A Similarity Integration Method based Information Retrieval and Word Embedding in Bug Localization

Shasha Cheng
College of Computer Science Technology
Nanjing University of Aeronautics and Astronautics
Collaborative Innovation Center of Novel Software Technology and Industrialization
Nanjing, China
shasha@nuaa.edu.cn

Xuefeng Yan
College of Computer Science Technology
Nanjing University of Aeronautics and Astronautics
Collaborative Innovation Center of Novel Software Technology and Industrialization
Nanjing, China
yxf@nuaa.edu.cn

Arif Ali Khan
College of Computer Science Technology
Nanjing University of Aeronautics and Astronautics
Collaborative Innovation Center of Novel Software Technology and Industrialization
Nanjing, China
arif.khan@nuaa.edu.cn

***Abstract*—To improve the performance of bug localization, there is necessity to solve the lexical mismatch between the natural language in the bug report and the programming language in the source file. A similarity integration method for bug localization is proposed, in which the similarity between bug report and source file is calculated by information retrieval (IR) and word embedding. More specifically, IR technique is used to collect the exact matches between bug report and source file. The terms in the bug report and the potential source files of different code tokens are connected by word embedding technique, which is used to complement with IR technique. Finally, deep neural network (DNN) is utilized to integrate extracted features to get the correlation between bug reports and source files. The experimental results show that the proposed approach outperforms several existing bug localization approaches in terms of Top N Rank, MAP, and MRR.**

Keywords—software bug localization, information retrieval, word embedding, similarity integration, bug report

I. INTRODUCTION

Software quality is vital for the success of a software project [1]. Developers often allow users and testers to submit bugs to the software bug tracking system (Bugzilla, MantisBT). Bug localization systems sort the source files based on the correlation between the given bug report and source files. The developers check the files from the ranked list to find the relevant faulty files. However, for a large system, the number of defects may range from hundreds to thousands [2], so the effectiveness and timeliness of bug localization will affect the reliability and availability of the software.

The existing bug localization techniques can be categorized into three main groups. The first category is spectrum-based which execute the program and collect its execution information, track the running state of the software system, and localize the possible defetcts [3][4][5]. It is a time-consuming process. The second category is based on information retrieval (IR), which mainly based on the text information of the source code, and usually use bug reports to locate relevant source files. Zhou et al. [1] proposed the BugLocator method, which locates the relevant source files based on a revised vector space model (rVSM) that considers previously fixed similar bug reports. Saha et al. [6] proposed the BLUiR method, which uses the structural information of the source file and the bug report to locate bugs. Wong et al. [7] used the most similar code snippets in a given bug report to represent the source file. They also analyzed stack traces given in the bug report to improve the accuracy of bug localization. Youm et al. [8] proposed a comprehensive method of BLIA, which locates software bugs utilizing texts and stack traces in bug reports, structured information and change histories of source files. All of the above IR-based techniques focus on the term weight of natural language text, however they ignore the semantic similarity which could improve the bug localization accuracy.

Machine learning is a well-known approach applied for bug localization. Ye et al. [9] used adaptive learning to sort features from source files, API descriptions, bug-fixing, and change history. Later, Ye et al. [10] introduced a new word embedding method to solve the lexical gap between programming language and natural language, and used the text similarity algorithm proposed by Mihalcea et al. [11] to calculate the semantic similarity between source files and bug reports. However, API documents contain text that involves more general tasks than project-specific defect behavior, which can affect bug localization performance. Recently, bug localization model based on deep learning has also been proposed. Yan et al. [12] used the concepts of enhanced convolution neural network, word embedding and feature detection to locate buggy files to improve the accuracy of bug localization. However, the integration of several deep neural network (DNN) models makes it more complex and difficult to accurately adjust the parameters of the model [12].

Based on the state of the art review of the existing bug localization techniques, the lexical mismatch between the bug report described in natural language and the source file written in programming language will affect the accuracy of bug localization. In IR-based software bug localization, terms are represented discretely and the correlation between source file and bug report is determined by the exact matches between them. Thus, the lexical mismatch problem is particularly important in IR-based bug localization. Word embedding projects bug reports and source files into the embedding space, and the rare or no apparent terms in the query can be retrieved. Moreover, Word2vec is widely used in similar bug recommendation and code search/retrieval. Yang et al. [13] proposed a novel approach to recommend similar bugs, which combines traditional IR technique and word embedding technique. Nguyen et al. [14] indicated that combining traditional IR with Word2Vec in fact

978-1-7281-8913-0/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS51102.2020.00034

achieves better retrieval accuracy. Therefore, a novel similarity integration method for software bug localization is proposed, which IR and word embedding techniques are used to calculate the similarity between source files and bug reports. In addition, the text properties (token matching, stack traces and fixed bug reports) of source files and bug reports are analyzed. In order to capture the nonlinear relationship between features, DNN is utilized to integrate the surface text similarity, semantic similarity and text properties features.

The main contributions of this paper are:

1) A novel similarity integration method for software bug localization is proposed, DNN is utilized to integrate the similarity of IR, word embedding and text properties.

2) The proposed approach is compared with the five existing software bug localization approaches in four dataset. The experimental results show that the proposed approach has statistical significance and substantial improvement.

The remainder of this paper is arranged as follows: The research motivation is introduced in Section II. Section III proposes a similarity method for software bug localization in detail. Next, the similarity integration method is applied to bug localization in Section IV. And experimental demonstration and full comparison between the proposed approach and the benchmark approach in Section V. Finally, the conclusion and future work are described in Section VI.

II. MOTIVATION

In this section, the motivation of this paper is presented. The application of word embedding in bug localization is described in section A. And the text properties in bug localization are introduced in section B.

A. Word embedding in bug localization

Traditional IR models use local representations of terms for query-document matching. In IR-based bug localization, VSM assumes that each word is independent and distributed. It only uses the frequency of each word to describe the text of source file and bug report, without recording any context information between words. The most straight-forward use case for term embeddings in IR is to enable inexact matching in the embedding space [15]. Ye et al. [10] experiments show that word embedding can be effectively applied to software bug localization and achieve good results.

Word embedding refers to the distribution representation of words in a vector space in which similar words are close to each other [16]. Based on a large number of corpus training, each word is mapped to a certain dimension of vector, and the relationship between two words is expressed by cosine distance. The word embedding model record the co-occurrence relationship between words mapped in low dimensional space. At one hand, the word embedding reduces the dimension and the sparseness of vector representation. On the other hand, it can mine the association attributes between words to improve the accuracy of vector semantics. However, the mapping process will cause lots of independent vocabulary information loss by training the co-occurrence of words and mapping them into low dimensional space.

Table I. The specific methods in five IR-based bug localization techniques

Approach	BugLocator	BLUiR	BRTracer	AmaLgam	BLIA
VSM	✓	✓	✓	✓	✓
Structure		✓		✓	✓
Fixed bug reports	✓		✓	✓	✓
Stack traces			✓	✓	✓
Segment			✓		
Version history				✓	✓

Therefore, traditional IR technique focuses on the relationship between different documents in the whole corpus. Word embedding technique pays more attention to the relationship between words and the context in which they appear. These two types of techniques are complement with each other. In this paper, IR and word embedding techniques are used to transform the source file and bug report into digital vector representation.

B. The text properties in bug localization

The five classical bug localization techniques based on IR are analyzed and summarized (BugLocator [1], BLUiR [6], BRTracer [7], AmaLgam [17], BLIA [8]), as show in Table I.

For VSM, each document is expressed as a vector of token weights typically computed as a product of token frequency and inverse document frequency (TFIDF) of each token. Cosine similarity is widely used to determine how close the two vectors are [1]. Important information like class and method names often get lost in the relatively large number of variable names and comments terms due to the term weighting function (TFIDF) [6]. This problem can be overcome by analyzing the structure of source file and bug report and giving higher weight to the exact matches between them. The fixed bug reports mean that user often submits many similar bug reports that correspond to different errors that affect the same buggy program elements [17]. And frequently modified source files will increase the probability of bugs. For stack trace, bug reports often contain stack-trace information, which may provide direct clues for possible faulty files [7]. Research on stack traces such as Schröter et al. [18] shows that 90% of the buggy source files are in the top 10 stack traces. Segment and version control refer to divide a source file into a series of segments and historical data of source file changes, respectively.

From Table I, VSM, structure, fixed bug reports, and stack traces are the most frequently used text properties in bug localization. The proposed approach does not further analyze other information (bug-fixing, code change history and so on) that can be mined from software repositories and bug databases. Therefore, in this paper, the VSM is used to calculate the text similarity between source files and bug reports, and structure, fixed bug reports and stack traces as text properties to improve the accuracy of software bug localization.

III. SIMILARITY INTEGRATION METHOD BASED ON IR AND WORD EMBEDDING

In this section, the proposed approach is elaborated. First, abstract syntax tree (AST) and part of speech (POS) are used to preprocess the source files and bug reports. Then the surface text similarity and semantic similarity of the source file and bug

report are calculated by IR and word embedding. Finally, DNN is used to integrate the analysis data.

A. Data preprocessing

The purpose of text preprocessing is to decompose the bug report and source file into terms that can be analyzed by IR technique. In this paper, the source files are transformed into an AST and the class names, method names, variables and comments can be extracted. Bug reports are also parsed to get summary, description, fixed files and stack traces. The Camel Case splitting [19] is used to segment the combined words. For example, "EventMouse" is split into "Event" and "Mouse". The English stop words such as "is" and "the" in the bug report and source file are removed. The keywords such as "private" and "public" are also removed from the source file. The standard Porter Stemmer[1] performs stem extraction to restore derived words into the root form, so that similar words can appear in the same form after processing.

In the natural language processing, POS tagging helps to make full use of the semantic knowledge contained in English words and sentences, and the important information in software components is contained in nouns [20]. Fig.1 shows the POS tagging results of the summary of the number 80120 AspectJ bug report by Stanford Tagger, and 'CTabFolder', 'layout', 'pixel' and 'right' are nouns. From the extracted nouns, 'CTabFolder' directly specifies the buggy file 'CTabFolder.java'. Therefore, the lexical weight of POS as nouns in source files and bug reports should be increased.

B. Similarity calculation based on IR and word embedding

The surface text similarity and semantic similarity between the bug reports and source files are represented by cosine distance. The similarity between them is represented by the maximum similarity value of the bug report with all methods and the whole document. The steps of calculating similarity between bug reports and source files are described as follows in detail.

Firstly, among a group of pre-trained words in skip-gram, some words may not exist, which are randomly initialized and can be fine-tuned during training. Ye et al. [10] confirmed that training text embedding using Wikipedia corpora and project-specific corpora (Eclipse and Java) has similar performance, and Wikipedia corpora have more vocabulary, which is beneficial to the deep learning model. Therefore, the proposed approach uses the open source Word2vec of google as the training tool, divides the text in the wiki corpus into training data and test data. And then the skip-gram model is used to train and obtain the vectors of each word in the training data. The vectors of each word dimension 100, 200 and 300 are obtained in the training data respectively. Through the experiment, the vector dimension is chosen as 300 to achieve the best effect of similarity calculation.

Secondly, VSM is commonly used to treat each text as a set of vectors, which is used to calculate TFIDF value for all terms in source files and bug reports. The larger source file has a higher error probability [1], the length score of the source file is combined with the result of cosine similarity.

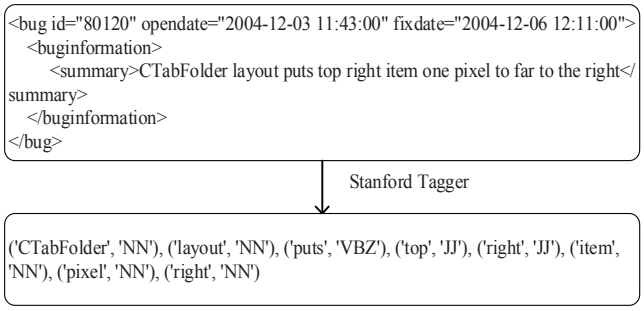

Fig. 1. BugID=80120 using POS results in AspectJ

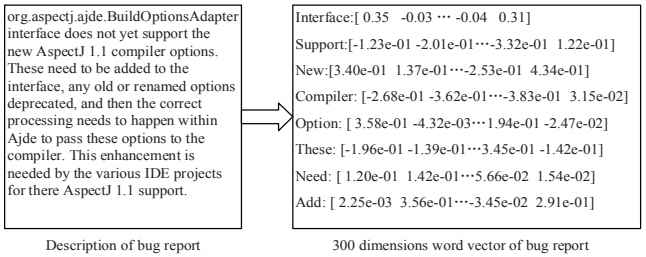

Fig. 2. Transforming description in bug report into vectors using skip-gram

$$\mathrm{LenScore}_s(\#\mathrm{terms}) = \frac{1}{1+e^{-\lambda \times \mathrm{nor}(\#\mathrm{terms})}} \quad (1)$$

In the above equation, #terms refers to the number of terms in the source file s and λ ($\lambda > 0$) adjust the preference for larger files. By setting this parameter, a better balance between increasing large files and reducing noise in large files can be achieved [1]. The value of surface similarity between bug report b and source file s is measured by following the equation.

$$\mathrm{rVSM}(b,s) = \cos(b,s) \times \mathrm{LenScore}_s(\#\mathrm{terms}) \quad (2)$$

Thirdly, bug report contains summary and description, which are composed by the natural language. Fig.2 shows an example of converting description of the number 29769 AspectJ bug report into a 300D numeric vector by using the trained skip-gram.

The source file is consist of various code tokens in the programming language, which is different from the bug report. Some keywords frequently appear in the source code, which may affect the performance of skip-gram. In order to reduce the influence of keywords that frequently appear in the source code, the proposed approach combines skip-gram and TFIDF to represent the source file and bug report by vector. Because the representation method of skip-gram model based on word embedding can mine the associated attributes between words. It will improve the accuracy of vector semantics. And TFIDF has high discrimination for words with high frequency and significant in a small number of documents. In other words, TFIDF can filter out some common but unimportant words while retaining the important words that affect the whole text [20].

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

$$s^* = \frac{1}{|s|}\sum_{i \in s} w_{i,s}\,tfidf_{i,s} \quad (3)$$

$$b^* = \frac{1}{|b|}\sum_{i \in b} w_{i,b} \quad (4)$$

In the above equations, |s| and |b| denote the number of terms in the source file and bug report. $w_{i,s}$ ($w_{i,b}$) is the word vector of a term i in the source file s (bug report b) by using the skip-gram model. $tfidf_{i,s}$ is the TFIDF weight of term i in the source file s. And s^* and b^* represent as numeric vector of the source file and bug report respectively.

Finally, defects are usually located in a small part of the code, for example a method. When the source file is large, its corresponding norm will also be large, which will result in a small cosine similarity with the bug report, even though one method in the file may be actually very relevant for the same bug report [9]. Therefore, the AST is used to divide the source file into methods to calculate the similarity between each method and bug report. Each method m is regarded as a separate document and the cosine distance is used to calculate the similarity between method and bug report. Then the maximum value of the bug report similarity with all methods and the whole document is employed to represent the surface lexical similarity and semantic similarity. The specific equations are as follows:

$$SurfaceSim = \max(\{rVSM(b,s)\} \cup \{rVSM(b,m) | m \in s\}) \quad (5)$$

$$SemanticSim = \max(\{\cos(b^*,s^*)\} \cup \{\cos(b^*,m^*) | m \in s\}) \quad (6)$$

C. DNN for similarity integration

The linear model is difficult to capture the nonlinear relationship between features, it may limit the performance of bug localization. The combination of DNN and nonlinear function is expected to perform better than the linear combination based on IR adaptive learning [21]. DNN is a series of algorithms in the artificial neural network (ANN). The purpose of DNN is to express the high-level abstraction in data by using the model structure with multiple nonlinear transformations. DNN is a forward-propagation ANN. There are many hidden layers between input and output layers, among which the higher layer can combine features from the lower layer. DNN has excellent capacity, it has been successful in dealing with the high complexity nonlinear relationship between input and output. Therefore, DNN is used to combine features, and enough training data are used to learn the weight of features from the nonlinear function.

Positive pairs are created through all the corresponding the fixed bug reports and their buggy files. Moreover, negative pairs are created by selecting source files that are similar in text and do not contain defects for each bug report. The features from each bug report and source file are extracted, feature vectors are respectively constructed using the similarity calculation based on IR and word embedding. The two feature vectors are considered as an input for DNN. The DNN transforms features using nonlinear functions into the hidden layer, and classifies these features by linear functions in output layer. In DNN, the hidden layer has abstract effect, and the number of hidden layers determine the ability of feature extraction. In order to get the best effect of the proposed approach, the number of hidden layers of DNN are set to 1000 and the number of nodes are set to 1000-1100. The output is a separate score to indicate the likelihood that the source file contains defects relative to the bug report.

IV. A SIMILARITY INTEGRATION METHOD FOR SOFTWARE BUG LOCALIZATION

A. Analyze the text properties of bug reports and source files

According to the section of motivation, token matching (structure), stack traces and fixed bug reports are analyzed in the proposed approach.

Token matching: The information such as file name, method names, class names and comments in the source file are used to match the summary and description of the bug report respectively. The similarity between the source file and bug report is represented by the number of token matches.

Stack traces: The regular expression (.*?)\((.*?)\) is used to extract the stack traces from the description of bug report. The reciprocal of the ranking of the source file in the stack traces is used to measure the similarity between the source file and the bug report [7].

Fixed bug reports: The multi-label classification algorithm is applied to each bug report. The terms in the fixed bug reports as input and their located source files as tags. The features are extracted from bug reports to make better use of multi-label classification algorithm. The summary of bug report is an important feature because it tends to use concise sentences to express what the defect is. Therefore, the TFIDF weight of terms in the summary part of the preprocessed bug report is taken as a feature. And the TFIDF weight of POS tag noun set in bug report is taken as another feature. The independent classifier is trained for each label using a multinomial naive Bayes classifier as the base classifier for a one-vs.-the-rest (OvR) method. Then, given the new bug report, it will output the probability score of the source file to be located.

B. Framework of Similarity integration method in bug localization

The similarity integration method based on IR and word embedding is applied to software bug localization, and the overall framework is shown in Fig.3. A better IR-based bug localization technique should include both lexical matching and semantic matching. Thus, on the basis of analyzing the text properties of source files and bug reports, IR and word embedding are further used to calculate the similarity between them to solve the problem of lexical mismatch. The analysis data are integrated by DNN to capture the nonlinear relationship between features. When there is enough training data, the weight of each feature (surface text similarity, semantic similarity and text properties) from the nonlinear combination can be reflected, in order to sort the source files that may contain defects.

Based on the similarity integration method described in above section, DNN is utilized to combine the five features of surface text similarity based on IR, semantic similarity based on word embedding, token matching, stack traces and fixed bug reports, as shown in Fig.4. The result is correlation scores between the bug report and all source files.

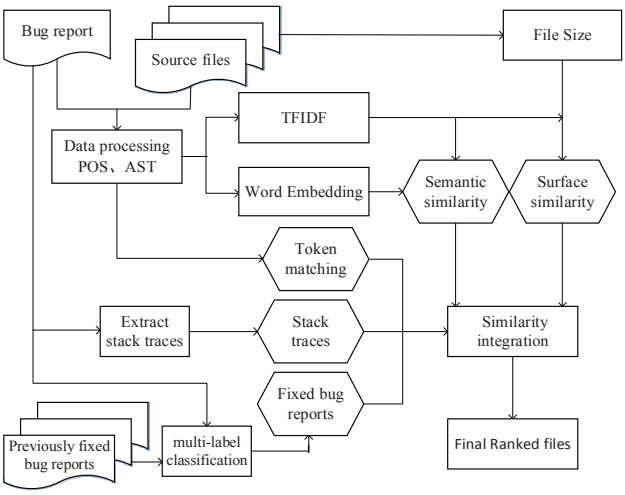

Fig. 3. The Overall Framework of the proposed approach

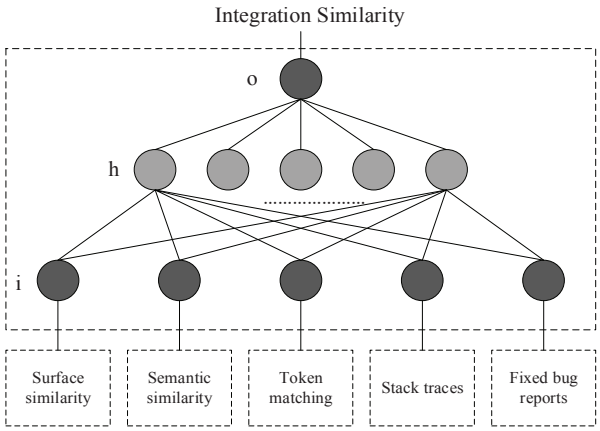

Fig. 4. DNN-based for similarity integration

Table II. The statistics of the Benchmark Dataset

Project	Time Range	#Bug Reports	#Source Files
AspectJ	03/01-01/14	593	4439
Eclipse	10/01-01/14	6495	3545
JDT	10/01-01/14	6247	8184
SWT	02/02-01/14	4151	1552

V. EMPIRICAL EVALUATION

A. Experimental Setting

Benchmark DataSet: In order to evaluate the effectiveness of the proposed approach, the dataset used in experiments come from Ye et al. [9] (Table II). The same dataset have been previously used in other bug localization techniques (i.e. LR+WR [10], DeepLoc [12]).

Because the number of source files in the dataset are very large, there will be a large number of unrelated source files for a bug report, it is impossible to train all negative pairs. Therefore, based on the source file samples used by Ye et al. [9], the top 300 source files in the surface similarity based on IR are selected as negative samples for training. For the final similarity calculation, all the source files in the dataset will be sorted. The bug reports from each benchmark dataset are sequenced chronologically by its submission time, and divide the bug report into two parts, 80% of which are used as the training set (older defect) and the other 20% are used as the test set (newer defect).

B. Evaluation metrics

Three metrics are used to measure the performance of the proposed bug localization technique.

(1) Top N Rank

Top N Rank is the number of bug reports that contain buggy source files that appear in the top N (N=1,5,10) files returned. Given a bug report, if the top N ranking results contain at least one source file needed to repair the bug, the bug is considered to be located. The higher value of the metric, the better performance of bug localization.

(2) Mean Average Precision (MAP)

MAP provides a measure of the quality of software bug localization when there are multiple related files in a query. The average accuracy of a single query is the average of the query accuracy.

$$AvgP_i = \sum_{i=1}^{M} \frac{P(j)\times pos(j)}{\text{number of positive instances}} \quad (7)$$

In the above formula, j is the rank, M is the number of retrieved instances, pos(j) indicating whether the instances j are relevant. P(j) is the accuracy at the end of the ranking j, defined as follows:

$$P(j) = \frac{\text{number of positive instances in top j position}}{j} \quad (8)$$

The MAP for a series of queries is the mean of the average accuracy of all queries. The higher value of the MAP, the better performance of bug localization.

(3) Mean Reciprocal Rank (MRR)

MRR is an evaluation statistics of a series of possible results produced by multiple queries. The reciprocal ranking of a query is the reciprocal of the ranking of the first correct answer, calculated as follows:

$$MRR = \frac{1}{M}\sum_{i=1}^{M} \frac{1}{\text{f-rank}_i} \quad (9)$$

The total number of bug reports is M, f-rank_i indicating the position of the corresponding buggy source file in the sorted list of the i bug report. The higher value of the MRR, the better performance of bug localization.

C. Results and Analysis

In order to comprehensively evaluate the performance of the proposed approach and its components in bug localization, the following research questions are answered.

RQ1: How effective is the proposed approach for bug localization and it outperforms than other bug localization approaches?

RQ2: Does the integrated similarity score generated using the proposed approach works better than the five incomplete versions similarity scores?

To answer the above questions, the proposed approach is applied to four dataset (Table II), and a list of buggy files are returned. The performance of the proposed approach is evaluated by Top N Rank, MAR, and MRR.

1) Research and analysis of RQ1:

To verify the effectiveness of the proposed approach, it compared with the five bug localization approaches, i.e. BugLocator [1], LR [9], LR+WE [10], DNNLoc [22] and DeepLoc [12]. BugLocator is the bug localization based on IR, LR and LR+WE are based on machine learning, DNNLoc and DeepLoc are based on deep learning.

Table III shows the overall performance comparison of the proposed approach with the given five bug localization approaches. And the proposed approach has improved the performance on Top N Rank, MAP and MRR. The Table III shows that the proposed approach has improved by more than 50% on Top@N (N=1,5,10), MAP and MRR on average compared with BugLocator. This is because BugLocator only uses rVSM and fixed bug reports to calculate the similarity between source files and bug reports, but the proposed approach also considers token matching, stack traces and semantic similarity. Therefore, using IR for further analysis of text properties and semantic similarity can greatly improve the performance of bug localization.

Compared with LR, the proposed approach increased the performance by 21.2-68.1% on Top@5, MAP and MRR raised by 47.0% and 54.0% respectively. Compared with LR+WE, Top@1 and Top@5 increased by 19.2%-77.0% and 3.0-31.8% respectively, and MAP and MRR are also increased. LR and LR+WE pay more attention to using APIs as additional information in the source file to make up for lexical mismatch. However, surface text similarity and text properties are not considered.

Table III. Consist of the results for the proposed approach, BugLocator, LR, LR+WE,DNNLoc and DeepLoc. The most effective approaches for the four dataset are highlighted in bold font

Project	Approach	Top@1	Top@5	Top@10	MAP	MRR
AspectJ	Our Approach	46.1%	**76.5%**	84.2%	0.38	**0.60**
	BugLocator	22.0%	46.0%	58.0%	0.28	0.36
	LR	20.2%	45.5%	61.1%	0.25	0.33
	LR+WE	29.0%	58.0%	74.0%	0.30	0.45
	DNNLoc	**47.8%**	71.2%	**85.0%**	0.32	0.52
	DeepLoc	45.0%	71.0%	80.0%	**0.42**	0.51
Eclipse	Our Approach	**47.2%**	**74.7%**	**83.5%**	**0.48**	**0.59**
	BugLocator	29.0%	50.0%	60.0%	0.33	0.38
	LR	36.5%	60.1%	70.7%	0.40	0.47
	LR+WE	39.0%	60.0%	71.0%	0.40	0.46
	DNNLoc	45.8%	70.5%	78.2%	0.41	0.51
	DeepLoc	45.0%	70.0%	79.0%	0.43	0.53
JDT	Our Approach	**48.9%**	**67.0%**	**81.9%**	**0.49**	**0.58**
	BugLocator	19.0%	40.0%	51.0%	0.29	0.37
	LR	30.0%	55.2%	68.1%	0.34	0.42
	LR+WE	41.0%	65.0%	75.0%	0.42	0.52
	DNNLoc	40.3%	65.0%	74.3%	0.34	0.45
	DeepLoc	43.0%	65.0%	77.0%	0.44	0.53
SWT	Our Approach	**60.2%**	**82.7%**	**89.8%**	**0.62**	**0.70**
	BugLocator	22.0%	39.0%	52.0%	0.27	0.31
	LR	28.3%	58.2%	70.0%	0.36	0.41
	LR+WE	34.0%	57.0%	71.0%	0.38	0.45
	DNNLoc	35.2%	69.0%	80.3%	0.37	0.45
	DeepLoc	39.0%	66.0%	77.0%	0.40	0.49

Therefore, compared with bug localization based on IR and machine learning, the conclusion can be drew that both IR and word embedding techniques are compatible with each other, and their combination could improve the accuracy of bug localization. In addition, LR and LR+WE use learning to ranking for linear combination of similarity. And in this paper, DNN is used for similarity integration. As shown in Table III, it can be found that the latter can further improve the performance of software bug localization.

In the dataset AspectJ, Top@1, Top@10 are lower than DNNLoc and MAP is lower than DeepLoc. This is because the bug fixing recency and frequency used in DNNLoc and DeepLoc and the change history of source files for bugs contains useful information for identification of fault-prone files. Therefore, in the future, the change history of source files are taken into account to further improve the accuracy of our proposed approach. Moreover, compared with DNNLoc, MAP and MRR increased by 27.9% and 28.9% respectively. Top@1, Top@5, Top@10 are on the whole increased. Compared with DeepLoc, Top@1, Top@5, Top@10 and MRR increased by 18.8%, 10.6%, 9.1% and 20.2% respectively. The result shows that the DNN model can be trained better in the case of a large number of bug reports and sample pairs, and the proposed approach is basically better than the deep learning based bug localization.

2) Research and analysis of RQ2:

For verify the performance of the features combination similarity score produced by the proposed approach, it compared with five incomplete versions, which are called Sub1 (only using rVSM to calculate surface text similarity), Sub2 (only using word embedding to calculate semantic similarity), Sub3 (only using text properties to analysis the relevance between bug reports and source files), Sub1+Sub3 (rVSM and text properties) and Sub2+Sub3 (word embedding and text properties).

From Fig.5 to Fig.10, the proposed approach performs better than the five incomplete versions in all aspects of measurement. Sub1 is obviously the most effective method in this paper, thus using rVSM to calculate surface text similarity can effectively improve the accuracy of bug localization. At the same time, only calculating semantic similarity itself cannot achieve high precision. But further analysis Sub2 based on Sub1+Sub3, Top@5, Map and MRR are increased by 9.2%, 7.1% and 7.3% respectively. Consequently, the analysis of semantic similarity is also significantly important for software bug localization. By adding Sub3 to Sub1 (Sub1+Sub3), MAP and MRR increased by 9.4% and 10.0% respectively. Add Sub3 to Sub2 (Sub2+Sub3), all metrics have improved significantly. Therefore, it is of great significance to analyze the text properties of source files and bug reports based on text similarity and semantic similarity.

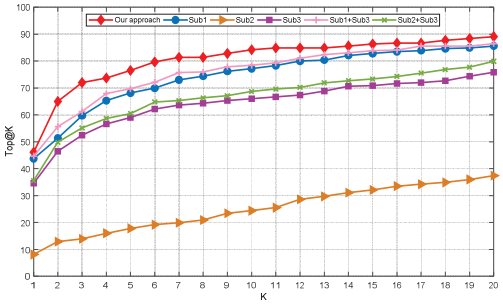

Fig. 5. Top@K Accuracy with the proposed approach and five sub-approach for the AspectJ dataset

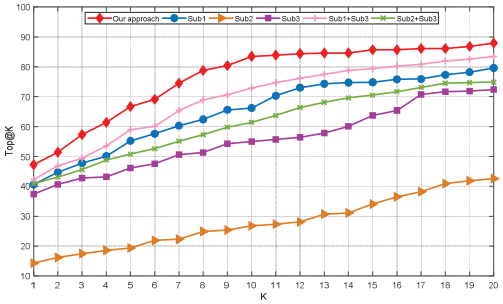

Fig. 6. Top@K Accuracy with the proposed approach and five sub-approach for the Eclipse dataset

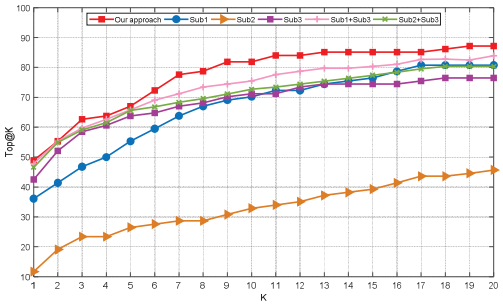

Fig. 7. Top@K Accuracy with the proposed approach and five sub-approach for the JDT dataset

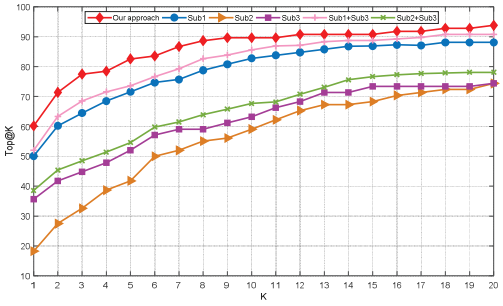

Fig. 8. Top@K Accuracy with the proposed approach and five sub-approach for the SWT dataset

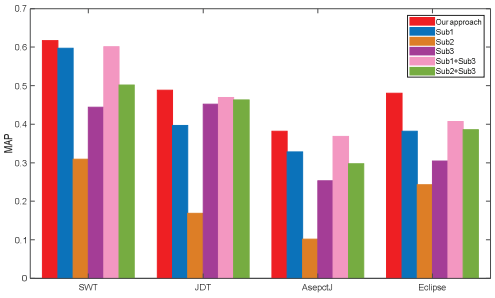

Fig. 9. MAP Accuracy with the proposed approach and five sub-approach

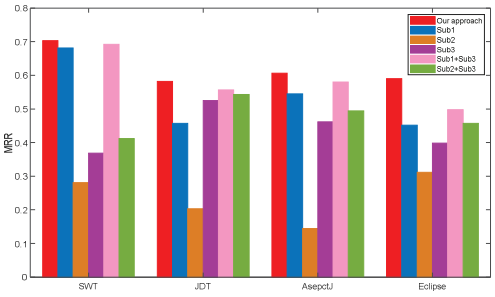

Fig. 10. MRR Accuracy with the proposed approach and five sub-approach

Five incomplete versions have different performance in different dataset. These results show that it is reasonable to use DNN to integrate the analysis data to capture the non-linear relationship between features. Moreover, combine Sub1, Sub2 and Sub3 into a final score through DNN, and the weakness of one part in different dataset can be made up by other parts.

D. Threats to Validity

In this section, the effectiveness and generalizability of the proposed approach are discussed.

The internal validity is related to the experimental error and the implementation of the proposed method. The quality of bug report will affect the performance of bug localization. In the future, query expansion, query replacement, term selection [23] will be learned to reformulate the bug report to improve the accuracy of the proposed approach.

The external validity indicates the generalizability of the proposed approach. In this paper, experimental dataset are used in the previous research [9][10][12], which are open source projects by java language. However, the proposed approach is not applied to other open source projects or industrial projects. Therefore, the proposed approach will be applied to open source projects for other programming languages in the future.

The construct validity refers to the applicability of metrics used in experimental evaluation. To reduce this threat, three metrics including Top N Rank, MRR and MAP are used in this paper. These metrics are widely used in related researches [1,2,6-10], which adequately capture different aspects of performance and there is little threat to construct validity.

VI. CONCLUSION AND FUTURE WORK

The developers of software project need to locate relevant faulty files for each received bug report to solve the system problems. However, it is inefficient to search a large number of source files. Bug localization techniques attempt to automate the process and sort the relevant source files for each bug report.

In this paper, a novel similarity integration method for software bug localization is proposed to solve the problem of lexical mismatch in IR based bug localization. rVSM is used to calculate the surface text similarity between the bug reports and source files. The skip-gram model and TFIDF are applied to convert the source files and bug reports into digital vectors to express the semantic similarity between them. Moreover, the text properties of the source files and bug reports are also taken into consideration. Finally, DNN is used to integrate the above features to get the sorting list of source files related to the bug report. The proposed approach is verified in four dataset, and experimental results show that IR and word embedding are compatible each other, for achieving bug localization with higher accuracy. The empirical results further reveal that the proposed approach has better performance than several existing bug localization approaches based on IR and machine learning techniques.

In the future, the features of source files and bug reports will be analyzed, such as the call relationship between source files, the frequency of bug fixing, etc. Using more meaningful features to improve the accuracy of bug localization. Many deep learning methods will be explored to further improve the performance of the proposed approach. Finally, the proposed approach will be applied to other dataset to verify its effectiveness.

ACKNOWLEDGMENT

This work is supported by the National Key R&D Program of China: Key Technology Research and Platform Development for Cloud Manufacturing Based on Open Architecture, under Grant No.: 2018YFB1702700.

REFERENCES

[1] J. Zhou, H. Zhang, and D. Lo. “Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports,” International Conference on Software Engineering, IEEE, 2012, pp. 14–24.

[2] Khatiwada, S., Tushev, M., & Mahmoud. “A. Just enough semantics: An information theoretic approach for IR-based software bug localization,” Information and Software Technology, 2017, pp. 45–57.

[3] Abreu, Rui, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. “A practical evaluation of spectrum-based fault localization,” Journal of Systems & Software, vol. 82, no. 11, Nov. 2009, pp. 1780–1792.

[4] J. A. Jones and M. J. Harrold. “Empirical evaluation of the tarantula automatic fault-localization technique,” International Conference on Automated Software Engineering, IEEE, 2005, pp. 273–282.

[5] J. A. Jones, M. J. Harrold, and J. Stasko. “Visualization of test information to assist fault localization,” International Conference on Software Engineering, IEEE, 2002, pp. 467–477.

[6] R.K. Saha, M. Lease, S. Khurshid, D.E. Perry. “Improving bug localization using structured information retrieval,” IEEE/ACM International Conference on, ACM, 2013, pp. 345–355.

[7] Wong, C.P., Xiong, Y., Zhang,H., Hao, D., Zhang,L. & Mei, H. “Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis,” International conference on software maintenance and evolution, IEEE, 2014, pp. 181–190.

[8] Youm KC, Ahn J, Lee E. “Improved bug localization based on code change histories and bug reports,” Information & software Technology, vol. 82, 2017, pp. 177-192.

[9] X. Ye, R. Bunescu, and C. Liu. “Learning to rank relevant files for bug reports using domain knowledge,” International Symposium on Foundations of Software Engineering, ACM, 2014, pp. 689–699.

[10] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu. “From word embeddings to document similarities for improved information retrieval in software engineering,” International Conference on Software Engineering (ICSE), 2016, pp. 404–415.

[11] Mihalcea, Rada, C. Corley, and C. Strapparava. “Corpus-based and knowledge-based measures of text semantic similarity,” National Conference on Artificial Intelligence & the Eighteenth Innovative Applications of Artificial Intelligence Conference, 2006, pp. 775–780.

[12] Yan Xiao, Jacky Keung, Kwabena E. Bennin, Qing Mi. “Improving bug localization with word embedding and enhanced convolutional neural networks,” Information & Software Technology, vol 105, 2019, pp. 17-29.

[13] Yang, Xinli, David Lo, Xin Xia, Lingfeng Bao, and Jianling Sun. “Combining Word Embedding with Information Retrieval to Recommend Similar Bug Reports,” IEEE International Symposium on Software Reliability Engineering, IEEE, 2016.

[14] T. V. Nguyen, A. T. Nguyen, H. D. Phan, T. D. Nguyen, and T. N. Nguyen. “Combining word2vec with revised vector space model for better code retrieval,” International Conference on Software Engineering Companion, 2017, pp. 183–185.

[15] Bhaskar Mitra and Nick Craswell. “Neural Models for Information Retrieval,” arXivpreprint arXiv: 1705.01509, 2017.

[16] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. “Distributed representations of words and phrases and their compositionality,” Advances in Neural Information Processing Systems, 2013, pp. 3111–3119.

[17] Wang, S., & Lo, D. “AmaLgam+: Composing rich information sources for accurate bug localization,” Journal of Software: Evolution and Process. 2016, pp. 921–942.

[18] Schröter A, Bettenburg N & Premraj. “Do stack traces help developers fix bugs?” IEEE working conference on mining software repositories, IEEE, 2010, pp. 118-121.

[19] Binkley, David W., Marcia Davis, Dawn J. Lawrie, and Christopher Morrell. “To camelcase or under_score,” IEEE International Conference on Program Comprehension, IEEE, 2009, pp. 158–167.

[20] Capobianco, G., Andrea De Lucia, R. Oliveto, Annibale Panichella, and S. Panichella. “Improving IR-Based Traceability Recovery via Noun-Based Indexing of Software Artifacts,” Journal of Software Maintenance and Evolution: Research and Practice, vol. 26, 2013, pp. 743–762.

[21] Arisoy, Ebru, Tara N. Sainath, Brian Kingsbury, and Bhuvana Ramabhadran. “Deep Neural Network Language Models,” Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-Gram Model? On the Future of Language Modeling for HLT, 2012, pp. 20–28.

[22] Lam, An Ngoc, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. “Bug Localization with Combination of Deep Learning and Information Retrieval,” 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), 2017, pp. 218–229.

[23] Rahman, Mohammad Masudur, and Chanchal K. Roy. “Improving IR-Based Bug Localization with Context-Aware Query Reformulation,” Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 621–632.