A Word Embedding Model for Fault Localization using Bug and Software Change Repositories

Aqib Rehman
Computer Science Department
Quaid-i-Azam University
Islamabad, Pakistan
a.rehman@cs.qau.edu.pk

Muddassar Azam Sindhu
Computer Science Department
Quaid-i-Azam University
Islamabad, Pakistan
masindhu@qau.edu.pk

Onaiza Maqbool
Computer Science Department
Quaid-i-Azam University
Islamabad, Pakistan
onaiza@qau.edu.pk

***Abstract*—Software developed and then deployed in a real world environment is inevitable to exhibit some undesirable behavior. Therefore, developers need to provide maintenance facilities to enable the bugs causing the undesirable behavior to be fixed. However, prior to fixing the bug, the suspicious part of the code needs to be identified. For this purpose, they usually perform fault localization. This can be done manually as well as automatically. Several techniques exist in the literature for fault localization. However, most of them are static based techniques because they do not depend on a specific programming language along with the possibility to work on underdeveloped software and some other benefits. These techniques are largely based on lexical matching of terms which leads to mismatch of terms, large precision value because of limited vocabulary of a programming language and some techniques consider the semantics but it is computationally expensive to localize faults through this. In this paper we have proposed a fault localization technique which is based on the machine learning concept of word embedding. Our proposed approach aims at looking at the relatedness between the bug terms and source code artifact. We mined the bug repositories and software change repositories to train the word embedding model on the mined repositories data. On the arrival of a new bug, the cluster of the bugs from the model is searched and the files from the software change repositories are retrieved which are used for fixing those bugs. We have compared the results of our approach with the latest technique proposed in year 2018 Pointwise Mutual Information (PMI) and Normalized Google Distance (NGD) which consider the context and also with existing lexical techniques Vector Space Model (VSM) and the semantic based method Latent Semantic Indexing (LSI). We have used the benchmark dataset "MoreBugs" which has been widely used in this domain. The results show that our approach outperforms other techniques.**

Keywords—word embedding, fault localization, bug repositories, IR

I. INTRODUCTION

In the recent decades there has been an enormous shift in the world of technology. Humans are relying more and more on machines to perform their day to day activities. Machines are helping humans in mundane tasks like opening doors to critical tasks like flying airplanes. Depending on the task that needs to be performed, software is installed to make the machines work. Software is programmed in different languages and its complexity depends on the underlying computation that it needs to perform. It can be as simple as an addition software or as complex as a software for a robot.

With the increasing demands of software in the market, a great amount of importance is placed on its quality [1]. Software development companies aim at creating software that is of the best possible quality. This not only refers to the interface and response time of a software but also to the low error rate of a software. The ideal scenario when creating a software is to create an error free software but this is not possible because it is inevitable that a software might depict undesirable behavior at some point. To tackle this problem a software is tested at every stage of development. If an error is not caught at an early stage of development, then chances are that over time it would become difficult to resolve it and the cost to fix it would also increase drastically.

Despite this error discovery in a software does not end when a software is deployed in the real world environment. The developers have to interact with the end- users during software maintenance for removal of possible defects. On interaction with the software, users can come across situations which might lead to a software behaving undesirably. In order to keep the users trust and satisfaction level, developers have to provide maintenance facilities to the users through which the users can convey their problems to the developers and the developers in turn have to provide solutions. The task of resolving a problem is relatively easy for the developers as compared to locating the actual part of code in the software which is causing the problem. This task of locating an error is called fault localization.

Researchers have documented various techniques through which fault localization can be achieved. The techniques for fault localization can be divided into two groups: static based techniques and dynamic based techniques. Static based techniques have some advantages over dynamic based techniques as they are independent of programming language and they do not require a software to be completely developed. Till now the majority of ways to perform static based techniques incorporate lexical matching. Lexical matching deals with the similarity between bug reports terms and artifacts of source code. The terms of a programming language are limited so this leads to less terms being matched with the reports and it causes the vocabulary mismatch problem. Also, due to limited terms of programming language it leads to large precision value. All of this leads to low accuracy. Apart from this there are some semantic based techniques available like LSI [2] and LDA [3] but they are computationally expensive [4]. LSI is a technique which calculate the similarity of a term with a unstructured text. It constructs the matrix of x*y where x is the terms and y is the documents containing text. LDA finds

out the word-topic probability distribution and topic-document distribution. Word-topic probability distribution includes the probability of every word of vocabulary belonging to topic of model. Topic-document probability distribution includes all the documents of document collection belonging to topic of model. In [4] authors have introduced a method to capture relatedness between bug terms and source code. They have also given a comparison with the existing static and dynamic based techniques and concluded that their proposed approach is better.

In this paper we have taken motivation from [4] and proposed a word embedding model which will find similar concepts based on their relatedness and group them together in the form of clusters. Word embedding is a technique of machine learning. We have mined the bug repositories. And trained the word embedding model with the bug repositories data. Now on arrival of a new bug, the terms of the bug report will be provided to the model as a testing and the similar and related terms in the form of clusters will be retrieved from the model. Based on these terms suspicious files containing the bug would be suggested.

Rest of the paper is organized as follows: Section II details the related work. After that in Section III We have explained our proposed approach for fault localization through word embedding. In Section IV we have documented the results of our experiments and explained our insights. Finally, Section V concludes our paper and gives future directions in which our proposed approach can be used.

II. Related Work

Fault localization is the task of locating suspicious part of the software which is causing the bugs after an undesirable behavior has been detected. Users report the undesirable behavior and now it is the task of the developers to provide assistance in getting it resolved. Developers have been providing this facility of resolving bugs ever since the existence of software. They have used various techniques which have evolved over the years.

In the beginning, developers used to make use of print statements in order to locate the faulty part of the code. After that break points were added and the code was debugged to narrow down the faulty part of code. These techniques required a lot of manual labor and consumed more time [5]. With time the techniques for fault localization were automated. This created a lot of ease and helped us in locating bugs quicker than before. Furthermore, the error rate was also reduced. These techniques can be divided into two broad categories i.e. static based and dynamic based. Dynamic based approaches require a functional system to localize the faults. By executing the test cases, the pass and fail traces are identified and through this the faulty part of code is localized. In [6-11] authors have used different modifications of dynamic based techniques. In spite of all the work done researchers have placed a greater focus on static based techniques because of their advantage over dynamic based techniques. Static based techniques can be use on a partially developed system, are independent of programming language and do not require the creation and execution of test cases. Our approach falls in the category of static based technique so we have only discussed static based approaches from the literature in detail here.

In static based techniques different Information Retrieval (IR) methods are used for fault localization. The conventional methods are based on lexical matching [12]. The contents from the bug report which are reported by the end user when he/she experiences an undesirable behavior from the system is matched with source code artifacts. Finally, the piece of code is suggested to the developer which is faulty.

In [13] authors used the IR methods for finding the traceability links between the source code and the different documentation of the software. The terms of documentation which are written in natural language (NL) are matched with the source code artifacts. They considered the Vector Space Model (VSM) and probabilistic model for that purpose. They tested their approach on two subject systems amongst which one is in C++ language and the other is in Java language. The results show that the approach gives 100% recall.

In [14] authors developed a system for a clinic. It takes the image as an input and using the contents of that image i.e. color, shape and texture the image is matched with the previous images stored in the database and as a result the same images are provided to the user.

In [15] authors proposed a method which suggest the interesting elements to the developer by taking his/her interested code elements. The method statically finds out the elements of the source code and the elements are shown to developer in a tree structure. From the elements the developer selects the element of his/her interest and the algorithm finds the fuzzy set of related elements of ~~the~~ interest. The results show that the method helps in finding the elements of interest of the code to developer which are needed to be investigated.

In [16] authors used the IR method to recommend the items to the user. They have found the last ‘w’ number of visited pages from the user profile history. From the founded pages the top ‘k’ terms are retrieved using the content based filtering (CBF). And from these terms the ‘k’ queries are defined and against these queries the relevant links are derived from the web. From the selected ‘k’ terms the association rules are found using collaborative filtering (CF). The combined results are shown to the user. The study is conducted on a university website and the results show that the performance of the approach is good.

In [17] authors used the IR method for fault localization at character level instead of word level. They used the n-gram based model which matched the terms of source code with the bug report terms. It matched the combined terms of source code which are written in camel case with the bug report terms. It also provided the advantage of stemming as it matches at character level. The results show that it produces effective mean average precision (MAP).

In [18] authors proposed a method named “BugLocator”. It firstly locates the previous similar bugs of a new bug from the history using revised vector space model (rVSM). The files are extracted which were modified for fixing the previous bugs. On the basis of the similarity score the old modified files are suggested to the developer. Th results show that the method improves the old VSM, LDA, LSI and SUM methods.

In [19] authors proposed a method "BLUiR" (Bug Localization Using Information Retrieval). The method finds out the elements from the source code i.e. class name, method name, variable name. It then tokenizes the identifiers and stores that information in the XML format. Later it uses indexing for linking the tokens of XML file. From the file the abstract syntax tree (AST) is found through Eclipse Java development tool (JDT). The experiments are conducted with 3,400 bugs and the results show that it performs better than the state of art "BugLocator".

In [20] authors proposed a method named "AmaLgam". It analyzed the version history which is used in google. From the history past 'K' days commits are identified containing the word "fix" or "bug". From these commits the changed files are retrieved. From the history the similar bugs are identified using the "BugLocator" method previously identified, and the files are retrieved which are modified in order to fix the old previous bugs. It tokenizes the source code into class name, method name and variable name using the "BLUiR" method previously proposed. From all the three components the results are combined and suggested to the developer. Experiments are conducted on four open source projects and the results show that it improves 24% as compared to "BugLocator" and 16% as compared to "BLUiR" using mean average precision (MAP) value.

In the above discussed papers most of the papers are based on lexical matching as conventional IR is based on lexical matching while some have considered using the semantic methods. In lexical matching the problem of vocabulary mismatch occurs as the vocabulary of any programming language is limited. Also, due to a lot of repetition in the source code the precision value decreases and as a result the accuracy of IR methods is very low. Some are semantic based i.e. LDA, LSI but they are computationally costly.

In [21] the authors proposed an information theoretic based IR approach for fault localization which is based on relatedness. Those methods are used namely Pointwise Mutual Information (PMI) [22] and Normalized Google Distance (NGD) [23] which keep in consideration the context. The results show that the approach outperforms the literature lexical matching based approaches Vector Space Model (VSM), Jensen-Shannon Model (JSM), and the semantic based method Latent Semantic Indexing (LSI).

III. Proposed Approach

In this paper we have proposed an IR based technique which uses word embedding model for localizing faults. Figure 1 shows our proposed approach. Firstly, we have trained our word embedding model by historical bug data. After that on arrival of a new bug, we have used our trained word embedding model in two dimensions i.e. first, it gives us all the similar terms against the bug terms and second it gives us all the relevant terms to the bug terms. When dealing with the similar terms the word embedding model will provide similar terms and now using these terms we will make combinations of the terms with the bug terms and suggest files from the source code that it deems suspicious. On the contrary when dealing with relevant terms the word embedding model will provide relevant terms to the bug terms and these relevant terms will be searched individually from the source code and files will be suggested.

In the next subsections we will explain our proposed approach in detail.

A. Training Section

1) Mining of Repositories

The training data for the word embedding model is mined from the bug repositories. This data is served as an input to the model after it is preprocessed.

2) Preprocessing

The preprocessing step involves tokenization, removal of stop word and lemmatization.

- Tokenization:

  In this step the bug data gets split into terms which are called *tokens*. The bug title and bug description terms are split on the bases of space between them and later compared with the source code terms.

- Removal of Stop Words:

  Stop words are frequently used words that are usually ignored when natural language data is preprocessed. Commonly used stop words are *the*, *and*, *it*, etc.

  Therefore, after splitting the terms, the remaining words are checked for stop words and those terms are removed.

- Lemmatization:

  Lemmatization shortens a word to a common prefix while keeping in mind the grammatical context.

  Here the words from the previous step are reduced to their respective lemmas.

After this the bug data is said to have been preprocessed and ready for training the word embedding model.

3) Word Embedding Model

The preprocessed data form the bug repositories from the previous step serves as an input for the word embedding model. The word embedding model assigns different weightages to the terms of the bug repository data. It keeps in consideration the similarity and relatedness factor when assigning weightage. The terms having higher similarity would have a greater weightage as compared to terms that do not have high similarity. For relatedness the terms which are more likely to be related to each other would have a higher weightage.

B. Testing Section

1) Arrival of bug

When a user interacts with a system it is bound to encounter a bug at some point. On encountering a bug, they report it by making a bug report. A bug report is a detailed description of the users account of the undesirable scenario that they have witnessed.

2) Preprocessing

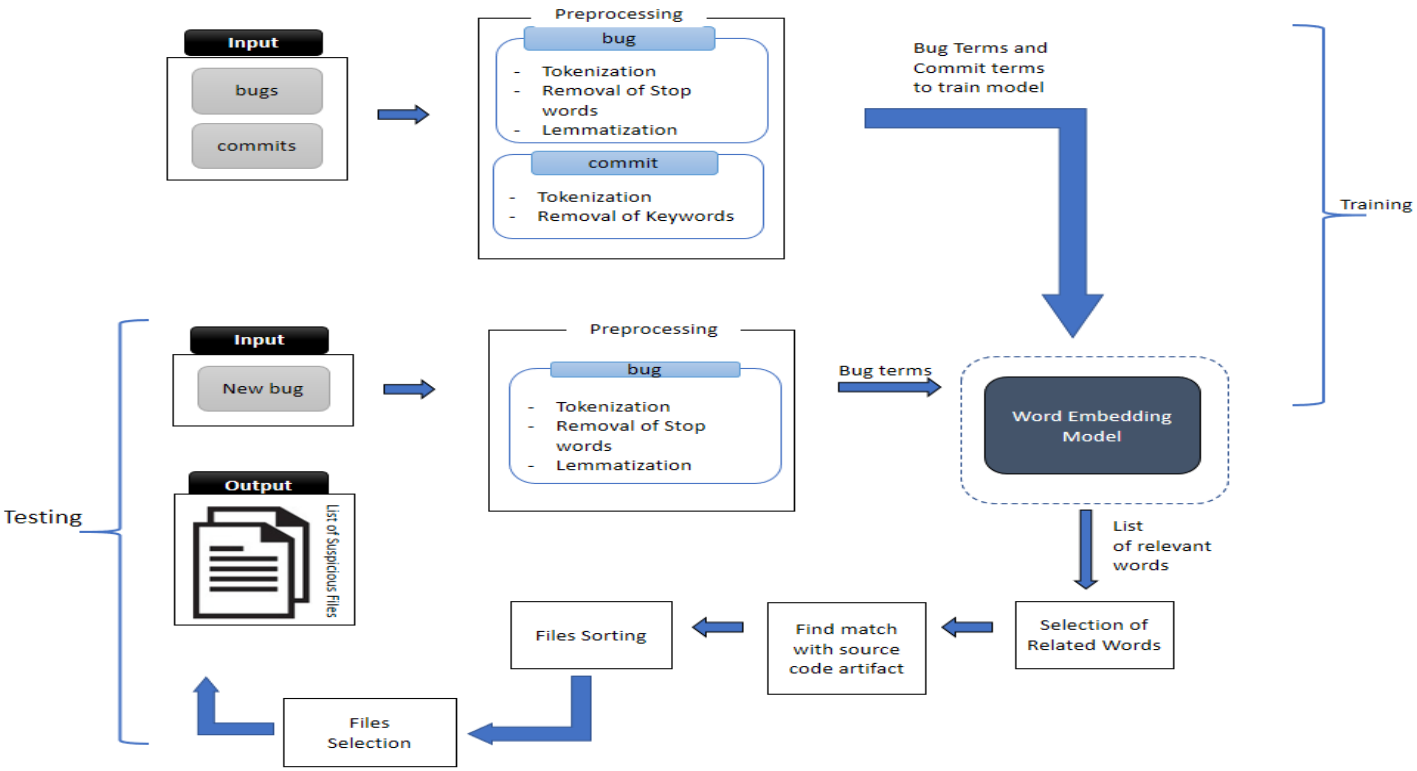

Figure 1 A Fault Localization Process Using Word Embedding Model

After a bug report has arrived, its contents are preprocessed by performing tokenization, removal of stop words and lemmatization. We have discussed these steps in detail earlier in Section III. Here also the same process would be done for the newly arrived bug.

3) Word Embedding Model

We have used a trained word embedding model and have already explained in detail in the previous subheading. In this step the preprocessed bug data from the previous step becomes an input for the trained word embedding model. The model works on the preprocessed data and produces a list of words based on comparison between the bug data and the model's trained data set. For our approach we have obtained two different sets of words from the model

a) Similar Context terms

The first list of words is made up of similar terms to the bug terms. For example, if the bug term is "memory" then the word embedding model come up with terms which are similar to memory like storage and RAM.

b) Relatedness Terms

The second list is made up of words which are related to the input bug terms. For example, here for the same term "memory" the word embedding model will come up with related terms like overflow and underflow.

4) Selection of terms

From the words produced by the model, the top ten words are considered for comparison with the source code terms as researchers have declared the top ten words are the most relevant words.[1]

5) Matching with source code

In case of similar words list, all the combinations of the similar bug terms are made and searched in the source code. In contrast when considering the related bug terms list, the terms of same topic are searched in the source code.

6) Sorting of Files

The files are sorted based on the amount of occurrence of each term produced by the word embedding model.

7) Selection and Suggestion of Files

Finally, from the sorted files the top three files are the ones that are supposed to contain the bug. These files are declared as suspicious files.

IV. Experiments, Results and Evaluation

The experiments are conducted to validate our proposed approach. We have performed experiments on "MoreBugs" [24] dataset[2] which is a benchmark data set and is widely used in research of fault localization. The results are shown in detail in the next subsections.

A. Performance of Proposed Approach

It is of utmost importance that our proposed approach detects the bugs correctly. The number of correctly identified bugs would directly reflect on its performance. For this purpose, we have calculated the number of correctly identified bugs by our proposed approach through the evaluation metrics of recall. Results indicate that our proposed approach has more recall value.

As we have explained in Section III we have used our word embedding model to obtain two different set of lists i.e. similar terms and related terms. Here while observing performance we have taken both those lists and calculated

[1] https://pydata.org/code-of-conduct.html

[2] https://engineering.purdue.edu/RVL/Database/moreBugs/

their respective recall. Table I and Table II show our performance of proposed approach using both the different word lists.

Table I. Performance of Proposed Approach Considering Similar Terms

System	Average Recall
AspectJ	52%

Table II. Performance of Proposed Approach Considering Related Terms

System	Average Recall
AspectJ	76%

B. Research Questions

RQ1. Does our approach improve the performance of bug localization?

In order to calculate the improvement made by our proposed approach we have made comparison with the previously proposed approaches in the literature. As we have discussed in the Related Work section of our paper that the IR based approaches are lexical based and some considered the semantics. So we have made comparison with the Vector Space Model (VSM), Latent Semantic Indexing (LSI) and also the methods Pointwise mutual information (PMI), Normalized Google Distance (NGD) which are proposed in the 2018 paper from which we get motivation. The results are shown in Table III.

Table III. Comparison of Proposed Approach with Other Approaches

Approaches	Average Recall
Proposed Approach (with Similarity Terms)	52%
VSM	22.5%
LSI	21.8%
PMI	45%
NGD	41.5%

RQ2. How many terms appear in a bug report on average and what is the impact of bug report length?

In Table IV we have shown the average number of terms that a bug report has. To confirm the effect of bug report length we have calculated the results of our proposed approach by considering bug title information and on the other hand we have also calculated our results by considering bug title and bug description information. Table V gives an insight to our findings related to this. We can conclude that if the length of bug report is greater than it gives better recall but there will be a slight decline in precision.

Table IV. Average Bug Report Length

System	Average length of bug report
AspectJ	10

Table V. Impact of Bug Report Length

System	Proposed Approach with Bug term	Proposed Approach with Bug Term and Bug Description
	Average Recall	**Average Recall**
AspectJ	76%	77.6%

RQ3. In which cases the proposed approach outperforms others?

Our proposed approach would help in regards when the vocabulary mismatch problem would occur. Using the similar words that the word embedding model produces, we can reach the part of the code that is more likely to contain the bug. In addition, when related words are produced by the word embedding model against the new arrival bug term, they can be used to obtain all the terms from history bugs related to it and help us in finding parts of source code that might contain them. This widens our search results and helps us in narrowing down the bug using the concept of relatedness.

C. Why our Technique Works

Whilst performing fault localization using bug reports, the user sometimes uses terms which are not directly used in the source code. Therefore, fault localization becomes difficult. Through our proposed approach we have found out the related and similar bug terms from historical bug repository's data which helps us in narrowing down the fault. Using similar and related terms to the ones in the bug reports, we are able to identify the actual location of bug in the source code.

V. Conclusions and Future Work

Providing maintenance by localizing faults is an important phase of software development. With the increase in the complexity and size of software, there is need for performing fault localization through efficient ways. In the past many different techniques have been proposed in literature. Most of these techniques are lexical and therefore result in the vocabulary mismatch problem. Others are semantic based and therefore are computationally expensive.

In this paper we have proposed a word embedding model which is trained on historical bug data and provides similar and related terms suggestions for newly arrived bug terms. We have validated our approach by using the MoreBugs dataset and implemented it on AspectJ system. We have also compared our proposed approach with existing techniques and discovered that our proposed approach gives better

recall. In the future we will also incorporate software change repositories with our proposed technique.

REFERENCES

[1] P. Ammann and J. Offutt, "Introduction to software testing" Cambridge University Press, 2016.

[2] Shao, P., Smith, R. K., 2009. Feature location by ir modules and call graph. In: Proceedings of the 47th Annual Southeast Regional Conference. ACM, p. 70.

[3] Lukins, S. K., Kraft, N. A., Etzkorn, L. H., 2010. Bug localization using latent dirichlet allocation. Information and Software Technology 52 (9), 972-990.

[4] Khatiwada, S., Tushev, M. and Mahmoud, A., 2018. Just enough semantics: an information theoretic approach for ir-based software bug localization. Information and Software Technology, 93 pp. 45-57.

[5] Wong, W. E., & Debroy, V. (2009). A survey of software fault localization (Vol. 9; Tech. Rep.).

[6] Renieres, Manos, and Steven P. Reiss. "Fault localization with nearest neighbor queries." Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on. IEEE, 2003.

[7] Abreu, Rui, Peter Zoeteweij, and Arjan JC Van Gemund. "On the accuracy of spectrum-based fault localization." Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007). IEEE, 2007.

[8] Abreu, Rui, Peter Zoeteweij, and Arjan JC Van Gemund. "Spectrum-based multiple fault localization." Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, 2009.

[9] Papadakis, Mike, and Yves Le Traon. "Using mutants to locate" unknown" faults." Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. IEEE, 2012.

[10] Le, Tien-Duy B., Ferdian Thung, and David Lo. "Theory and practice, do they match? a case with spectrum-based fault localization." 2013 IEEE International Conference on Software Maintenance. IEEE, 2013.

[11] Ju, Xiaolin, et al. "HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices." Journal of Systems and Software 90 (2014): 3-17.

[12] Baah, George K., Andy Podgurski, and Mary Jean Harrold. "Causal inference for statistical fault localization." Proceedings of the 19th international symposium on Software testing and analysis. ACM, 2010.

[13] Antoniol, Giuliano, et al. "Recovering traceability links between code and documentation." IEEE transactions on software engineering 28.10 (2002): 970-983.

[14] Müller, Henning, et al. "A review of content-based image retrieval systems in medical applications—clinical benefits and future directions." International journal of medical informatics 73.1 (2004): 1-23.

[15] Robillard, Martin P. "Automatic generation of suggestions for program investigation." ACM SIGSOFT Software Engineering Notes. Vol. 30. No. 5. ACM, 2005.

[16] Khribi, Mohamed Koutheaïr, Mohamed Jemni, and Olfa Nasraoui. "Automatic recommendations for e-learning personalization based on web usage mining techniques and information retrieval." Advanced Learning Technologies, 2008. ICALT'08. Eighth IEEE International Conference on. IEEE, 2008.

[17] Lal, Sangeeta, and Ashish Sureka. "A static technique for fault localization using character n-gram based information retrieval model." Proceedings of the 5th India Software Engineering Conference. ACM, 2012.

[18] Zhou, Jian, Hongyu Zhang, and David Lo. "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports." Software Engineering (ICSE), 2012 34th International Conference on. IEEE, 2012.

[19] Saha, Ripon K., et al. "Improving bug localization using structured information retrieval." Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. IEEE, 2013.

[20] Wang, Shaowei, and David Lo. "Version history, similar report, and structure: Putting them together for improved bug localization." Proceedings of the 22nd International Conference on Program Comprehension. ACM, 2014.

[21] Khatiwada, Saket, Miroslav Tushev, and Anas Mahmoud. "Just enough semantics: an information theoretic approach for ir-based software bug localization." Information and Software Technology 93 (2018): 45-57.

[22] Church, Kenneth Ward, and Patrick Hanks. "Word association norms, mutual information, and lexicography." Computational linguistics 16.1 (1990): 22-29.

[23] Cilibrasi, Rudi L., and Paul MB Vitanyi. "The google similarity distance." IEEE Transactions on knowledge and data engineering 19.3 (2007).

[24] Rao, S., & Kak, A. (2013). morebugs: A new dataset for benchmarking algorithms for information retrieval from software repositories (Tech. Rep.). ECE Technical Reports.

[25] Wong, Chu-Pan, et al. "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis." 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, 2014.