Combining Word Embedding with Information Retrieval to Recommend Similar Bug Reports

Xinli Yang*, David Lo†, Xin Xia*‡, Lingfeng Bao*, and Jianling Sun*
*College of Computer Science and Technology, Zhejiang University, Hangzhou, China
†School of Information Systems, Singapore Management University, Singapore
zdyxl@zju.edu.cn, davidlo@smu.edu.sg, {xxkidd, lingfengbao, sunjl}@zju.edu.cn

***Abstract*—Similar bugs are bugs that require handling of many common code files. Developers can often fix similar bugs with a shorter time and a higher quality since they can focus on fewer code files. Therefore, similar bug recommendation is a meaningful task which can improve development efficiency. Rocha et al. propose the first similar bug recommendation system named $NextBug$. Although $NextBug$ performs better than a start-of-the-art duplicated bug detection technique REP, its performance is not optimal and thus more work is needed to improve its effectiveness. Technically, it is also rather simple as it relies only upon a standard information retrieval technique, i.e., cosine similarity. In the paper, we propose a novel approach to recommend similar bugs. The approach combines a traditional information retrieval technique and a word embedding technique, and takes bug titles and descriptions as well as bug product and component information into consideration. To evaluate the approach, we use datasets from two popular open-source projects, i.e., Eclipse and Mozilla, each of which contains bug reports whose bug ids range from [1,400000]. The results show that our approach improves the performance of $NextBug$ statistically significantly and substantially for both projects.**

***Index Terms*—Similar Bugs, Word Embedding, Information Retrieval, Recommendation Systems**

I. Introduction

To improve the quality of software systems, developers often allow users and testers to report bugs that they encounter. Such bugs are often reported in bug tracking systems, e.g., Jira or Bugzilla. It is often the case that a large number of bugs are reported. For example, Anvik et al. highlight that for Eclipse up to 300 new bug reports are reported daily [1]. This is far too many for developers to handle. Indeed, in many bug tracking systems, often hundreds of bug reports remain open for a long period of time (even years). Thus, there is a need to help developers to resolve bug reports more efficiently. As bug reports are resolved, the reliability and security of a software system would be improved. Defects would be fixed and vulnerabilities would be closed before it gets exploited.

To help address the above-mentioned need, recently, Rocha et al. propose the recommendation of similar bugs [2]. Similar bugs are defined as those that require handling of a large proportion of common files (e.g., at least 50% of common files). Intuitively, developers are often more efficient when they need to solve similar bugs one after another, and the quality of the bug fixes can also potentially be improved. This is the case since developers can focus on fewer source code files which reduces program comprehension effort. Rocha et al. have supported this intuition by a field study with Mozilla developers and built a recommendation system named $NextBug$. They show that $NextBug$ achieves better performance than a start-of-the-art technique that recommends duplicated bug reports.

Unfortunately, $NextBug$ performance is still not optimal and thus more accurate solutions are needed. Also, technically $NextBug$ is rather simple since it relies only on a standard information retrieval technique namely *cosine similarity*. More advanced solutions can potentially be built to achieve higher accuracy which can push the technique closer to adoption. Thus, in the paper, we propose a novel approach to recommend similar bugs. The approach combines a standard information retrieval technique and a word embedding technique, and takes bug titles and descriptions as well as bug product and component information into consideration. With preprocessed bug report documents (i.e., bug titles and descriptions), we build TF-IDF (Term Frequency-Inverse Document Frequency) vectors and word embedding vectors and calculate two similarity scores based on them respectively. In addition, we calculate a third similarity score based on bug product and component information. Finally, we combine the three similarity scores into one final score and make similar bug recommendation with it.

To evaluate the effectiveness of our approach and compare it with $NextBug$, we perform experiments on two datasets from two large open source software projects, i.e., Eclipse and Mozilla, containing a total of 763,729 bug reports (389,975 bug reports in Eclipse and 373,754 bug reports in Mozilla). As accuracy yardsticks, we use three evaluation metrics, i.e., *recall-rate@k*, *mean average precision* (MAP) and *mean reciprocal rank* (MRR). These metrics are commonly-used in evaluating past recommendation systems to aid developers in performing software engineering tasks [3], [4], [5], [2], [6]. The experimental results show that our approach can achieve an improvement than $NextBug$ by a statistically significant and substantial margin for similar bug recommendation. Specifically, our approach improves the performance in terms of all the metrics by nearly 60% for the Eclipse dataset and by nearly 85% for the Mozilla dataset. In addition, our approach performs better than the three incomplete versions of our approach (its sub-approaches that only uses one of the three similarity scores respectively), which highlights the benefit of combining the three similarity scores.

‡Corresponding author.

The main contributions of this paper are:

1) We propose a novel approach for similar bug recommendation. The approach leverages bug titles, descriptions and product and component information of bug reports, and combines a standard information retrieval technique and a word embedding technique.
2) We compare our approach with a state-of-the-art baseline approach *NextBug* on two large open source projects, i.e., Eclipse and Mozilla. The experiment results show that our approach achieves a statistically significant and substantial improvement over *NextBug* for similar bug recommendation.

The rest of our paper is organized as follows. Section II introduces the background of our work. Section III presents the data collection and preprocessing steps. Section IV presents the overall framework of our approach. Section V describes our experiments and the results. Section VI discusses the related work. Conclusion and future work are presented in the last section.

II. Preliminaries

In this section, we first introduce the basic concepts of similar bugs in Section II-A. Next, we briefly introduce the main technique we leverage in our approach, namely word embedding technique, in Section II-B. Finally, we present the motivation of our proposed approach in Section II-C.

A. Similar Bug

Similar bug is a newly-proposed concept, which refers to the bugs that require handling of many common source code files (e.g., 50% of files modified to resolve the bugs are common). Similar bug is related to duplicate bug, which has been widely studied in the literature [7], [8], [9], [10], [11]. However, the condition for two bugs being similar is much more loose than that for duplicate bugs, which means that there are more similar bugs than duplicate bugs, given a query bug. Table I and II show a pair of similar bugs in Eclipse's bug tracking system. From the figures, we can see that although both of them are bugs about breakpoints, these two bugs are by no means duplicate bugs. However, they are in the same product, the same component, and they require handling the same file *"core/plugins/org.eclipse.dltk.debug/src/org/eclipse/dltk/internal/debug/core/model/ScriptBreakpointManager.java"*.

TABLE I
Bug 355616 in Eclipse's Bug Tracking System

Item	Content
Bug ID	355616
Component	Common-Debug
Product	DLTK
Title	Path mapping not done for "Run to line" breakpoints
Description	When using "Run to line", the created breakpoint contains the path for original file without any mapping. The proposed patch just adds this mapping before creating the breakpoint.

TABLE II
Bug 399991 in Eclipse's Bug Tracking System

Item	Content
Bug ID	399991
Component	Common
Product	DLTK
Title	Blocking breakpoint command
Description	When a breakpoint is added a "set breakpoint" command is sent to each thread (session) ... This problem is also available for the other commands manage by ScriptBreakpointManager

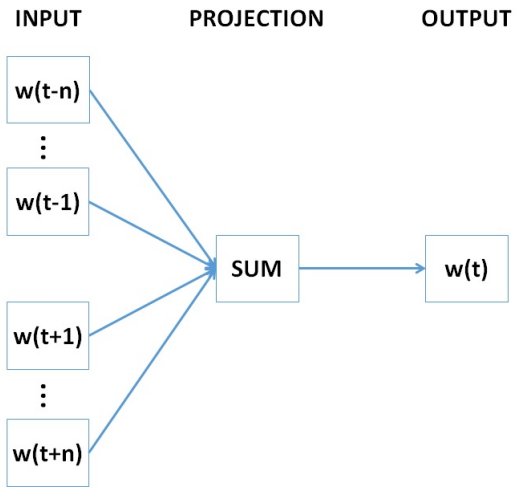


Fig. 1. The Architectures of CBOW Models

We can see that based on similar bug recommendation, developers can focus on fewer files when handling the same number of bugs. A developer assigned to similar bugs are likely to be able to resolve them faster than if he/she is assigned to irrelevant bugs since they can spend less time on program comprehension, which has been shown to take a substantial proportion of development and debugging time [12].

B. Word Embedding

In 1954, Harris proposed the distributional hypothesis, which states that words appearing in similar context tend to have similar meaning [13]. From then on, many distributional semantic models (DSMs) are proposed. In DSMs, each word is represented as a d-dimensional vector of real numbers such that words appearing in similar context have similar vector representations. Most of traditional DSMs are count-based models. Recently, some novel models based on neural network are proposed [14], [15], [16], [17]. These models use deep neural networks to learn from the context of the corpus to generate low-dimensional word vector representations, which is called "word embedding". Word embedding models have been proven to perform much better than the traditional count-based models for various information retrieval tasks [6], [18].

There are two popular word embedding models, i.e., CBOW model and skip-gram model [16], [17]. Figure 1 and 2 shows the architectures of the two models. In the figure, $w(t)$ represents the current word and $w(t \pm i)$ (i=1,2,...,n) represents

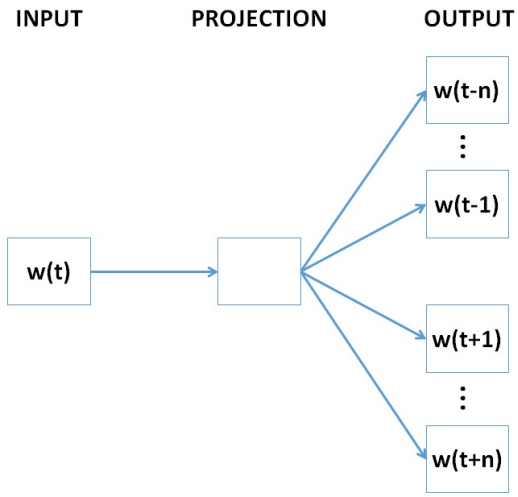


Fig. 2. The Architectures of Skip-gram Models

the surrounding words of $w(t)$ (i.e., the context). Specifically, CBOW model predicts the current word based on its context, while skip-gram model predicts the surrounding words given the current word. In our work, we use skip-gram model, since it has been shown to work well by past studies to solve other software engineering tasks [6], [18].

C. Motivation of Using Word Embedding

In software engineering, there are many information retrieval tasks such as duplicate bug identification in open source projects and tag recommendation in Q&A websites [9], [10], [19]. Most of them leverage traditional information retrieval techniques such as cosine similarity and TF-IDF method and extend the techniques to achieve good performance on these tasks.

Recently, word embedding techniques have been applied to several information retrieval tasks in software engineering and shown to achieve good performance [6], [18]. Word embedding techniques focus more on the relationship of words considering the context they appear, while traditional information retrieval techniques such as TF-IDF method focus more on the relationship of different documents in the whole corpus. We think the two classes of techniques are complementary. Therefore, in our approach we compute and combine two similarity scores based on word embedding vectors and TF-IDF vectors.

III. Data Collection and Preprocessing

In this section, we describe the details our data collection and data preprocessing step. We first introduce how we extract useful information from bug reports in Section III-A. Then we describe the process of building ground truth in detail in Section III-B.

A. Extracting Information From Bug Reports

We first collect bug reports whose bug ids range from [1,400000] for each of two large open-source projects, i.e., Eclipse and Mozilla, from the projects' Bugzillas. The bug reports of Eclipse are from October 2001 to February 2013, and those of Mozilla are from September 2001 to October 2007. Note that we use bug reports that are submitted at least 3 years ago since we want to consider only a set of fixed bug reports that are not likely to be reopened. Newer bug reports may not be stable; they can be reopened and the corresponding new fixes can touch more files. The latest bug reports considered by Rocha et al.'s study [2] are from 2012. For each bug report, we extract its title and description as well as the product and component information. These information will be used by our approach for recommending similar bugs.

For each bug report, we combine its title and description into a single document. And then we preprocess the document with the following steps. First, we extract all the terms (i.e., words) from the document. And then we remove stop words, numbers, punctuation marks and other non-alphabetic characters since they contain little information. Finally, we use the Snowball stemmer [20] to transform the remaining terms to their root forms (e.g., 'reading' and 'reads' are reduced to 'read') in order to reduce the feature dimensions and unify similar words into a common representation.

For the product and component information, we directly extract them from two fields of bug reports, i.e., *product* field and *component* field. We integrate the two kinds of information as a set. Specifically, each bug corresponds to a set $set = \{p, c\}$, in which p denotes the product information of the bug and c denotes the component information of the bug.

B. Building Ground Truth

To build the ground truth, we also collect in total 3,838,708 commit logs of Eclipse and Mozilla from Github. Generally, a commit log contains a message describing which bug the commit is aimed to fix and a list of committed files. Therefore, we can have bugs (bug ids) and their corresponding commit file lists based on these commit logs. Specifically, we use the approach proposed by Sliwerski et al. [21] to extract bug ids from commit messages. Since one bug may correspond to several commit logs, we union the committed file lists which correspond to the same bug id. Finally, we link bug reports and committed file lists according to the bug ids in order to ensure each bug has both bug report and committed file list.

Using the linked bug reports and committed file lists, we identify similar bugs. As mentioned before, similar bugs are bugs that require handling of many common code files. We use the method proposed by Rocha et al. to label similar bugs. Specifically, given two bugs x and y, we first calculate the ratio of mutually committed files, denoted as $Mutual(x, y)$, as follows:

$$Mutual(x, y) = \frac{|F_x \bigcap F_y|}{min(|F_x|, |F_y|)}$$

In the above formula, F_x and F_y represents the committed file lists of x and y. Next, we set a threshold for $Mutual(x, y)$ to identify whether x and y are similar. By default, the threshold is set as 0.5. That is, if at least half of the committed files of x and y are mutually committed, they are labeled as

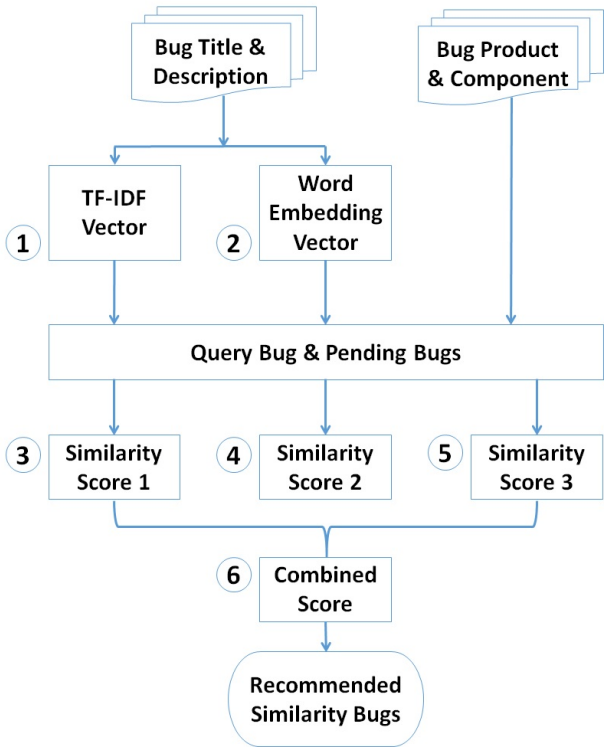


Fig. 3. The Overall Framework of Our Approach.

similar; otherwise, they are labeled as not similar. We follow the process to compare each pair of bugs and label all the similar bugs as the ground truth. In total, we find 13,337,653 pairs of similar bugs in Eclipse and 10,596,675 pairs of similar bugs in Mozilla.

IV. Our Proposed Approach

In this section, we elaborate our proposed approach. In particular, we first present an overall framework of our approach. Then, we describe in detail the implementation of each important step of our framework.

A. Overall Framework

Figure 3 presents the overall framework of our proposed approach. The framework mainly contains four components, i.e., TF-IDF component, word embedding component, product & component information component and combination component. In the first three components, we calculate three similarity scores based on the documents (i.e., bug titles and descriptions) and product and component information in the bug reports for the query bug and each of the pending bugs. In the last component, we combine the three similarity scores into a final score for each pending bug and recommend the bugs with the highest final scores as the similar bugs with respect to the query bug.

Specifically, given a query bug, in the first component we transform the documents (i.e., bug titles and descriptions) into TF-IDF vectors (cf. Section IV-B), and calculate the cosine similarity of the query bug with each of pending bugs based on their TF-IDF vectors to generate the first similarity score $Score_1$ (Step 1-2). In the second component, we transform the documents (i.e., bug titles and descriptions) into word embedding vectors (cf. Section IV-C), and calculate the cosine similarity of the query bug with each of pending bugs based on their word embedding vectors to generate the second similarity score $Score_2$ (Step 3-4). In the third component, we generate the third score $Score_3$ for the query bug and each of pending bugs based on the product and component information (cf. Section IV-D) in their bug reports (Step 5).

After having the three similarity scores, we combine them into a final score for each pending bug (cf. Section IV-E). The higher the score, the more similar the corresponding pending bug is with the query bug. Therefore, we sort the pending bugs based on the final scores, and recommend the top bugs as similar bugs (Step 6).

B. TF-IDF Vectors

TF-IDF (Term Frequency-Inverse Document Frequency) is one of the most popular information retrieval techniques. The main idea of TF-IDF is that if a term appears many times in one document and a few times in the other documents, the term has a good capability to differentiate the documents, and thus the term has high TF-IDF value.

Specifically, given a term t and a document d, we can define TF and IDF as follows:

$$TF(t,d) = \frac{\textit{Number of times } t \textit{ appears in } d}{\textit{Number of terms in } d}$$

$$IDF(t) = \log \frac{\textit{Total number of documents}}{\textit{Number of documents that contain } t + 1}$$

Finally, TF-IDF is computed as:

$$\text{TF-IDF}(t,d) = TF(t,d) \times IDF(t)$$

With the above formula, a document d (i.e., a bug report's title and description) can be represented as a TF-IDF vector, i.e., $d = (w_1, w_2 ... w_n)$, where w_i denotes the TF-IDF value of the i^{th} term in the document d.

With the TF-IDF vectors, we can compute similarity of two documents. We use cosine similarity since it is a popular method and has been shown to work well for TF-IDF vectors. Given two TF-IDF vectors v_1 and v_2, the similarity score $Score_1$ is calculated as follows:

$$Score_1 = \frac{v_1 \cdot v_2}{|v_1| * |v_2|}$$

A bigger value of $Score_1$ indicates that the corresponding two documents are more similar.

C. Word Embedding Vectors

In our paper, we leverage word embedding technique using skip-gram model. Specifically, given a word w, we denote the set of the surrounding context words of w as C_w. The objective function J of a skip-gram model (which needs to be

maximized) is the sum of log probabilities of the surrounding context words under the condition of a center word:

$$J = \sum_{i=1}^{n} \sum_{w_j \in C_{w_i}} \log p(w_j|w_i)$$

In the above formula, n denotes the whole length of the word sequence. In addition, $p(w_j|w_i)$ is the conditional probability defined using the following $softmax$ function:

$$p(w_j \in C_{w_i}|w_i) = \frac{exp(v_{w_j}^T v_{w_i})}{\sum_{w \in W} exp(v_w^T v_{w_i})}$$

In the above formula, v_w is the vector representation of the word w, and W is the vocabulary of all words. By training a whole corpus, all the words in the vocabulary of the corpus can be represented as a d-dimensional vectors where d is a variable parameter and generally set as an integer such as 100.

With skip-gram model, each word is transformed into a fixed-length vector. In theory, a document can be represented as a matrix in which each row represents a word. However, since different documents have different numbers of words, it is difficult to measure document similarity in this way. Therefore, we transform the document matrix into a vector by averaging all the word vectors the document contains. Specifically, given a document matrix that has n rows in total, we denote the i-th row of the matrix as r_i and the transformed document vector v_d is generated as follows:

$$v_d = \frac{\sum_i r_i}{n}$$

With the above formula, each document can be represented as a word embedding vector. We also use cosine similarity to measure the similarity of each pair of word embedding vectors to generate the similarity score $Score_2$. Also, the bigger $Score_2$ is, the more similar two documents are.

D. Leveraging Product & Component Information

Based on the definition of similar bugs (i.e., bugs that require handling of many common code files), almost all similar bugs are in the same product and component. Therefore, we also generate a similarity score $Score_3$ based on the product and component information in bug reports to better recommend similar bugs. As mentioned in Section III, each bug can be mapped to a set containing its product and component information. Given two bugs, we denote their corresponding sets as set_1 and set_2. The similarity score $Score_3$ is calculated as follows:

$$Score_3 = \frac{|set_1 \bigcap set_2|}{|set_1 \bigcup set_2|}$$

According to the above formula, if two bugs are in the same product and in the same component, $Score_3$ equals to 1. If two bugs are neither in the same product nor in the same component, $Score_3$ equals to 0. In other cases (two bugs are in the same product or in the same component), $Score_3$ equals to 0.5.

E. Similar Bug Recommendation

After $Score_1$, $Score_2$ and $Score_3$ have been computed, we combine them to produce a final score to better recommend similar bugs. As mentioned before, although both $Score_1$ and $Score_2$ represent document (bug title and description) similarity, they are complementary. The score $Score_1$ is generated based on TF-IDF vectors, which focus more on relationship of different documents in the whole corpus. The score $Score_2$ is generated based on word embedding vectors, which focus more on the relationship of words considering the context they appear. Since it is unclear which of the two scores are more important, we simply add them up so that they have equal weight. The score $Score_3$ can be seen as a filter. If two bugs are in the same product and in the same component, whether they are similar bugs or not depends mainly on their document similarity, while if two bugs are neither in the same product nor in the same component, they are unlikely to be similar bugs even if their document similarity is high. Therefore, we multiply the sum of the first two scores with $Score_3$ as the final score $Score$:

$$Score = (Score_1 + Score_2) \times Score_3$$

Based on the above formula, $Score$ is a non-negative number. A bigger $Score$ indicates that the corresponding pair of bugs are more similar.

V. Experiments and Results

In this section, we evaluate the effectiveness of our approach. The experimental environment is an Intel(R) Xeon(TM) E5-2650 2.00 GHz CPU, 80GB RAM desktop running Windows Server 2008 (64-bit). We first present our evaluation metrics and experiment setup in Sections V-A and V-B. We then present three research questions and our experiment results that answer these questions in Section V-C.

A. Evaluation Metrics

We use three evaluation metrics, i.e., *recall-rate@k*, *mean average precision* (MAP) and *mean reciprocal rank* (MRR), to evaluate the effectiveness of our approach. These metrics are commonly-used to evaluate recommendation systems to solve software engineering tasks [3], [4], [5], [2], [6]. To briefly introduce these metrics, we first give some notations. Given a query bug q, let us denote its ground truth similar bug set as $S(q)$, and the top-k recommendation set produced by a recommendation system as $R(q)$. Based on these notations we define the three metrics as follows:

1) **Recall-rate@k.** *Recall-rate@k* checks whether a top-k recommendation is useful. The definition of *recall-rate@k* for a query bug q is as follows:

$$\textit{Recall-rate@k}(q) = \begin{cases} 1 & \text{if S(q)} \bigcap \text{R(q)} \neq \emptyset \\ 0 & \text{Otherwise} \end{cases}$$

According to the formula, if there is at least one ground truth similar bug in the top-k recommendation, the top-k recommendation is useful for the query bug q. Given a

set of query bugs, we compute the proportion of useful top-k recommendations by averaging the *recall-rates@k* of all query bugs to get an overall *recall-rate@k*.

2) **MAP.** MAP (Mean Average Precision) is defined as the mean of the Average Precision (AvgP) values obtained for all the evaluation queries. The AvgP of a single query q is calculated as follows:

$$AvgP(q) = \sum_{n=1}^{|S(q)|} \frac{Prec@k(q)}{|S(q)|}$$

In the above formula, $Prec@k$ is the retrieval precision over the top-k bugs in the ranked list, i.e., the ratio of ground truth similar bugs of the query bug q in the top-k recommendation:

$$Prec@k(q) = \frac{\textit{\# of ground truth similar bugs in top-k}}{n}$$

3) **MRR.** MRR (Mean Reciprocal Rank) is defined as the mean of the Reciprocal Rank (RR) values obtained for all the evaluation queries, where RR of a single query q is the multiplicative inverse of the rank of the first correct answer $first_q$ (i.e., first ground truth similar bug in the recommendation list):

$$RR(q) = \frac{1}{first_q}$$

B. Experimental Settings

In our experiment, we simulate real bug resolution process to recommend similar bugs. When a bug triager (i.e., a person who evaluates incoming bug reports to decide the subsequent course of action) processes a new bug report, he/she may search for similar bugs reported before to find similar bugs, and assign the new bug to a developer who have resolved a similar bug recently. Therefore, all bugs in our datasets are regarded as query bugs once. And given a query bug q, we only recommend its potential similar bugs whose bug ids are less than that of q (i.e., bugs reported before q was reported).

Since our approach recommends similar bugs based on the combined similarity scores, we can rank all the pending bugs given a query bug. Therefore, we can achieve top-k recommendation with any integer less than the total number of pending bugs for k. Specifically, for each query bug, we recommend k most similar bugs which have the highest similarity scores in a top-k recommendation.

In our approach, word embedding technique is applied by using the python package named $gensim$[1]. The implementation contains many parameters such as context window size s, initial learning rate $alpha$ and the dimension of the word vector d. We use the default values for all the parameters. Specifically, s is set as 5, $alpha$ is set as 0.025 and d is set as 100.

[1]http://radimrehurek.com/gensim/models/word2vec.html

C. Research Questions

Our experiments are designed to answer the following research questions:

RQ1 How effective is our approach to recommend similar bugs?

Motivation. In the first research question, we want to investigate the effectiveness of our approach in similar bug recommendation. We also need compare our approach with the state-of-the-art approach to investigate whether and to what extent it improves over the prior work.

Methodology. We compare our approach against $NextBug$ [2]. $NextBug$ relies on a standard information retrieval technique for preprocessing documents (i.e., titles of bug reports) and calculating the cosine similarity of the documents for the query bug and all pending bugs. After having all the similarity scores, $NextBug$ sets a threshold (ranging from 0.0 to 0.8) to recommend similar bugs. That is, $NextBug$ only recommends similar bugs whose similarity scores are above the threshold. In our experiment, we set the threshold as 0.3 for $NextBug$ since the value achieves a good result [2]. Although $NextBug$ is simple, it has been proven to perform better than an alternative approach (REP [10]) for similar bug recommendation by Rocha et al. [2].

We use the three metrics presented earlier to make comparison. For *recall-rate@k*, we consider *recall-rate@1*, *recall-rate@5* and *recall-rate@10*. We use *recall-rate@1* to investigate the effectiveness of our approach and $NextBug$ under the strictest requirement. However, the *recall-rate@1* result may not be good; thus, we also investigate other top-k results (where k is a relatively small number, e.g., 10). Past recommendation studies in software engineering also consider larger values of k [5], [22]. For example, for bug localization, Kochhar et al. show that many developers are willing to check top-5 and even top-10 results [22]. Therefore, we consider *recall-rate@5* and *recall-rate@10* as well.

TABLE III
PERFORMANCE OF OUR APPROACH COMPARED WITH NEXTBUG FOR THE ECLIPSE DATASET

Project	NextBug	Our Approach	Improvement
Recall-Rate@1	0.1140	0.1691	48.33%
Recall-Rate@5	0.2499	0.3791	58.90%
Recall-Rate@10	0.3176	0.4867	53.24%
MAP	0.1696	0.2669	57.37%
MRR	0.2858	0.4489	57.07%

TABLE IV
PERFORMANCE OF OUR APPROACH COMPARED WITH NEXTBUG FOR THE MOZILLA DATASET

Project	NextBug	Our Approach	Improvement
Recall-Rate@1	0.1093	0.1993	82.34%
Recall-Rate@5	0.2349	0.4342	84.84%
Recall-Rate@10	0.2908	0.5373	84.77%
MAP	0.1821	0.3330	82.87%
MRR	0.3202	0.5905	84.42%

Results. Tables III and IV present the results of our approach as compared with those of the baseline *NextBug*. From the tables, we can conclude several points.

First, *NextBug* does not perform as well as what we expect in our experiment, which results from two main reasons. First, the similarity calculation method is not good enough. Second, since *NextBug* sets a threshold, it at times recommends few bugs. For example, given a query bug q and a threshold 0.3, if there are only one pending bug whose similarity score is above 0.3, then *NextBug* only recommends one similar bugs for q.

Second, our approach beats *NextBug* in terms of all the metrics for both Eclipse and Mozilla datasets. Specifically, our approach achieves a *recall-rate@10* of 0.49 and a MRR of 0.45 for the Eclipse dataset, and a *recall-rate@10* of 0.54 and a MRR of 0.59 for the Mozilla dataset. In addition, our approach achieves an improvement of nearly 60% for the Eclipse dataset and an improvement of nearly 85% for the Mozilla dataset over *NextBug* in terms of all the metrics.

TABLE V
MAPPINGS OF CLIFF'S DELTA VALUES TO THEIR INTERPRETATIONS [23]

Cliff's Delta (δ)	Interpretation
-1 <= δ < 0.147	Negligible
0.146 <= δ < 0.33	Small
0.33 <= δ < 0.474	Medium
0.474 <= δ <= 1	Large

To better demonstrate the superiority of our approach, we perform the Wilcoxon signed-rank statistical test to compute the p-value, and also compute the Cliff's delta. Wilcoxon signed-rank statistical test is often used to check if the difference in two data groups is statistically significant (which corresponds to a p-value of less than 0.05) or not. We include the Bonferroni correction to counteract the impact of multiple hypothesis tests. Cliff's delta is often used to check if the difference in two data groups are substantial. The range of Cliff's delta is in [-1, 1], where -1 or 1 means all values in one group are smaller or larger than those of the other group, and 0 means the data in the two groups is similar. The mappings between Cliff's delta scores and effectiveness levels are shown in Table V. We use all results from all query bugs to compute p-value and Cliffs delta. For each query bug, we have one value for NextBug and another for our approach. By computing the p-value and Cliff's delta, the extent of which our approach improves over *NextBug* can be more rigorously assessed.

Tables VI and VII present the p-values and Cliff's deltas of our approach compared with *NextBug* in terms of five metrics for the Eclipse and Mozilla dataset, respectively. From the tables, we can see the effectiveness of our approach more clearly. Compared with *NextBug*, our approach statistically significantly (i.e., p-value < 0.05) and substantially (i.e., Cliff's delta is not negligible) achieves a better performance in terms of all the metrics for both Eclipse and Mozilla datasets.

TABLE VI
P-VALUES AND CLIFF'S DELTA COMPARING THE FIVE METRICS OF OUR APPROACH WITH NEXTBUG'S FOR THE ECLIPSE DATASET

Metrics	P-Value	Cliff's Delta
Recall-Rate@1	<2.2e-16	1 (large)
Recall-Rate@5	<2.2e-16	1 (large)
Recall-Rate@10	<2.2e-16	1 (large)
MAP	<2.2e-16	1 (large)
MRR	<2.2e-16	1 (large)

TABLE VII
P-VALUES AND CLIFF'S DELTA COMPARING THE FIVE METRICS OF OUR APPROACH WITH 'S FOR THE MOZILLA DATASET

Metrics	P-Value	Cliff's Delta
Recall-Rate@1	<2.2e-16	1 (large)
Recall-Rate@5	<2.2e-16	1 (large)
Recall-Rate@10	<2.2e-16	1 (large)
MAP	<2.2e-16	1 (large)
MRR	<2.2e-16	1 (large)

Our approach improves the performance of the start-of-the-art approach *NextBug* statistically significantly and substantially for similar bug recommendation. Specifically, our approach achieves an improvement of nearly 60% for the Eclipse dataset and an improvement of nearly 85% for the Mozilla dataset in terms of all the metrics.

RQ2 Does the combined similarity score generated by our approach works better than the three individual similarity scores?

Motivation. We have validated the effectiveness of our approach through the first research question. Our approach clearly outperforms the state-of-the-art baseline for similar bug recommendation. Since our approach combines three similarity scores, in this RQ we want to go further by investigating the recommendation effectiveness of the three individual similarity scores. We want to know whether the combined similarity score generated by our approach is better than the three individual similarity scores.

Methodology. To demonstrate the superiority of the combined similarity score generated by our approach, we compare our approach with three incomplete versions of our approach (sub-approaches) – referred to as *Sub-1*, *Sub-2* and *Sub-3*. For *Sub-1*, we only use the first similarity score (generated by computing cosine similarity of TF-IDF vectors) to recommend similar bugs. The first sub-approach is similar to *NextBug*. The only difference is that *Sub-1* recommends similar bugs only according to the similarity score without setting a threshold. For *Sub-2*, we only use the second similarity score (generated by computing cosine similarity of word embedding vectors) to make recommendation. And for *Sub-3*, we only use the third similarity score, which is based on product and component information in a bug report, to make recommendation.

Results. Table VIII and IX present the effectiveness of our approach compared with the three incomplete versions of our approach (its sub-approaches). From the tables, we can conclude several points.

TABLE VIII
PERFORMANCE OF OUR APPROACH COMPARED WITH THE THREE SUB-APPROACHES FOR THE ECLIPSE DATASET

Metrics	Sub-1	Sub-2	Sub-3	Our Approach
Recall-Rate@1	0.1153	0.0781	0.0280	0.1691
Recall-Rate@5	0.2608	0.1864	0.1082	0.3791
Recall-Rate@10	0.3434	0.2495	0.1777	0.4867
MAP	0.1696	0.1135	0.0636	0.2669
MRR	0.2858	0.1972	0.1417	0.4489

TABLE IX
PERFORMANCE OF OUR APPROACH COMPARED WITH THE THREE SUB-APPROACHES FOR THE MOZILLA DATASET

Metrics	Sub-1	Sub-2	Sub-3	Our Approach
Recall-Rate@1	0.1152	0.0606	0.0933	0.1993
Recall-Rate@5	0.2726	0.1646	0.2858	0.4342
Recall-Rate@10	0.3665	0.2302	0.4043	0.5373
MAP	0.1821	0.1013	0.2204	0.3330
MRR	0.3202	0.1821	0.4073	0.5905

First, our approach performs much better than the three incomplete versions of our approach (its sub-approaches) in terms of all the metrics. Specifically, for the Eclipse dataset our approach improves the best sub-approach by 42%, 57% ,57% for *recall-rate@10*, MAP, MRR respectively, and for the Mozilla dataset our approach improves the best sub-approach by 33%, 51% ,45% for *recall-rate@10*, MAP, MRR respectively. The results indicate that the combined score generated by our approach works better than the three individual similarity scores indeed.

Second, the three sub-approaches have different performances in different datasets and there is no exclusive one that performs the best in both datasets. For example, for the Eclipse dataset the first sub-approach *Sub-1* performs the best, while the third sub-approach *Sub-3* performs the worst in terms of all the metrics. On the contrary, for the Mozilla dataset the third sub-approach *Sub-3* performs the best in terms of four out of the five metrics except *recall-rate@1*. These results highlight that it is reasonable to combine the three individual scores into one final score. By doing so, the weakness of one score can be covered by the strength of the others for different datasets.

To better demonstrate the superiority of our approach over the three sub-approaches, we also compute the p-value (using Wilcoxon signed-rank statistical test) and the Cliff's delta.

TABLE X
P-VALUES AND CLIFF'S DELTA COMPARING THE FIVE METRICS OF OUR APPROACH WITH SUB1 FOR THE ECLIPSE DATASET

Metrics	P-Value	Cliff's Delta
Recall-Rate@1	<2.2e-16	1 (large)
Recall-Rate@5	<2.2e-16	1 (large)
Recall-Rate@10	<2.2e-16	1 (large)
MAP	<2.2e-16	1 (large)
MRR	<2.2e-16	1 (large)

Tables X to XV present the p-values and Cliff's deltas of our approach compared with the three incomplete versions of our approach (its sub-approaches) in terms of five metrics for the Eclipse and Mozilla dataset, respectively. From the tables, we can see the superiority of our approach over its three sub-approaches more clearly. Compared with the sub-approaches, our approach statistically significantly (i.e., p-value < 0.05) and substantially (i.e., Cliff's delta is not negligible) achieves a better performance in terms of all the metrics for both Eclipse and Mozilla datasets.

TABLE XI
P-VALUES AND CLIFF'S DELTA COMPARING THE FIVE METRICS OF OUR APPROACH WITH SUB1 FOR THE MOZILLA DATASET

Metrics	P-Value	Cliff's Delta
Recall-Rate@1	<2.2e-16	1 (large)
Recall-Rate@5	<2.2e-16	1 (large)
Recall-Rate@10	<2.2e-16	1 (large)
MAP	<2.2e-16	1 (large)
MRR	<2.2e-16	1 (large)

TABLE XII
P-VALUES AND CLIFF'S DELTA COMPARING THE FIVE METRICS OF OUR APPROACH WITH SUB2 FOR THE ECLIPSE DATASET

Metrics	P-Value	Cliff's Delta
Recall-Rate@1	<2.2e-16	1 (large)
Recall-Rate@5	<2.2e-16	1 (large)
Recall-Rate@10	<2.2e-16	1 (large)
MAP	<2.2e-16	1 (large)
MRR	<2.2e-16	1 (large)

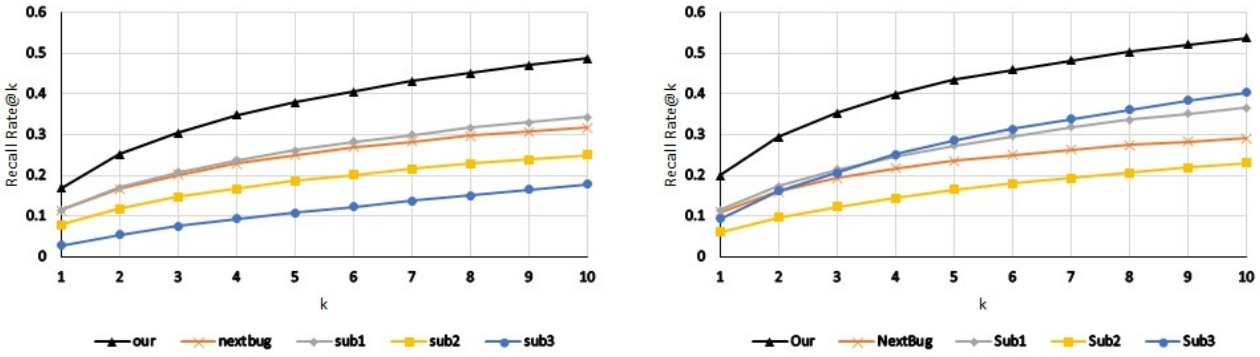
The combined similarity score generated by our approach works better than the three individual similarity scores. Our approach statistically significantly and substantially achieves a better performance in terms of all the metrics for both Eclipse and Mozilla datasets, which highlights the benefit of combining the three scores together.

D. Discussion

The above two research questions has shown the effectiveness and benefit of a design decision of our approach by comparing it with the start-of-the-art approach $NextBug$ and the three incomplete versions of our approach. As mentioned before, a recommendation system can generate top-k recommendation in which k is a variable. Therefore, we also vary the value of k from 1 to 10 to better investigate the performance of our approach for similar bug recommendation.

We use the metric *recall-rate@k* and we plot five curves on one chart representing five approaches, i.e., our approach, $NextBug$, *Sub-1*, *Sub-2* and *Sub-3*. From the curves, we can observe how the recommendation performance improves with the increase of k.

Figure 4 shows the recommendation performance of the five approaches when varying the parameter k in top-k recommendation for both datasets. From the figure, we can see that the recommendation performance of our approach steadily rises with the increase of k and is better than those of the other four approaches by a large margin.



(a) Eclipse

(b) Mozilla

Fig. 4. The recall-rate@k of five approaches when varying the parameter k in top-k recommendation on the Eclipse and Mozilla datasets

TABLE XIII
P-VALUES AND CLIFF'S DELTA COMPARING THE FIVE METRICS OF OUR APPROACH WITH SUB2 FOR THE MOZILLA DATASET

Metrics	P-Value	Cliff's Delta
Recall-Rate@1	<2.2e-16	1 (large)
Recall-Rate@5	<2.2e-16	1 (large)
Recall-Rate@10	<2.2e-16	1 (large)
MAP	<2.2e-16	1 (large)
MRR	<2.2e-16	1 (large)

TABLE XIV
P-VALUES AND CLIFF'S DELTA COMPARING THE FIVE METRICS OF OUR APPROACH WITH SUB3 FOR THE ECLIPSE DATASET

Metrics	P-Value	Cliff's Delta
Recall-Rate@1	<2.2e-16	1 (large)
Recall-Rate@5	<2.2e-16	1 (large)
Recall-Rate@10	<2.2e-16	1 (large)
MAP	<2.2e-16	1 (large)
MRR	<2.2e-16	1 (large)

TABLE XV
P-VALUES AND CLIFF'S DELTA COMPARING THE FIVE METRICS OF OUR APPROACH WITH SUB3 FOR THE MOZILLA DATASET

Metrics	P-Value	Cliff's Delta
Recall-Rate@1	<2.2e-16	1 (large)
Recall-Rate@5	<2.2e-16	1 (large)
Recall-Rate@10	<2.2e-16	1 (large)
MAP	<2.2e-16	1 (large)
MRR	<2.2e-16	1 (large)

E. Threats to Validity

Threats to construct validity relate to the suitability of our evaluation metrics. We use three metrics, i.e., recall-rate@k, MAP (Mean Average Precision) and MRR (Mean Reciprocal Rank). These metrics are commonly-used in information retrieval tasks in software engineering [3], [4], [5], [2], [6]. Thus, we believe there is little threat to construct validity.

Threats to internal validity relate to errors in our experiments. When building ground truth and linking bug reports and committed file lists, two large databases are involved. One is bug report database which contains a total of 763,729 bug reports, and the other is commit log database which contains a total of 3,838,708 commit logs. There may be some errors when processing so much data. We have double checked our implementations and also inspect part of ground truth to ensure they are truly similar bugs. Hence, we believe there are minimal threats to internal validity.

Threats to external validity relate to the generalizability of our results. We have evaluated our approach on two popular open source projects, i.e., Eclipse and Mozilla. In the future, we plan to reduce this threat further by analyzing more datasets from more open source projects.

VI. RELATED WORK

In this section, we first highlight the most related work about similar bug recommendation in Section VI-A. Then we present several works about duplicate bug recommendation, which is related to similar bug recommendation, in Section VI-B. Next, we present several other works about bug report management in Section VI-C. At last, we introduce some software engineering studies that also leverage word embedding in Section VI-D.

A. Similar Bug Recommendation

Similar bug recommendation is a newly-proposed task. The most related work to ours is the recent study by Rocha et al. [2]. They propose a recommender named *NextBug* which relies on standard information retrieval techniques for preprocessing documents (i.e., titles of bug reports) and calculating cosine similarity between documents (i.e., query bug and all pending bugs). After having all the similarity scores, *NextBug* sets a threshold to recommend similar bugs. In our work, we compare our approach with *NextBug* since it is the first and state-of-the-art approach. Rocha et al. have shown that *NextBug* outperforms a state-of-the-art duplicated bug detection technique *REP* for similar bug recommendation.

B. Duplicated Bug Recommendation

There are a number of studies focusing on the recommendation of duplicate bug reports [7], [8], [9], [10]. Runeson et

al. use three similarity metrics, i.e., cosine, dice and Jaccard similarity, to identify potential duplicates of a given bug report [7]. Wang et al. analyze both the textual contents in bug reports and execution information to detect if two bug reports are duplicates of each other [8]. They find that execution information is very useful and can largely improve the performance of duplicate bug report detection. Sun et al. leverage a discriminative model to identify duplicate bug reports [9]. In a later work, they propose a retrieval function REP to improve the performance of their previous model [10]. REP utilizes more information available in bug reports and extends BM25F to measure text similarity more accurately.

In our work, we consider a related but different problem, i.e., similar bug recommendation. The condition for similar bugs is much looser than that of duplicate bugs.

C. Bug Report Management

There are many studies on other bug report management tasks [24], [25], [26], [27], [28], [29], [30], [31], [32], [33]. We highlight three categories of them below.

There are many studies on bug categorization in the literature [31], [34], [35]. Vcubranic proposes to apply text categorization techniques to assist in bug triage [31]. Huang et al. propose a novel Orthogonal Defect Classification (ODC) system by integrating experts' experience and domain knowledge [34]. Thung et al. propose a text mining solution that can categorize bugs into various types [35]. They compare six classic classification algorithms and conclude that SVM achieves the best performance for automatic bug categorization.

There are a number of studies on re-opened bug identification by leveraging machine learning algorithms [32], [36], [37]. Shihab et al. study re-opened bugs on three projects and propose prediction models based on decision trees [32]. They use sampling methods to handle the imbalanced datasets. Xia et al. investigate the performance of different supervised learning algorithms for re-opened bug identification [36]. They find that bagging of decision trees achieves the best performance. In a later work, Xia et al. propose a novel approach *ReopenPredictor* which extract more textual features from the bug reports [37]. The approach automatically estimate thresholds to maximize the identification performance.

Also, there are some studies that predict the severity of bug reports [33], [38], [39]. Menzies and Marcus propose a novel automated method called $SEVERIS$ [33]. The method is based on text mining and machine learning techniques and it is applied to predict the severity of bug reports from NASA. Lamkanfi et al. investigate whether the severity of a reported bug can be accurately predicted by analyzing its textual description using text mining algorithms [38]. Different from Menzies and Marcus, Lamkanfi et al. focus on coarse-grained severity levels (i.e., severe and not-severe) rather than fine-grained ones. In a later work, they go further to compare four well-known text mining algorithms to accurately predict the severity of bug reports [39].

Similar to the above studies, the goal of this work is also to help developers better manage bug reports, which are often too many for developers to deal with [1]. We consider an orthogonal research problem than the above studies though, namely similar bug recommendation.

D. Studies Leveraging Word Embedding

To our best knowledge, there are only a few software engineering studies that leverage word embedding techniques to solve software engineering tasks [6], [18], [40]. We highlight two of them below. Ye et al. propose an approach which projects natural language statements and code snippets as meaningful vectors in a shared representation space [6]. Their empirical evaluations show that word embedding techniques lead to improvements in information retrieval tasks for software engineering. Chen et al. present a novel approach to recommend analogical libraries in Q&A discussions [18]. The approach incorporates relational and categorical knowledge into word embedding and achieves a good performance in analogical libraries recommendation.

In this work, we are the first to leverage word embedding techniques for similar bug recommendation. We consider a different problem than the problems addressed by prior software engineering studies that also leverage word embedding techniques.

VII. Conclusion and Future Work

In the paper, we propose a novel approach to recommend similar bugs. The approach combines a traditional information retrieval technique and a word embedding technique, and takes bug report titles and descriptions as well as product and component information in bug reports into consideration. With preprocessed bug report documents (i.e., bug titles and descriptions), we build TF-IDF vectors and word embedding vectors and calculate two similarity scores based on them respectively. In addition, we calculate a third similarity score based on product and component information. Finally, we combine the three similarity scores into one final score and recommend similar bugs with it. We compare our approach with a state-of-the-art approach named $NextBug$ and perform experiments on two large open source software projects, i.e., Eclipse and Mozilla, containing a total of 763,729 bug reports (389,975 bug reports in Eclipse and 373,754 bug reports in Mozilla). The experimental results show that our approach achieves better performance than $NextBug$ for similar bug recommendation. In particular, our approach achieves an improvement of nearly 60% for the Eclipse dataset and an improvement of nearly 85% for the Mozilla dataset in terms of all the metrics.

In the future, we plan to perform experiments on more datasets to reduce the threats to external validity. We also plan to further improve the approach by optimizing the parameters of the word embedding technique.

Acknowledgment. This research is supported by NSFC Program (No.61602403 and 61572426) and National Key Technology R&D Program of the Ministry of Science and Technology of China under grant 2015BAH17F01.

REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *Proceedings of the 2005 OOPSLA workshop on Eclipse Technology eXchange, ETX 2005, San Diego, California, USA, October 16-17, 2005*, 2005, pp. 35–39.

[2] H. Rocha, M. T. Valente, H. Marques-Neto, and G. C. Murphy, "An empirical study on recommendations of similar bugs," in *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016.

[3] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *Software Engineering, IEEE Transactions on*, vol. 31, no. 6, pp. 429–445, 2005.

[4] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 689–699.

[5] 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 *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 14–24.

[6] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 404–415.

[7] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007, pp. 499–510.

[8] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 461–470.

[9] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 45–54.

[10] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 253–262.

[11] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 2012, pp. 385–390.

[12] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer: an investigation of how developers spend their time," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015*, 2015, pp. 25–35.

[13] Z. S. Harris, "Distributional structure," *Word*, vol. 10, no. 2-3, pp. 146–162, 1954.

[14] Y. Bengio, H. Schwenk, J.-S. Senécal, F. Morin, and J.-L. Gauvain, "Neural probabilistic language models," in *Innovations in Machine Learning*. Springer, 2006, pp. 137–186.

[15] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 160–167.

[16] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[17] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[18] C. Chen, S. Gao, and Z. Xing, "Mining analogical libraries in q&a discussions," 2016.

[19] X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 287–296.

[20] M. F. Porter, "Snowball: A language for stemming algorithms," 2001.

[21] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM sigsoft software engineering notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[22] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 165–176.

[23] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[24] X. Xia, D. Lo, Y. Ding, J. M. Al-Kofahi, T. N. Nguyen, and X. Wang, "Improving automated bug triaging with specialized topic model."

[25] Y. Tian, D. Lo, X. Xia, and C. Sun, "Automated prediction of bug report priority using multi-factor analysis," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1354–1383, 2015.

[26] X. Xia, D. Lo, E. Shihab, and X. Wang, "Automated bug report field reassignment and refinement prediction," 2015.

[27] 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.

[28] ——, "Accurate developer recommendation for bug resolution," 2013.

[29] X. Xia, D. Lo, M. Wen, E. Shihab, and B. Zhou, "An empirical study of bug report field reassignment," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 174–183.

[30] X. Xia, D. Lo, E. Shihab, X. Wang, and X. Yang, "Elblocker: Predicting blocking bugs with ensemble imbalance learning," *Information and Software Technology*, vol. 61, pp. 93–106, 2015.

[31] D. Čubranić, "Automatic bug triage using text categorization," in *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer, 2004.

[32] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Studying re-opened bugs in open source software," *Empirical Software Engineering*, vol. 18, no. 5, pp. 1005–1042, 2013.

[33] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE, 2008, pp. 346–355.

[34] L. Huang, V. Ng, I. Persing, R. Geng, X. Bai, and J. Tian, "Autoodc: Automated generation of orthogonal defect classifications," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 2011, pp. 412–415.

[35] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 205–214.

[36] X. Xia, D. Lo, X. Wang, X. Yang, S. Li, and J. Sun, "A comparative study of supervised learning algorithms for re-opened bug prediction," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 2013, pp. 331–334.

[37] 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, 2014.

[38] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 1–10.

[39] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE, 2011, pp. 249–258.

[40] B. Xu, D. Ye, Z. Xing, X. Xia, g. Chen, and S. Li, "Predicting semantically linkable knowledge in developer online forums via convolutional neural network," in *31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, 2016.