Comparison of Two Different Studies in Automating Software Bug Localization

Proposed to Cigdem Gunduz Demir
Emre Dogan, Hamdi Alperen Cetin
Department of Computer Engineering
Bilkent University

***Abstract*—In this technical report, we briefly show the implementation steps of the paper *Bug Localization with Combination of Deep Learning and Information Retrieval* [1] and compare its results with a previous study of this paper, *Where Should the Bugs Be Fixed? More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports* [2].**

Bug localization is defined as the task of locating potentially buggy source files in a software project given a bug report. It is an important and helpful tool for software developers so that they can easily focus on the related source files for the corresponding bug report. Recent studies deal with the automation of this process by using different tools. In [2], the authors proposed an advanced information retrieval (IR) technique to help this process to be automated. The results of this study showed that IR can be a powerful tool to solve the lexical similarity problems between source code and bug reports. Then, with the recent developments in deep learning area, it is noticed that deep neural networks can be complementary to the IR approach on bug localization. The study we investigated and implemented achieves different features by using information retrieval and other text processing approaches. Then by using a deep neural network architecture, all these features are combined to obtain a relevancy score which results with the most probable source code file causing the corresponding bug report.
After experiments, we find the top-k accuracy value for k=20 78.99% on test data. The original accuracy value from the paper[1] is approximately 80%.

***Index Terms*—Deep learning, bug localization, information retrieval.**

I. Introduction

During a software project, one of the most time & energy consuming process is debugging and fixing software bugs. Each found bug is reported by the textual documents called 'bug reports' to be fixed by developers. These reports include some important details regarding to the bug such as bug description and bug report time. After a bug report is created by an end user or a test engineer, it is assigned to a member of the developer team. At this point, all the related work of localizing and fixing the bug belongs to the developer.

For a developer trying to fix a bug in a large scaled software project, the most painful part of this process is to find the related source code files causing the corresponding bug [3]. Because most of the time, the developer needs to track a large number of source code files so that he/she can find the file(s) reasoning to the related software defects. In general, this process is named as bug localization.

As it is too inefficient to try to localize bugs manually, there have been several studies on automating this process. These studies help developers such that they can direct their attention on fixing the bugs in localized source code files.

There are different approaches in the studies of automating bug localization. Up to now, some advanced information retrieval techniques [4], [5], [6] and different machine learning approaches [7], [2] have been employed to establish a relation between bug reports and source code files. In this report, we give the details of implementation of two different studies regarding the automating bug localization problem.

In Section 2, the related background and previous studies are discussed. In Section 3, the details of the dataset is illustrated. The implemented model is explained clearly in Section 4. In section 5, the evaluation of experiments is done and lastly, our implementation study is concluded briefly.

II. Background

In this report, two recent studies[1], [2] on the automation of bug localization have been investigated. In the study of Zhou J. et al., an information retrieval based tool, BugTool, for locating the relevant files for fixing a bug is proposed. BugLocator ranks all files based on the textual similarity between the initial bug report and the source code using a revised Vector Space Model (rVSM), taking into consideration information about similar bugs that have been fixed before. This study showed a performance that is much better than any other study done up to 2012. In the dataset, they used relevant buggy files for 62.60% of Eclipse project whose bugs are ranked in the top ten among 12,863 files. It is noticeable that getting such an accuracy among 12,863 files was a remarkable result at that time.

Five years later than this study, Lam A. et al. proposed a new approach for bug localization by using rVSM method as well as some other textual features that have not been used before[1]. rVSM collects the feature on the textual similarity between bug reports and source files. DNN is used to learn to relate the terms in bug reports to potentially different code tokens and terms in source files. It is observed that DNN and rVSM complement well to each other to achieve higher bug localization accuracy than individual models. This model is trained and tested with the same dataset and relevant buggy source files are observed in the top-10 results successfully in more than 80% of all the cases.

TABLE I
A ROW FROM THE DATASET

id	bug_id	summary	description	report_time	report_timestamp	status	commit	commit_time	files
1	384108	Bug 384108 JUnit view icon no longer shows progress while executing tests	Build Identifier: Version: Juno Release Build id: 2012061 -1722 Before I upgraded to Juno this ...	2012-07-03 03:39:25	1341300000	resolved fixed	5da5952	1389970000	bundles/ org.eclipse.e4.ui. workbench/ src/org/eclipse/e4/ui /internal/workbench/ PartServiceImpl.java

III. DATASET

Even though there are many bug localization datasets available online, both papers [1], [2] provide the dataset[1] that they used for experiments. To make comparisons with results of our implementation and the original values more explicitly, we use the same dataset. It includes bug histories for 6 different project which are AspectJ, Birt, Eclipse Platform UI, JDT, SWT and Tomcat. We use only Eclipse Platform UI dataset in our implementation. It includes 6495 bug reports with their bug id, summary, description, report time, report time-stamp, status, commit, commit time and files features. Table 1 shows a sample from the dataset. Commit, commit time and files can be accessed from the GitHub repository of the project. All the added, modified and deleted files are specified in the files section. We need java source files of the project as the version before the related commit so that we can extract similarity features between bug reports and files. Totally, we have 8735 buggy files and corresponding bug reports, but we did not have the features extracted from these files and reports. All the feature extraction part is completed by us. More details of feature definitions and extraction can be found in Section IV.A and IV.B.

IV. MODEL DESCRIPTIONS

In the study[1] of Lam et al., they propose a bug localization approach named DNNLOC which can be seen in Figure 1. Its DNN-based feature combinator consists of an input layer, a hidden layer and an output layer. The inputs of the combinator are DNN relevancy, text similarity, collaborative filtering score, class name similarity, bug fixing recency and bug fixing frequency features which are extracted from the bug history and the files committed while fixing the regarding bug. The number of hidden nodes are changing during experiments. The output of the model is a score of fault proneness of the source code file.

A. Feature Definitions

1) Text Similarity(rVSM): For the bug reports, we split into the words using whitespaces. The punctuation and standard stopwords are removed. The words are converted to their stems using the standard Porter stemming program. Then, the name of an entity is split into individual words using CamelCase or Hungarian notations. We also keep the original name. Finally, the term significance weights of all the words are computed using Term Frequency - Inverse Document Frequency (tf-idf) [21]. The weights of the terms in a bug report are used as its features and fed into the projection module.

From the source files, we extract terms with the same manner in bug reports. While applying this manner, most of the source code is deleted. Rest of the textual data is mostly the names of identifiers in a source file and the names of API classes and interfaces that are used in the source file.

We consider the textual similarity between a bug report and a corresponding source file as a feature. To extract such feature, we adopted the revised Vector Space Model (rVSM) from Zhou et al.[8]. The textual similarity of a bug report B and a file f computed by rVSM is used as the score of this feature for the pair (B, f) . For such similarity score, we merge the vocabulary from the reports with that of the source files.

The rVSM method is an IR approach based on the cosine similarity which is shown in Equation (1).

$$Similarity = cos(q, d) = \frac{V_q * V_d}{(\|V_q\|) * (\|V_d\|)} \quad (1)$$

Equation (2) is a logistic function that ensures that larger documents are given higher scores during ranking. We use Equation (2) to compute the length value for each source file according to the number of terms the file contains.

$$g(\#terms) = \frac{1}{1 + e^{-N.(\#terms)}} \quad (2)$$

Combining the above analysis, we thus propose a new scoring algorithm for rVSM as follows:

$$rVSMScore(q, d) = g(\#terms) \times cos(q, d) \quad (3)$$

Given a bug report, we use Equation (3) to determine the relevance scores (rVSMScore) between each source code file and the bug report.

2) Collaborative Filtering Score: This aims to measure the similarity of a bug report and previously fixed bug reports by the same file. It is defined as follows. Given a bug report B and a source code file f, the score is the textual similarity (measured by rVSM) between B and all the combined texts in the bug reports that were fixed by f and before B was reported.

3) Class Name Similarity: We also consider the textual similarity between the names of classes mentioned in a bug report and those in a source file. Cosine similarity is used for this similarity calculation.

4) Bug fixing Recency: We compute the bug-fixing recency score as follows. Let us call S the set of bug reports that were filed before bug report B and were fixed in file f. Among S , let B' be the report that was most recently fixed. The bug-fixing

[1]http://dx.doi.org/10.6084/m9.figshare.951967

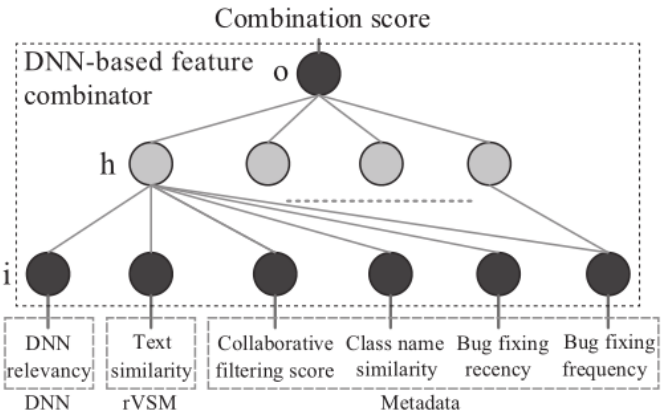

Fig. 1. DNNLOC Bug Localization[1]

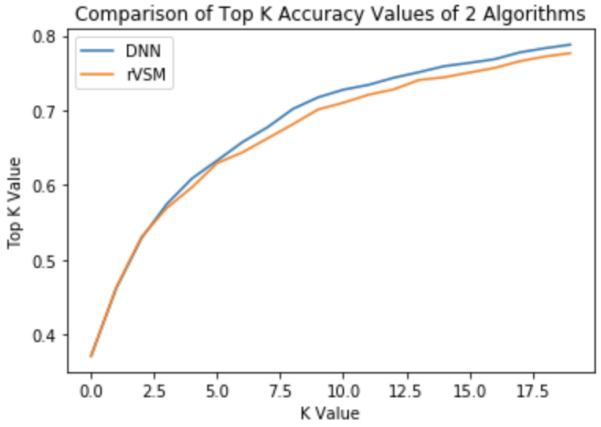

Fig. 2. Comparison of Our 2 Implemented Model

recency for a pair of a bug report B and a file f is defined as $(B.month - B'.month + 1)^{-1}$. If f was fixed for a report B' in the same month that B was filed, the value is 1. The further in the past f was last fixed, the smaller the bug-fixing recency score.

5) Bug fixing Frequency: We also extract the bug-fixing frequency for a source file f. We count the number of times that f was fixed before the bug report B was filed, and use that number as a feature.

6) DNN Relevancy: In the study, the authors use another DNN model to tolerate the lexical mismatch between bug report and source file. Briefly, this DNN gets tf-idf scores of all the terms that we generated for the first feature which is Text Similarity(rVSM) as input and gives a relevancy score between 0 and 1 (regression model).

B. Feature Extraction

Because the features for each source code file and bug report pair are not available in the dataset, we extracted features for these pairs. We used Python NLTK library[2] for text processing explained in Feature Definitions. Also, we couldn't be able to extract all features since calculating DNN relevancy for all samples takes a lot of time.

First of all, we crawled all java files which are identified as buggy in the bug reports and calculated all features for each pair. For example, if 4 different java source files are changed to fix the bug, we produce 4 different samples from this information and label each sample as buggy, that means the same bug report become pair with each of these java files and all 5 features are calculated for each pair. To train our model, we need bug report and source file pairs which has no relation while fixing the bug reported at the bug report. We iterated over all bug reports and paired them with 50 irrelevant source files for each buggy java files which belongs to the bug report. That means, if a bug report has 3 different buggy java source files, we calculated features for 150 different irrelevant pairs and labeled them as clean. That increased the number of samples, but it makes the data unbalanced. The ratio of buggy labels and clean labels is 1 to 50. We would like to emphasize that we randomly chose other source files in the data set to find irrelevant pairs until we reached enough number of files which are different from the buggy files. For a better representation of the real life, we need the other files which are available when the bug is reported. To find these files we tried to use git in the our code script, but we got some inconsistency between the files from dataset and the files the git checkouts. After that, we decided to stick with just that dataset.

At the end, we had 445485 bug report and source code file pairs with 5 different features. The number of different samples labeled buggy is 8735, and the number of different samples labeled clean is 436750. we oversampled the data labeled as buggy. After oversampling, the ratio between buggy samples and clean samples became 1 to 5. At the end, we achieved a dataset with length 524100.

[2]https://www.nltk.org

V. Experiments

The DNN model that we used for experiments has 3 layers. The number of input nodes are 5 for the features text similarity, collaborative filtering score, class name similarity, bug fixing recency and bug fixing frequency. The number of output nodes is 1 to represent relevancy score between the bug report and the source file. First, we split all dataset into train and test data in proportion to 80% and 20% respectively. For training, the alpha value was 0.00005 and the stop criteria was 10000 iterations or 30 consecutive steps without loss change, whichever happens first. After that, we made experiments to find the proper hidden node count which gives high top-k accuracy and has reasonable running times. For the number of hidden nodes 100, 200, ..., 1000, accuracy values didn't change significantly. Maybe, even 100 was enough for high accuracy but while using 300, the running time of training was at reasonable levels and we wanted to use number of hidden units as high as possible. Starting 400, it was not fast enough to test our algorithm again and again. Then, we decided to choose 300 for the number of hidden nodes. After choosing it, we made 10 fold experiments on train dataset. The accuracies of our implementations in Figure 2, are the average accuracies of 10-fold experiments.

TABLE II
COMPARISON OF OUR IMPLEMENTATIONS WITH THE RESULTS FROM PAPERS

k Value	Top-K rVSM Results of the Paper	Top-K rVSM Results of Our Implementation	Top-K DNN Results of the Paper	Top-K DNN Results of Our Implementation
1	26.5%	24.4%	45.8%	37.5%
5	49.3%	51.7%	70.5%	61.5%
10	60.1%	58.9%	78.1%	72.3%

VI. EVALUATION

While calculating top-k score, a sample is accepted as predicted correctly if the one of the first k predicted results is in the label of the test sample. This sounds like a light-weighted definition of the accuracy, but, in both studies that we implemented, top-k score is used as the main evaluation method. So, we calculated and compared accuracies in terms of top-k score. We implemented two seperate models, DNN Based Model and rVSM Based Model. The top-k accuracy comparison of these algorithms can be observed in Figure 2.

While calculating the success of models that we implemented, the results on both papers are baseline for us. Top-k accuracies of our implementations give pretty similar results to the ones from papers.

Equation 4 shows the real error rate with %99 confidence level(zn = 2.58). The interval is nearly zero, because the number of samples that we used is high enough to make error interval nearly zero.

$$error(M) = 0.2101 \pm 2.58 * \sqrt{\frac{0.2101 * 0.7899}{524100}} \approx 02101 \quad (4)$$

Also, we performed a paired t-test between the results of our two algorithm implementations with the the top-k accuracies between 1 and 20. The test finds p value as 2.8037e-7 which means the average results of two algorithms are statistically different with the confidence level 95%. So, the DNNLOC is statistically better than rVSM approach. This is natural, because DNNLOC uses metadata features beside rVSM.

VII. CONCLUSION

In this study, we implemented two different studies on bug localization based on information retrieval and deep learning approaches. In the previous paper[2], the only used feature to localize bugs is the rVSM score which is an improved cosine similarity metric. In the second paper we implemented[1], this rVSM feature and other textual features are combined by a deep neural network to be able to increase the accuracy.

It is noticeable that deep learning and information retrieval approaches work well on this area. In our implementation, also the top-k accuracy value increased in a significant way. Table 2 shows the top-k accuracies from both papers and our implementations. It is clear that top-k results of both models are quite similar to the original values. Still, there is a little difference which comes from the lack of DNN Relevancy Score in our implementation.

Consequently, this study illustrates the implementation details of two studies on bug localization. In the first study, an IR appraoch ,rVSM, is used alone while in the second study rVSM and other textual features are fed to neural network which leads to a greater top-k accuracy success.

REFERENCES

[1] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*. IEEE, 2017, pp. 218–229.

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

[3] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design of bug fixes," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 332–341.

[4] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[5] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011, pp. 43–52.

[6] 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.

[7] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 263–272.

[8] 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 *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 14–24.