dataset from Stack Overflow which contains more than ten million posts. We compared our model with Stack Overflow's search engine on 10 domains, and the experiment results show that RFEB achieves the average $NDCG_{10}$ score of 0.96, which improves Stack Overflow's search engine by 20%.

ACM Reference Format:
Yun Zhang*, David Lo§, Xin Xia‡, Jing Jiang†, Jianling Sun*. 2018. Recommending Frequently Encountered Bugs. In *Proceedings of IEEE/ACM International Conference on Program Comprehension 2018 (ICPC'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

During the software development process, developers may encounter different kinds of bugs which would affect a program's functionality, cause the program to crash or freeze, and reduce software reliability. Many bugs are commonly occurring and have been experienced by many other developers [5, 12, 22, 39, 44, 50, 52]. We refer to these bugs as *frequently encountered bugs* (FEBugs), i,e., generic bugs that have been encountered frequently in many situations and may affect many developers. Informing developers about FEBugs may help them avoid similar bugs and better comprehend program in the future. This is especially so for novice developers who are in the beginning of their learning process; understanding FEBugs in their domain of interest can help them cultivate good coding habits, improve the ability of program comprehension and avoid common pitfalls. Unfortunately, information about FEBugs are not readily available; people often do not know what are the FEBugs in a domain they are interested in.

To help developers better find FEBugs, we propose a technique to identify FEBugs from the many questions in Stack Overflow, which is currently one of the most popular sites where software engineers search, communicate, collaborate, and share their experience and expertise with one another. In the past 8 years, Stack Overflow has built up over 15 million questions (up to January 2018) along with their corresponding answers that cover a wide range of topics and has become a large knowledge repository. Among Stack Overflow posts (i.e., questions and answers), many of them are related to bugs in different domains – an example is shown in Figure 1[1]. Many users post descriptions of bugs that they encountered in Stack Overflow, and some other users who know how these bugs can be fixed provide solutions. Other users who encounter these bugs in the future may search and view these posts, and vote whether they are helpful or not. Many of these bug-related posts are submitted to Stack Overflow daily, and thus one promising way to identify FEBugs is by analyzing the large amount of data available in this knowledge repository.

To recover FEBugs in Stack Overflow, we initially try to use its search engine. For example, we can search using the keyword "bug" "$\langle domain \rangle$", where $\langle domain \rangle$ corresponds to a domain of interest (e.g., Java). Unfortunately, we find that the current search engine of Stack Overflow does not allow one to find FEBugs well. Often, many of the top results are not what we want. The search results can be ranked by "relevance" (best match to search terms), "newest" (most recently posted), "votes" (highest number of votes), and "active" (most recently edited). When ranked by "relevance", "newest" or "active", many top results are related to bugs in the specified domain, but often with low vote scores or receiving no answer. These questions often describe highly specific bugs. On the other hand, when we chose to rank by "votes", although many questions are generic ones, they are often irrelevant to the specified domain of interest. For example, Figure 2 shows the top-5 results when we searched using the keywords "bug" and "Java" and ranked the search results by "votes". We find that 4 of the 5 results are not related to Java bugs.

To deal with the limitation of Stack Overflow's search engine, we propose a solution that can identify FEBugs more accurately.

[1] http://stackoverflow.com/questions/2194808, June 2016.

ICPC'18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

We propose an approach named RFEB (Recommending Frequently Encountered Bugs) which considers *both the relevance and popularity of questions in Stack Overflow*. RFEB includes an iterative query refinement process to improve the quality of the simple query (i.e., "bug" "$\langle domain \rangle$") by adding additional terms so that more *relevant* results can be retrieved. Moreover, RFEB embeds a number of question popularity metrics in Stack Overflow (e.g., the votes score and the number of views) in the retrieval process. The output of RFEB are the top N FEBugs of a specified domain.

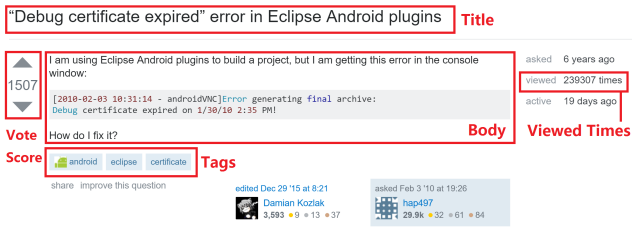


Figure 1: A Frequently Encountered Bug (FEBug) in Stack Overflow

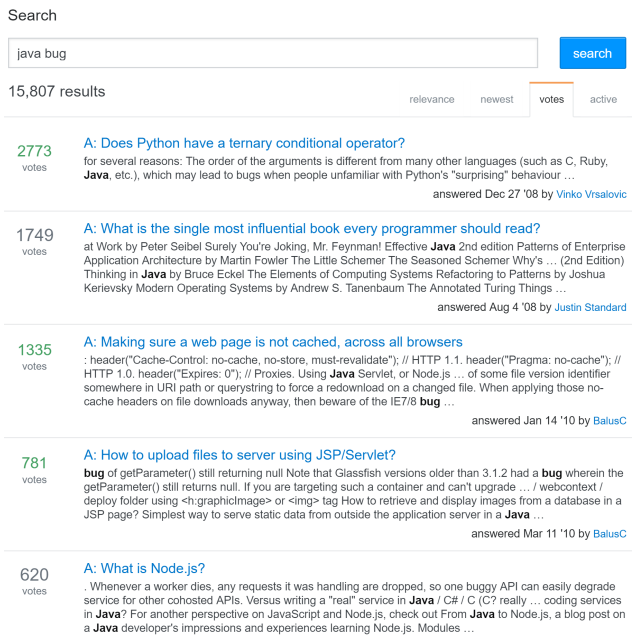


Figure 2: A Stack Overflow Search Result

We compare our model RFEB with Stack Overflow's search engine, which we use as a baseline. As the evaluation metric, we use the normalized discounted cumulative gain (NDCG), which evaluates the quality of a search engine based on the graded relevance of the entities in the result set. The value of NDCG varies from 0.0 to 1.0, while 1.0 means the ranking is ideal. NDCG is widely used in text retrieval to evaluate the performance of a web search engine [19, 33]. Since only the top few results matters, we focus on the top-10 results and compute $NDCG_{10}$.

We evaluate RFEB on a Stack Overflow data dump considering 10 domains, and compare it with the search engine of Stack Overflow. The experiment results show that RFEB is more effective than Stack Overflow's search engine on all 10 domains when evaluated in terms of $NDCG_{10}$. Among the 10 domains, the $NDCG_{10}$ scores of RFEB range from 0.908 to 0.992, and the average $NDCG_{10}$ score of RFEB is 0.96, which are very high. Furthermore, in terms of the average $NDCG_{10}$ score, RFEB outperforms Stack Overflow's search engine by 19.701%.

The main contributions of this paper are listed as follows:

(1) To our best knowledge, we are the first to recommend frequently encountered bugs from Stack Overflow, and we propose a novel model named RFEB which could output the top N FEBugs given a specific domain of interest.
(2) We evaluate RFEB on 10 domains, and demonstrate that RFEB performs better than Stack Overflow's search engine in terms of $NDCG_{10}$.

Paper structure. In Section 2, we present the overall framework and the details of RFEB. In Section 3, we present our experimental results. Section 4 discusses some issues about the performance, efficiency, and threats to validity of RFEB. In Section 5, we briefly review the related work. In Section 6, we conclude and mention future work.

2 PROPOSED APPROACH

In this section, we first provide a birds-eye-view of various steps of RFEB in Section 2.1. The steps are elaborated in the subsequent subsections.

2.1 Overview

The overall framework of RFEB is presented in Figure 3. Our framework takes as input the $\langle domain \rangle$ and a Stack Overflow dataset. The $\langle domain \rangle$ corresponds to a particular programming topic that interests a developer, e.g., Java, Android, etc. The Stack Overflow dataset contains information of historical questions and answers in Stack Overflow such as the id, title, body, tags, vote score, and view times. We make use of a data dump of Stack Overflow which has been made publicly available.[2] In our framework, RFEB first preprocesses the textual contents of questions in Stack Overflow, to convert the questions to the form of "bag of words" (Step 1). Then, when a user gives a domain of interest (Step 2), RFEB creates and analyzes the preprocessed dataset to find a number of questions that are likely to describe bugs or defects in the domain based on a simple keyword matching strategy. We refer to these questions as the *candidate domain-specific bug dataset* (CDB) (Step 3). RFEB ranks the questions in CDB in descending order of their vote scores multiplied by view times, and chooses the top K questions for a *query refinement* step (Step 4). These top K questions are popular ones that are likely to be generic questions that affect many developers. Next, RFEB performs an iterative query refinement process to improve the relevance of the top K questions to bugs in the given domain. It expands the query using additional terms (i.e., words) from the top K questions to find more relevant questions (Step 5), re-ranks the questions in the domain-specific bug dataset according to the new query (Step 6), and then judges if the ordering of the re-ranked questions is stable (Step 7). If it is stable, then RFEB

[2] https://archive.org/details/stackexchange

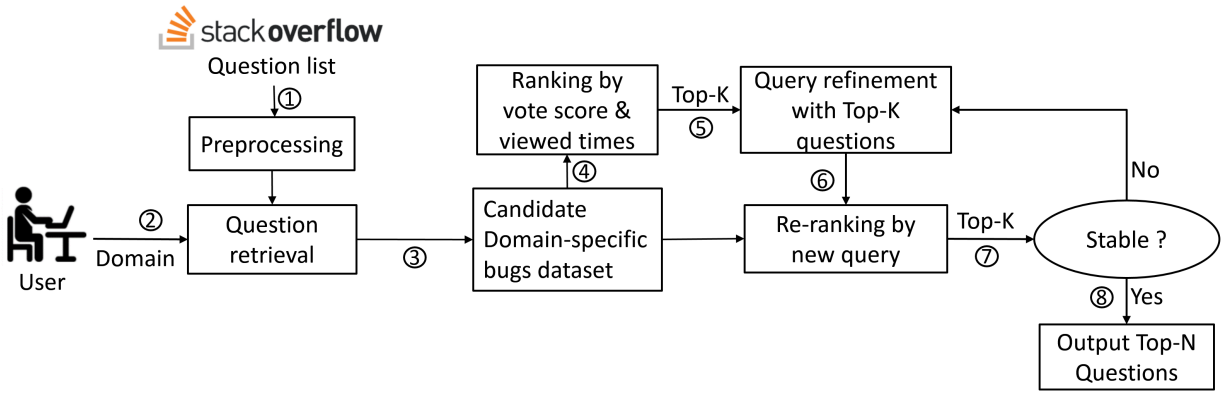


Figure 3: Overall Framework of RFEB

outputs the top N questions as FEBugs (Step 8), if not, it repeats Steps 5-7 until the ordering is stable. We elaborate the steps of the framework in the next subsections.

2.2 Step 1: Preprocessing

In the preprocessing step, we extract the title and body texts from the questions in the data set. Then, we tokenize the extracted texts, remove the stop words, and do stemming. Stop words refer to the widely used words, e.g., "a", "the", "and", "he". The stop words are of little help in distinguishing different questions as they are used too often. Stemming refers to the process of reducing the words to their root forms. For instance, after stemming, "marks", "marked", "marking" are all reduced to the same root word "mark". We apply the popular Porter stemming algorithm[3], which has been used in many other studies [26, 38]. We perform the stemming process to standardize words which keep the same meaning but are presented in different forms. Finally, we create a bag of words to represent each question, and forward the preprocessed questions to the next step.

2.3 Steps 2 and 3: Construction of Candidate Domain-Specific Bug Dataset (CDB)

In step 2, a developer needs to provide a keyword or a number of keywords that describe a domain of interest (e.g., Java, Android, etc.). Next, in step 3, we filter the preprocessed questions based on two criteria: (1) they need to contain *all* the keywords that the user has provided or being tagged with the keywords, (2) they need to contain the keyword "bug" or "defect". Questions that satisfy these criteria are selected to form the candidate domain-specific bug dataset (CDB).

2.4 Step 4: Question Ranking

After the CDB is built, we rank the questions in the CDB to identify the generic ones that interest many people. These questions are likely to be FEBugs. We integrate two popularity metrics in Stack Overflow, namely the vote score and the number of views, to rank the questions. For post p, we calculate its score (denoted as $Rank_p$) as follows:

$$Rank_p = VoteScore_p \times ViewedTimes_p \quad (1)$$

[3] http://tartarus.org/martin/PorterStemmer/

Next, we rank the questions in the domain-specific bug dataset in descending order of their $Rank$ scores, and forward the top K questions to the next step.

2.5 Step 5: Query Refinement

We use query refinement to get a good set of words that can be used as a good query to retrieve FEBugs from the candidate domain-specific bug dataset (CDB). FEBugs need to be questions that are *generic* and *relevant*. In Step 4, we can get the *generic* bugs by looking into the popularity metrics. In this step, we would like to improve the relevance of the top-K results. This step is done many times initially with the input query "bug" and "$\langle domain \rangle$". This query is refined at the end of the subsequent query refinement steps, by adding additional words to the query. The query refinement process has been proved to be an effective way to improve the quality of a search engine, and it has been used in many previous software engineering studies [8, 14, 21].

The query refinement process in RFEB can be broken down into three steps: candidate expansion terms generation and ranking, expansion terms selection, and the query reformation. Query refinement is based on pseudo-relevance feedback, which considers the top K questions ranked in the domain-specific bug dataset as likely relevant questions to the query. Then we rank the terms in the top K questions by their *Dice similarity* [14] with the terms in the initial query, and select the ones with high similarity scores for query refinement. The common assumption in text retrieval is that if two terms co-occur in the same questions, they are semantically related [8]. Given a term m from the *query* and a term n from the top K *questions*, the Dice similarity score of m and n is defined as:

$$DiceSim = \frac{2df_{m \wedge n}}{df_m + df_n} \quad (2)$$

Where $df_{m \wedge n}$ means the number of questions in the dataset which contain both m and n, and df_m, df_n are the number of questions containing m and n, respectively.

The score of $DiceSim$ ranges from 0 to 1, and the higher the score, the more similar the two terms are. For a term v from the top K questions that does not appear in the input query, we sum up its $DiceSim$ scores with all terms from the input query and get the average score. Then we rank the terms from the top K posts in descending order of their average dice similarity scores.

After ranking, we remove all terms in the top K questions whose average dice similarity score is smaller than the query refinement threshold $threshold_e$. Next, we choose the top-10 terms as the expansion terms. If there are less than 10 terms remaining, we use all of them as the expansion terms. These terms would be added to the input query.

2.6 Step 6: Re-ranking

After a new query is generated by Step 5, we re-rank the questions in the candidate domain-specific bug dataset. We integrate the similarities between the refined query and the questions and the popularity metrics of the questions (i.e., the vote score and the number of views).

We use cosine similarity [29] to measure the similarity between a query and a question. For a question p and a query q, their vector representations are $\boldsymbol{PostVec}_p$ and $\boldsymbol{QueryVec}_q$. The cosine similarity between p and q is calculated as follows:

$$CosSim(\boldsymbol{PostVec}_p, \boldsymbol{QueryVec}_q) = \frac{\boldsymbol{PostVec}_p \cdot \boldsymbol{QueryVec}_q}{\mid \boldsymbol{PostVec}_p \mid\mid \boldsymbol{QueryVec}_q \mid}$$

The numerator of Equation (3) – the dot product of the two vectors $\boldsymbol{PostVec}_p = \langle wt_{p,1}, wt_{p,2}, \ldots, wt_{p,v} \rangle$ and $\boldsymbol{QueryVec}_q = \langle wt_{q,1}, wt_{q,2}, \ldots, wt_{nq,v} \rangle$ is computed as follows:

$$\begin{aligned} \boldsymbol{PostVec}_p \cdot \boldsymbol{QueryVec}_q \quad = \quad & wt_{p,1} \times wt_{q,1} + wt_{p,2} \times wt_{q,2} \\ & + \ldots + wt_{p,v} \times wt_{q,v} \end{aligned}$$

The terms $\mid \boldsymbol{PostVec}_p \mid$ and $\mid \boldsymbol{QueryVec}_q \mid$ in the denominator of Equation (3) refer to the sizes of the two vectors. The size of a vector $\boldsymbol{PostVec}_p$ is computed as follows:

$$\mid \boldsymbol{PostVec}_p \mid = \sqrt{wt_{p,1}^2 + wt_{p,2}^2 + \ldots + wt_{p,v}^2}$$

The computed cosine similarities are combined with popularity metrics (i.e., the vote score and the number of views) to re-rank the questions. For question p, we calculate its score as follows:

$$ReRank_p = CosSim_p \times VoteScore_p \times ViewedTimes_p \quad (3)$$

Then we rank the questions in the candidate domain-specific bug dataset in descending order of their $ReRank$ scores, and forward the top K questions to the next step.

2.7 Steps 7 and 8: Stability Check

After the re-ranking step, we get a new set of top K questions based on the new query. Then we judge the stability of the top K list by comparing the new set of top K questions with the previous one. We compute the intersection and union of the two sets of top K questions, and calculate a stability score as follows:

$$Stability = \frac{Size_{intersection}}{Size_{union}} \quad (4)$$

Given a suitable stability threshold $threshold_s$, we decide if the ordering is stable by the following equation:

$$Stability = \begin{cases} Stable, & if\ Stability \geqslant threshold_s \\ Not\ Stable, & Otherwise \end{cases} \quad (5)$$

Table 1: Dataset.

Domain	Number of questions related to bugs
Android	50829
css	10100
html	10408
ios	12169
Java	52971
Javascript	45825
jquery	24737
mysql	7324
php	27613
Python	19620

In the above equation, an ordering is classified as Stable, if $Stability$ is larger than or equal to $threshold_s$; otherwise it is classified as Not Stable. When the top K list is stable, RFEB outputs it; otherwise, RFEB repeats the query refinement process again until it is stable.

3 EXPERIMENT AND ANALYSES

In this section, we evaluate the performance of RFEB and compare it with the baseline approach. We use an Intel(R) Core(TM) i7-4710HQ 2.50 GHz CPU, 16GB RAM server to run the experiments.

3.1 Experiment Setup

We evaluate RFEB on historical questions of Stack Overflow from a Stack Exchange Data Dump[4]. The dataset contains more than 11 million questions from 2008 to 2016 on Stack Overflow. From this large question repository, we extract questions related to bugs on 10 domains: Android, css, html, ios, Java, javascript, jquery, mysql, php, and Python. Notice that the domains we chose are hot topics in Stack Overflow and plenty of posts are related to them; developers frequently discuss solutions for bugs in these 10 domains. For example, there are more than 40,000 questions which are related to fixing bugs in Java code in our collected dataset[5].

For each domain, we build a domain-specific bugs dataset which contains all the questions related to bugs, we extract the information such as id, title, body, tags, vote score, and view times of each question from Stack Overflow. Notice we remove the questions which are marked as "Closed", and the questions which have no answers. Table 1 shows the numbers of questions related to bugs of the 10 domains after extraction process. When a question is created, we just need to analyze which domain it belongs to and if it is describing a bug, if in case, add it to corresponding domain dataset, and there is no need to repeat the extraction process again.

Our approach has several parameters: K (i.e., number of questions used in the query refinement step), $threshold_e$ (i.e., query refinement threshold), $threshold_s$ (i.e., stability check threshold), and N (i.e., number of questions to be output). We set K to 20 following Carpineto et al. [8], while $threshold_e$, $threshold_s$, and N are set to 0.3, 1, and 10 respectively.

[4]https://archive.org/download/stackexchange

[5]These questions include the word Java and bug in their title, description or tags

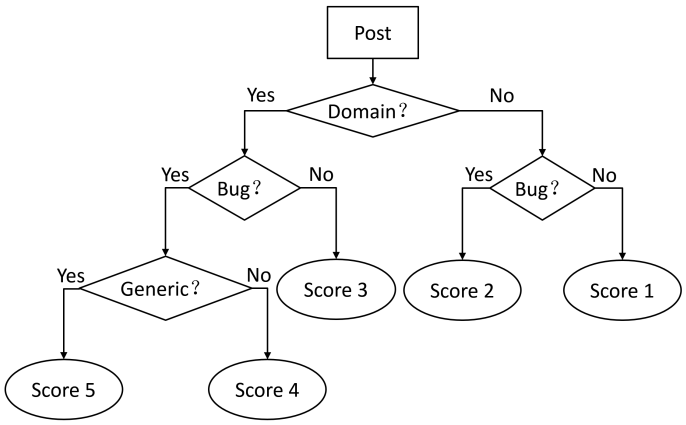


Figure 4: Approach of computing the final score of the labeled questions.

3.2 Baseline

The baseline approach in our experiment is the search engine of Stack Overflow. We use the query "bug" and "$\langle domain \rangle$" (e.g., "bug Java" for domain "Java") to search on the Stack Overflow website[6], and rank the results by "votes". In Stack Overflow, the search results can be ranked by "relevance", "newest", "votes", and "active", we choose to rank by "votes" as a question with high vote score means it was useful and clear to many users, and is more likely to be a frequently encountered bug.

3.3 User Study

A user study is performed to evaluate the top-10 questions produced by our approach RFEB and Stack Overflow's search engine on the 10 domains. We invited 16 programmers for the user study, who are PhD students in the lab and developers in industrial companies, and all of them have programming experience of more than 4 years. Each domain was manually labeled by two programmers. To ensure a high level of rigour in our manual annotation process, all participants are at least moderately familiar with the domains they labeled and have good English reading skills.

For each domain, the evaluation process contains two steps. First, we ask the two participants to independently label the 20 questions for this domain (10 from RFEB and 10 from Stack Overflow's search engine). Second, the two annotators discuss their disagreements to make a common decision; for questions that the two annotators cannot reach an agreement, the other participants were involved in the decision process.

The purpose of labeling the questions is to judge whether each of the questions is a frequently encountered bug in the domain. For each question, participants need to answer 3 questions:

(1) Is this question related to the domain?
(2) Does this question describe a bug?
(3) Is this question generic, rather than very specific?

The first question assesses whether a retrieved question is relevant to the domain of interest. The second question assesses whether a retrieved question is about a bug rather than other concerns. The third question assesses whether a retrieved question is likely to be frequently asked by many. Based on the three answers, we assign a score ranging from 1 to 5 to each retrieved question according to the criteria shown in Figure 4.

In the labeling process, we randomly mix the top-10 questions generated by our approach RFEB and the top-10 questions generated by Stack Overflow's search engine – participants do not know which result is produced by which approach. We used Fleiss Kappa [13] to evaluate the agreement between the two annotators. The overall Kappa value between the two participants considering all domains is 0.613, which indicates substantial agreement[7] between them.

3.4 Evaluation Metrics

The normalized discounted cumulative gain (NDCG) is used to evaluate the effectiveness of our model and the baseline approach. NDCG is a measure of search engine quality, which is widely used to evaluate the performance of search engine algorithms or other related applications. The value of NDCG varies from 0.0 to 1.0, while 1.0 means the ranking is ideal. NDCG is commonly used in text retrieval and to measure the performance of a web search engine [19, 33]. Two assumptions are made in using NDCG:

(1) The documents with high relevance are more useful than the documents with marginal relevant.
(2) The relevance document with low rank position is useless for the user as it is less likely to be examined.

The NDCG at a particular rank position p is defined as follows:

$$NDCG_p = \frac{DCG_p}{IDCG_p} \tag{6}$$

where

$$DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log_2(i+1)} \tag{7}$$

$IDCG_p$ is the maximum possible (ideal) DCG through position p for a given set of queries, documents, and relevances, and rel_i is the graded relevance of the result at position i. Note in a excellent ranking algorithm, the DCG_p will be the same as the $IDCG_p$ producing an NDCG of 1.0 [7].

3.5 Research Questions

This paper addresses the following research questions:

Research Question 1: How effective is RFEB**?**

Motivation. In this research question, we investigate whether RFEB can be used to identify FEBugs in Stack Overflow. Also, it is necessary to compare the performance of RFEB with the standard search engine of Stack Overflow. Answer to this research question could shed light to whether and to what extent RFEB improves over the baseline approach.

Approach. To address this research question, we build the experiments on 10 domains: Android, css, html, ios, Java, javascript, jquery, mysql, php, and Python. We compute NDCG at rank position 10 ($NDCG_{10}$) of RFEB when performing experiments on the Stack

[6] http://stackoverflow.com/

[7] Kappa values of < 0, [0.01, 0.20], [0.21, 0.40], [0.41, 0.60], [0.61, 0.80], [0.81, 1.00] are considered as poor agreement, slight agreement, fair agreement, moderate agreement, substantial agreement, almost perfect agreement, respectively

Table 2: $NDCG_{10}$ scores of RFEB compared with the standard search engine of Stack Overflow (SO).

Domain	RFEB	SO's search engine	Improvement
Android	0.991	0.912	8.662%
Css	0.992	0.771	28.664%
Html	0.965	0.701	37.66%
Ios	0.97	0.853	13.716%
Java	0.908	0.648	40.123%
Javascript	0.944	0.939	0.532%
Jquery	0.979	0.675	45.037%
mysql	0.951	0.886	7.336%
php	0.955	0.698	36.819%
Python	0.946	0.939	0.745%
Average	0.96	0.802	19.701%

Overflow dataset, and compare the results with the $NDCG_{10}$ scores of Stack Overflow's default search engine.

Results. Table 2 presents the experiment results of $NDCG_{10}$ scores of RFEB and the standard search engine of Stack Overflow, and shows the relative improvements that RFEB achieves over Stack Overflow's search engine. The relative improvement is calculated by dividing the difference of the $NDCG_{10}$ scores of RFEB and Stack Overflow's search engine with the $NDCG_{10}$ scores of Stack Overflow's search engine. We also report the average results of the 10 domains in the last row. The results demonstrate that RFEB is more effective than Stack Overflow's search engine on all 10 domains. The average $NDCG_{10}$ score of RFEB is 0.96, which outperform Stack Overflow's search engine by 19.701%. Among the 10 domains, $NDCG_{10}$ scores of RFEB range from 0.908 to 0.992, the $NDCG_{10}$ scores of Stack Overflow's search engine range from 0.648 to 0.939, and the improvement RFEB made of Stack Overflow's search engine range from 0.532% to 45.037%. Thus, we can draw the conclusion that RFEB performs better than Stack Overflow's search engine in terms of $NDCG_{10}$.

RFEB *outperforms Stack Overflow's default search engine on all 10 domains when evaluated in terms of* $NDCG_{10}$.

Research Question 2: In terms of NDCG, how effective are RFEB and Stack Overflow's search engine at different rank positions in search result lists?

Motivation. In RQ1, we compute NDCG at rank position 10 ($NDCG_{10}$) of RFEB and Stack Overflow's search engine. In this RQ, we investigate the performance of these two approaches at different rank positions. Answer to this research question can shed light as to whether RFEB outperforms Stack Overflow's search engine when other rank positions are considered.

Approach. To address this RQ, we calculate NDCG scores of RFEB and Stack Overflow's search engine when they are applied to the Stack Overflow question dataset on the 10 domains mentioned in RQ1. We investigate different rank positions in the search result list, from 1 to 10, at 1 interval. Next, we plot the NDCG graphs of the 10 domains that show the performance of RFEB and Stack Overflow's search engine when evaluated by $NDCG_1$ to $NDCG_{10}$.

Results. Figure 5 presents the NDCG graphs of RFEB and Stack Overflow's search engine for Android, css, html, ios, Java, javascript, jquery, mysql, php, and Python domains. From the graphs, for most of the domains, for most rank positions, we notice that RFEB performs better than Stack Overflow's search engine. Notice the graphs of Android, css, html, ios, Java, jquery, mysql, and php, RFEB obviously achieves better performance than Stack Overflow's search engine at all rank position from 1 to 10. For the domain javascript, RFEB performs better at the rank position from 4 to 10, and Stack Overflow's search engine performs better at the rank position from 1 to 3. For the domain Python, RFEB performs better at the rank position 4-7, 10, and Stack Overflow's search engine performs better at the rank position 2, 3, 8, 9.

When evaluated in terms of NDCG at rank position 1 to 10, for most cases, RFEB *outperforms Stack Overflow's search engine.*

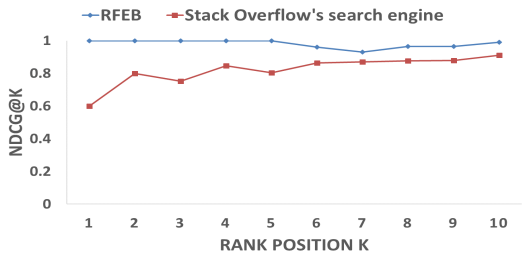
Research Question 3: What is the effect of varying the threshold in the query refinement step of RFEB?

Motivation. In the query refinement step, terms whose average dice similarity scores are higher than $threshold_e$ are selected to refine the query. In RQ1 and RQ2, we set the query refinement threshold as 0.3, which means the terms with dice similarity scores equal or higher than 0.3 will be added to the query (i.e., $threshold_e$ in Section 2.5). In this research question, we investigate the performance of RFEB with different query refinement thresholds. Answer to this research question can shed light to the effect of query refinement threshold in RFEB.

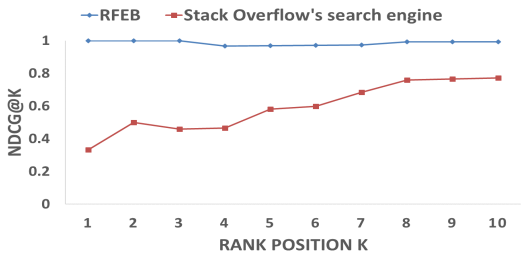
Approach. To address this question, we conduct an experiment with five different query refinement thresholds (i.e., 0.1, 0.2, 0.3, 0.4, and 0.5). We set the threshold up to 0.5 as we noticed that few terms' dice similarity scores are larger than 0.5. If the threshold is too high, there may be no terms could add to the query. As manually data labeling is a time-consuming work, we choose 2 domains in this experiment, the one achieves the highest $NDCG_{10}$ score in RQ1 – css and the one achieves the lowest $NDCG_{10}$ score in RQ1 – Java. We then compare the results achieved by RFEB using these different query refinement thresholds in terms of NDCG at rank position 10.

Results. Table 3 presents the experiment results. We report the $NDCG_{10}$ scores of RFEB with different query refinement thresholds. From the table, we can see that for css, RFEB achieves the same $NDCG_{10}$ score when threshold equals to 0.1, 0.2, and 0.3, and $NDCG_{10}$ is a bit low when threshold equals to 0.4 and 0.5, RFEB achieves the lowest $NDCG_{10}$ score when threshold equals to 0.5. For Java, RFEB achieves the same $NDCG_{10}$ score when threshold equals to 0.3 and 0.4, which is better than other three, RFEB achieves the lowest $NDCG_{10}$ score when threshold equals to 0.1. When considering the average scores of the 2 domains, RFEB performs best when threshold=0.3, and the performance of other threshold settings do not differ a lot, which are close to the result of threshold=0.3. RFEB performs relatively low $NDCG_{10}$ score with the threshold as 0.1 and 0.5 when compared to the other three thresholds (i.e., 0.2, 0.3, and 0.4).

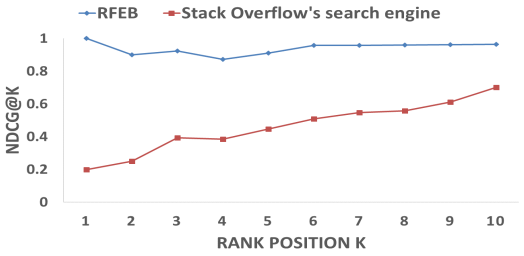
RFEB *with threshold=0.3 performs better than* RFEB *with other 4 threshold settings in terms of* $NDCG_{10}$. *The results of the 5 threshold do not vary much, which means query refinement threshold do not affect much to the performance of* RFEB.



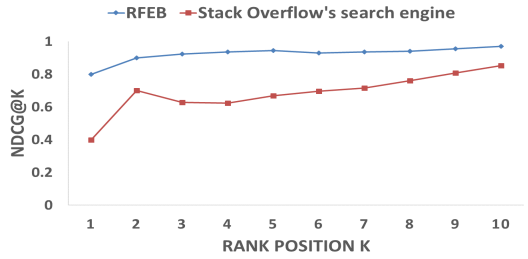
(a) Android



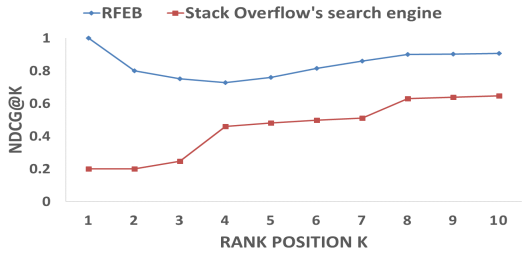
(b) Css



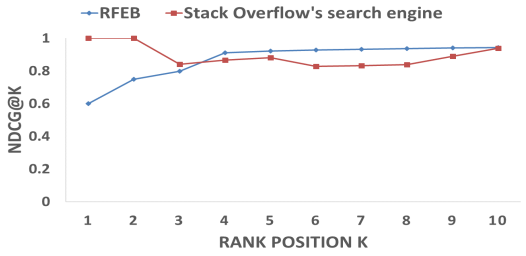
(c) Html



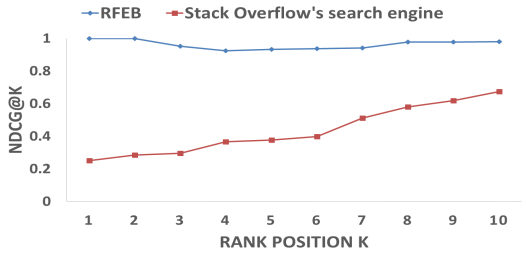
(d) Ios



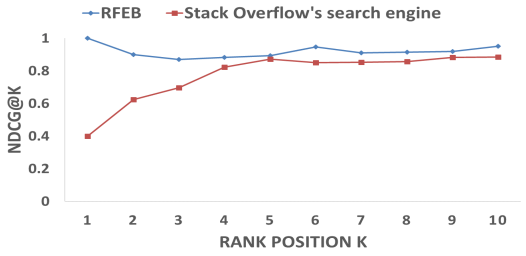
(e) Java



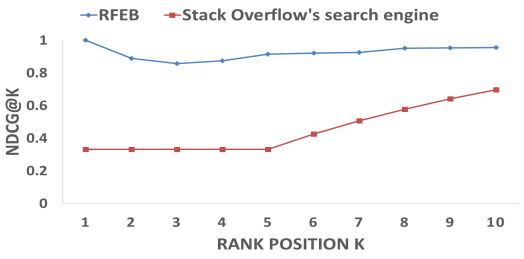
(f) Javascript



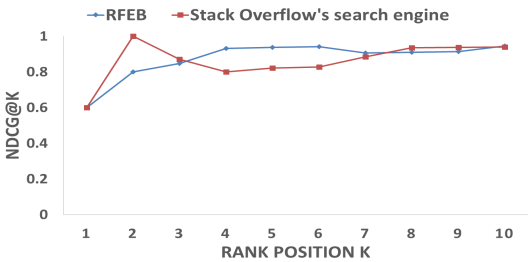
(g) Jquery



(h) Mysql



(i) Php



(j) Python

Figure 5: Graphs of NDCG at rank positions from 1 to 10.

Table 3: $NDCG_{10}$ scores of RFEB with different query refinement thresholds.

Domain	threshold=0.1	threshold=0.2	threshold=0.3	threshold=0.4	threshold=0.5
Css	0.992	0.992	0.992	0.961	0.946
Java	0.863	0.89	0.908	0.908	0.901
Average	0.928	0.941	0.95	0.935	0.924

Research Question 4: What is the effect of varying the threshold in the stability check step of RFEB?

Motivation. In the stability check step, a model is deemed stable if its stability is larger than or equal to $threshold_s$. In RQ1 and RQ2, we set the stability check threshold as 1 (i.e., $threshold_s$ in Section 2.7), which means the top K questions produced by RFEB in this iteration are the same as the top K question produced in last iteration. In this research question, we investigate the performance of RFEB with different stability check thresholds. Answer to this research question can shed light to the effect of stability check threshold in RFEB.

Approach. To address this RQ, we conduct an experiment with five different stability check thresholds (i.e., 0.2, 0.4, 0.6, 0.8, and 1). As manually data labeling is a time-consuming work, we choose 2 domains in this experiment, the one that achieves the highest $NDCG_{10}$ score in RQ1 – css and the one that achieves the lowest $NDCG_{10}$ score in RQ1 – Java. We then compare the results achieved by RFEB using these different stability check thresholds in terms of NDCG at rank position 10.

Results. The experiment results are shown in Table 4. We report the $NDCG_{10}$ scores of RFEB with different stability check thresholds. From the table, we can see that for css, RFEB achieves the best $NDCG_{10}$ score when threshold equals to 1, and $NDCG_{10}$ scores are the same when threshold equals to 0.8 and 0.6, RFEB achieves the lowest $NDCG_{10}$ score when threshold equals to 0.2. For Java, RFEB achieves the same $NDCG_{10}$ score when threshold equals to 1 and 0.8, which is better than other three, the results of RFEB are also the same when take threshold 0.6 and 0.4, RFEB achieves the lowest $NDCG_{10}$ score when threshold equals to 0.2. When considering the average scores of the 2 domains, RFEB performs best when threshold=1. We also notice that the higher the threshold is, the better performance RFEB could achieve, so if time permits, it's better to take a high stability check threshold.

RFEB *with threshold=1 performs better than* RFEB *with other 4 threshold settings in terms of* $NDCG_{10}$. RFEB *performs better with higher stability check threshold (no more than 1).*

4 DISCUSSION

4.1 Qualitative Analysis

Here, we perform a brief qualitative analysis on RFEB and Stack Overflow's search engine outputs. In Table 5 and Table 6, we present the top-5 questions returned by RFEB and Stack Overflow's search engine in the domain Python, respectively. We post the question ids, titles, short descriptions and the labelled results of the questions in the two tables. From the tables, we can see that all 5 questions returned by RFEB are related to Python, and the second, the third, and the forth questions are FEBugs; 2 of the 5 questions returned by Stack Overflow's search engine are not related to Python, and only the second one is a FEbug. From the example, we notice that the results returned by RFEB are of higher quality and are more likely to be FEBugs than those returned by Stack Overflow's search engine.

4.2 Time Complexity

We analyze the time complexity of RFEB in this section. Given a domain as an input, RFEB first finds a set of questions that are likely to describe bugs or defects in the domain. The time complexity of this step is $O(m)$, where m indicates the number of questions in Stack Overflow. Then RFEB re-ranks this set of questions several times with expanded queries until the output is stable. For this step, the time complexity is $O(n^2)$, where n indicates the number of questions RFEB selected in the previous step. Considering that RFEB re-ranks the questions k times, the total time complexity of RFEB is $O(m + kn^2)$. In our experiments, using an Intel(R) Core i7 2.5 GHz server with 16GB RAM running Windows 7 (64-bit), the time taken to return a query is between 1-3 minutes. We did not index the documents and the run time needed would have been much shorter if an index had been used. To improve the response time further, as a deployment strategy, query results can be cached, and only new documents need to be analyzed when the same query is asked again by another user of RFEB. We leave these and additional optimization strategies for future work.

4.3 Threats to Validity

Threats to internal validity refer to errors in our experiments. We have double checked our implementations and all the experiment results. Hence, we believe there are minimal threats to internal validity. Still, there could be errors that we did not notice.

Threats to external validity refer to the generalizability of our results. We tried to mitigate this threat by evaluating our approach on millions of questions on Stack Overflow. We performed a user study to evaluate whether the retrieved questions are actually describing FEBugs or not. To reduce this threat, we invited 16 programmers for the user study. All participants are moderately familiar with the domain they labeled and have basic English reading skills. Finally,

[7] http://stackoverflow.com/questions/5082452, June 2016.
[8] http://stackoverflow.com/questions/1132941, June 2016.
[9] http://stackoverflow.com/questions/2988017, June 2016.
[10] http://stackoverflow.com/questions/191010, June 2016.
[11] http://stackoverflow.com/questions/5178416, June 2016.
[12] http://stackoverflow.com/questions/394809, June 2016.
[13] http://stackoverflow.com/questions/1132941, June 2016.
[14] http://stackoverflow.com/questions/49547, June 2016.
[15] http://stackoverflow.com/questions/4113299, June 2016.
[16] http://stackoverflow.com/questions/231767, June 2016.

Table 4: $NDCG_{10}$ scores of RFEB with different stability check thresholds.

Domain	threshold=0.2	threshold=0.4	threshold=0.6	threshold=0.8	threshold=1
Css	0.924	0.948	0.981	0.981	0.992
Java	0.826	0.88	0.88	0.908	0.908
Average	0.875	0.914	0.931	0.945	0.95

Table 5: Top-5 questions returned by RFEB for the domain Python.

Question ID	Title	Short Description	Domain?	Bug?	Generic?
5082452[7]	Python string formatting: % vs. .format	Ask questions about str.format() method.	Yes	No	Yes
1132941[8]	"Least Astonishment" in Python: The Mutable Default Argument	Describe "a dramatic design flaw" of Python on binding the default argument at function definition.	Yes	Yes	Yes
2988017[9]	String comparison in Python: is vs. ==	Describe a problem in python that '==' and 'is' sometimes produce a different result when used to compare strings.	Yes	Yes	Yes
191010[10]	How to get a complete list of object's methods and attributes?	Describe a problem in Python that the function "dir()" does not return a complete list.	Yes	Yes	Yes
5178416[11]	pip install lxml an error	Describe a error when installing a package lxml using pip.	Yes	No	No

Table 6: Top-5 questions returned by Stack Overflow's search engine for the domain Python.

Question ID	Title	Short Description	Domain?	Bug?	Generic?
394809[12]	Does Python have a ternary conditional operator?	Ask if Python supports a ternary conditional operator.	Yes	No	No
1132941[13]	"Least Astonishment" in Python: The Mutable Default Argument	Describe "a dramatic design flaw" of Python on binding the default argument at function definition.	Yes	Yes	Yes
49547[14]	Making sure a web page is not cached, across all browsers	A question about web page caching.	No	No	No
4113299[15]	Ruby on Rails Server options	Question about setting up a development server for Ruby on Rails applications.	No	No	No
231767[16]	What does the yield keyword do in Python?	Question about how the Python's "yield" keyword should be used and what does it do.	Yes	No	No

our choice of baseline clearly impacts the results. As future work, we plan to study more baselines. We only investigate one query format "bug $\langle domain \rangle$". Other query format could have been used. Due to the cost involved in performing the user study[8], we can only afford to investigate one query format. We leave investigation into other query formats for future work.

Threats to construct validity mean the suitability of the proposed evaluation metrics. We use NDCG which is also used by many software engineering studies to evaluate the quality of web search engines [20, 40]. Thus, the threat to validity is mitigated.

[8]In our user study, participants need to spend around one hour to label the 20 questions for a domain.

5 RELATED WORK

Studies on Online Social Media. Durning the process of software development and maintenance, developers frequently use online social media such as Stack Overflow to improve their efficiency [42, 46–48]. We highlight some of this research below.

Storey et al. [37] and Begel et al. [4] present the viewpoint of study in social media for software engineering in two papers and research the effect of social media at team, project, and community levels in software engineering. Zhang et al. proposed an approach named DupPredictor to detect the duplicate questions in Stack Overflow which considers multiple factors from the questions [53]. Hong et al. made a research on developer social networks, and compare

it with other social networks [17]. Prasetyo et al. propose an automatic approach to judge whether a microblog is software-related or not [27].

Xia et al. proposed an approach which automatic recommend tags to users by analyze the information in Freecode and Stack Overflow [43]. In a later work, Wang et al. proposed a better recommendation approach by integrating Bayesian inference and frequentists [41]. Barua et al. proposed an approach to explore the central topic of questions in Stack Overflow by leveraging LDA topic modeling [3]. Correa et al. propose a model to predict whether a question posted in Stack Overflow will be deleted or not [10]. And they also built a model to predict closed questions in Stack Overflow by machine learning techniques [9]. Duijn et al. proposed a method to better predict the quality the questions in Stack Overflow by analyzing the code fragments posted [11]. Romano et al. proposed a weighted votes metric to distinguish high quality answers from low quality ones in Stack Overflow, which can give different weights to the votes of the answers in Stack Overflow and tend to emphasize the answer that receives most of the votes when most of the answers were already posted [30]. Xu et al. propose an approach, named AnswerBot, that can automatically generate an answer summary to a developer's technical question [45]. Ahasanuzzaman et al. performed a manual investigation to understand why users submit duplicate questions in Stack Overflow, and based on the manual investigation, they proposed a classification technique that uses a number of carefully chosen features to identify duplicate questions [1]. Beyer et al. presented an approach to group tag synonyms to meaningful topics [6]. The synonyms are represented as a directed and weighted graph and several community detection algorithms are used to identify meaningful groups of tags from the graph.

Our work is different from the above mentioned researches: we focus on recommending frequently encountered bugs in Stack Overflow. We propose a new approach named FEBugs to help developers improve work efficiency.

Studies on Query Refinement. In the text retrieval field, query refinement has been assured to be an effective method to improve the quality of results returned by a text retrieval engine [28, 32]. A considerable amount of strategies have been proposed, which classified in two main categories: query expansion [8] and query reduction [2, 49]. In software engineering field, some works also use query refinement strategies to improve effectiveness of software engineering tasks. We introduce some of these studies below.

Petrenko et al. investigated query refinement using ontology fragments in the context of concept location [25]. Starke et al. conducted a formative study in how users searched source code on an unfamiliar system using query refinement strategy [36]. Marcus et al. performed a query expansion process by adding the most similar words from the source code to the query [23]. Yang et al. proposed a general technique which uses the context of words in source code and comments to automatically inferred synonyms, antonyms, abbreviations and related words, then add them to the refined query [51]. Hill et al. perform a query refinement process by expand possible terms which searched from source code by contextual search method [16]. Shepherd et al. developed a semi-automated code search tool which using selective terms drawn from verb-direct object pairs to refine starting queries [34]. Sisman et al. proposed an automatic query Reformulation framework which expand users's queries by specific terms learned from top-ranked documents searched by the initial query [35]. In this paper, we also apply automatic query refinement in our model, which expand the query by terms similar or related to the initial query terms.

Studies on Frequently Asked Questions. Frequently Asked Questions (FAQs) refers to a documentation format which contains questions and answers, as they are supposed to be asked frequently in some context. There are various research on question answering systems, in order to find the most appropriate answer to the question. Henbeta et al. proposed an automatic approach to collect frequently asked questions from software development discussions using natural language processing and text mining [15]. Hu et al. perform a semi-automatic approach to find frequently asked questions, which could help forum managers to construct the FAQs [18]. Romero et al. built a system beyond classic FAQ retrieval algorithms which integrates a tag cloud generation module and a FAQ retrieval model to help users better understand the retrieved information [31]. Ng and al. applied data-mining techniques to FAQs, and they built a system which is able to predict user questions by information mining from FAQ lists [24]. Similar to FAQs, the bugs we recommended in this paper are also in the format which lists questions and answers, and these bugs are commonly encountered in some context.

However, these prior works have not identified frequently encountered bugs from the many questions in Stack Overflow. The concern addressed by many of these prior works (e.g., a question being asked many times) has been addressed in Stack Overflow by a community filtering process (i.e., the marking of duplicate questions).

6 CONCLUSION AND FUTURE WORK

In this paper, we propose RFEB to recommend frequently encountered bugs (FEBugs) in Stack Overflow. Given a target domain, RFEB considers both relevance and popularity of Stack Overflow questions, and employs an iterative refinement process to improve relevance. RFEB eventually outputs a ranked list of questions for the input domain of interest. We conduct experiments on 10 domains using the dataset of Stack Overflow, which includes more than ten million questions. Our experiment finds that: (1) RFEB outperforms Stack Overflow's search engine on all the 10 domains when evaluated in terms of $NDCG_{10}$; (2) When evaluated in terms of NDCG at rank position from 1 to 10, RFEB achieves better performance than Stack Overflow's default search engine for most cases; (3) The query refinement threshold do not affect the performance of RFEB much; (4) RFEB performs better with higher stability check threshold. Overall, our experiment shows that our proposed model RFEB is more effective compared with Stack Overflow's search engine when identifying FEBugs.

In the future, we plan to evaluate RFEB on more domains, using datasets from other software question and answer sites or forums or repository hosting sites (e.g., GitHub).

Acknowledgment. Xin Xia is the corresponding author. The authors would like to thank the user study participants for their labeling effort. This work was partially supported by NSFC Program (No. 61602403 and 61572426)

REFERENCES

[1] Muhammad Ahasanuzzaman, Muhammad Asaduzzaman, Chanchal K Roy, and Kevin A Schneider. 2016. Mining Duplicate Questions of Stack Overflow. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 402–412.

[2] Niranjan Balasubramanian, Giridhar Kumaran, and Vitor R Carvalho. 2010. Exploring reductions for long web queries. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*. ACM, 571–578.

[3] Anton Barua, Stephen W Thomas, and Ahmed E Hassan. 2014. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering* 19, 3 (2014), 619–654.

[4] Andrew Begel, Robert DeLine, and Thomas Zimmermann. 2010. Social media for software engineering. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 33–38.

[5] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiß, Rahul Premraj, and Thomas Zimmermann. 2007. Quality of bug reports in Eclipse. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*. ACM, 21–25.

[6] Stefanie Beyer and Martin Pinzger. 2016. Grouping android tag synonyms on stack overflow. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 430–440.

[7] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. 2005. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*. ACM, 89–96.

[8] Claudio Carpineto and Giovanni Romano. 2012. A survey of automatic query expansion in information retrieval. *ACM Computing Surveys (CSUR)* 44, 1 (2012), 1.

[9] Denzil Correa and Ashish Sureka. 2013. Fit or unfit: analysis and prediction of'closed questions' on stack overflow. In *Proceedings of the first ACM conference on Online social networks*. ACM, 201–212.

[10] Denzil Correa and Ashish Sureka. 2014. Chaff from the wheat: characterization and modeling of deleted questions on stack overflow. In *Proceedings of the 23rd international conference on World wide web*. ACM, 631–642.

[11] Maarten Duijn, Adam Kučera, and Alberto Bacchelli. 2015. Quality questions need quality code: classifying code fragments on stack overflow. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 410–413.

[12] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *ACM Sigsac Conference on Computer & Communications Security*. 73–84.

[13] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin* 76, 5 (1971), 378.

[14] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *ICSE*. IEEE Press, 842–851.

[15] Stefan Henβ, Martin Monperrus, and Mira Mezini. 2012. Semi-automatically extracting FAQs to improve accessibility of software development knowledge. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 793–803.

[16] Emily Hill, Lori Pollock, and K Vijay-Shanker. 2009. Automatically capturing source code context of NL-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 232–242.

[17] Qiaona Hong, Sunghun Kim, Shing Chi Cheung, and Christian Bird. 2011. Understanding a developer social network and its evolution. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 323–332.

[18] Wei-Chung Hu, Dung-Feng Yu, and Hewijin Christine Jiau. 2010. A faq finding process in open source project forums. In *Software Engineering Advances (ICSEA), 2010 Fifth International Conference on*. IEEE, 259–264.

[19] Kalervo Järvelin and Jaana Kekäläinen. 2002. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems (TOIS)* 20, 4 (2002), 422–446.

[20] Ray R Larson. 2010. Introduction to information retrieval. (2010).

[21] X Allan Lu and Robert B Keefer. 1995. Query expansion/reduction and its impact on retrieval effectiveness. *NIST SPECIAL PUBLICATION SP* (1995), 231–231.

[22] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. 2016. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In *The ACM*. 711–722.

[23] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I Maletic. 2004. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. IEEE, 214–223.

[24] Dick Ng'Ambi. 2002. Pre-empting user questions through anticipation: data mining faq lists. In *Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*. South African Institute for Computer Scientists and Information Technologists, 101–109.

[25] Maksym Petrenko, Václav Rajlich, and Radu Vanciu. 2008. Partial domain comprehension in software evolution and maintenance. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE, 13–22.

[26] Martin F Porter. 1980. An algorithm for suffix stripping. *Program* 14, 3 (1980), 130–137.

[27] Philips K Prasetyo, David Lo, Palakorn Achananuparp, Yuan Tian, and Ee-Peng Lim. 2012. Automatic classification of software related microblogs. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 596–599.

[28] Mohammad Masudur Rahman, Shamima Yeasmin, and Chanchal K Roy. 2014. Towards a context-aware IDE-based meta search engine for recommendation about programming errors and exceptions. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 194–203.

[29] BY Ricardo and RN Berthier. 2011. Modern Information Retrieval: the concepts and technology behind search second edition. *Addision Wesley* 84 (2011), 2.

[30] Daniele Romano and Martin Pinzger. 2013. *Towards a weighted voting system for Q&A sites*. Technical Report. Delft University of Technology, Software Engineering Research Group.

[31] M Romero, Alejandro Moreo, and Juan Luis Castro. 2013. A cloud of FAQ: A highly-precise FAQ retrieval system for the Web 2.0. *Knowledge-Based Systems* 49 (2013), 81–96.

[32] Gerard Salton. 1971. The SMART retrieval system—experiments in automatic document processing. (1971).

[33] Hinrich Schütze. 2008. Introduction to Information Retrieval. In *Proceedings of the international communication of association for computing machinery conference*.

[34] David Shepherd, Zachary P Fry, Emily Hill, Lori Pollock, and K Vijay-Shanker. 2007. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th international conference on Aspect-oriented software development*. ACM, 212–224.

[35] Bunyamin Sisman and Avinash C Kak. 2013. Assisting code search with automatic query reformulation for bug localization. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE, 309–318.

[36] Jamie Starke, Chris Luce, and Jonathan Sillito. 2009. Searching and skimming: An exploratory study. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 157–166.

[37] Margaret-Anne Storey, Christoph Treude, Arie van Deursen, and Li-Te Cheng. 2010. The impact of social media on software engineering practices and tools. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 359–364.

[38] Ferdian Thung, David Lo, and Lingxiao Jiang. 2012. Automatic defect categorization. In *2012 19th Working Conference on Reverse Engineering*. IEEE, 205–214.

[39] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2017. Bug characteristics in blockchain systems: a large-scale empirical study. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, 413–424.

[40] Shaowei Wang, David Lo, and Lingxiao Jiang. 2014. Active code search: incorporating user feedback to improve code search relevance. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 677–682.

[41] Shaowei Wang, David Lo, Bogdan Vasilescu, and Alexander Serebrenik. 2014. EnTagRec: An Enhanced Tag Recommendation System for Software Information Sites.. In *ICSME*. 291–300.

[42] Xin Xia, David Lo, Denzil Correa, Ashish Sureka, and Emad Shihab. 2016. It takes two to tango: Deleted stack overflow question prediction with text and meta features. In *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, Vol. 1. IEEE, 73–82.

[43] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. 2013. Tag recommendation in software information sites. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 287–296.

[44] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. 2014. Automatic defect categorization based on fault triggering conditions. In *Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on*. IEEE, 39–48.

[45] Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. 2017. AnswerBot - Automated Generation of Answer Summary to Developers'Technical Questions. In *32nd IEEE/ACM International Conference on Automated Software Engineering*. ACM.

[46] Bowen Xu, Zhenchang Xing, Xin Xia, David Lo, and Xuan-Bach D Le. 2017. Xsearch: a domain-specific cross-language relevant question retrieval tool. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 1009–1013.

[47] Bowen Xu, Zhenchang Xing, Xin Xia, David Lo, Qingye Wang, and Shanping Li. 2016. Domain-specific cross-language relevant question retrieval. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 413–424.

[48] Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. 2016. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 51–62.

[49] Xiaobing Xue, Samuel Huston, and W Bruce Croft. 2010. Improving verbose queries using subset distribution. In *Proceedings of the 19th ACM international*

conference on Information and knowledge management. ACM, 1059–1068.

[50] Meng Yan, Xiaohong Zhang, Dan Yang, Ling Xu, and Jeffrey D Kymer. 2016. A component recommender for bug reports using Discriminative Probability Latent Semantic Analysis. *Information and Software Technology* 73 (2016), 37–51.

[51] Jinqiu Yang and Lin Tan. 2012. Inferring semantically related words from software context. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE Press, 161–170.

[52] Xin-Li Yang, David Lo, Xin Xia, Qiao Huang, and Jian-Ling Sun. 2017. High-impact bug report identification with imbalanced learning strategies. *Journal of Computer Science and Technology* 32, 1 (2017), 181–198.

[53] Yun Zhang, David Lo, Xin Xia, and Jian-Ling Sun. 2015. Multi-factor duplicate question detection in stack overflow. *Journal of Computer Science and Technology* 30, 5 (2015), 981–997.