Comparing different Information Retrieval based Fault Localization Techniques

1st Michael Jeremy Olea
Department of Electrical and Software Engineering
University of Calgary
Calgary, Canada
michaeljeremy.olea@ucalgary.ca

***Abstract*—Locating bugs is an important in software development and increases in difficulty as the size of the project increases. Many methods of locating bugs already exist, and we want to implement and compare several methods to see which method performs the best. The first method we will implement is a simple Information Retrieval Fault Localization approach using a revised Vector Space Model, comparing the textual similarities between source code files and bug reports. The second method will be the proposed BugLocator [1] using a combination of the first method and an indirect comparison method that compares new bug reports with existing bug reports. Finally the third method we will be implementing is the proposed BLUiR [2] method that uses structured information retrieval based on code constructs, such as class, and method, names. We found that BugLocator proved to be the best of the three algorithms we tested.**

***Index Terms*—Fault Localization, Information Retrieval**

I. Introduction

When developing large-scale software products, bugs in the code are inevitable. The scale and severity of the bugs vary, but ideally, developers would like to decrease the amount of bugs in the code as possible to create the best product possible. To do this, developers usually have a process to take user feedback on bugs and create bug report tickets to store in a large backlog to be fixed at a later time. These bug reports usually contain a short description on what the bug is doing and how to reproduce the bug, along with a stack trace of the console output of the error. Developers take these bug reports try to identify where in the code-base it is occurring using their own knowledge of the architecture of the code-base. This process of identifying where bugs are located given a bug report can range from very easy to very hard. We want to see if we can use any existing fault localization techniques to help us locate bugs faster and easier. We also want to compare these different methods to see which ones are the best.

Many fault localization techniques have already been proposed, but in this paper we will focus on Information Retrieval based Fault Localization methods. The first fault localization method we will explore is a simplified version of the proposed BugLocator [1], where we will only be comparing textual similarities between bug reports and source code files. This method will use their revised Vector Space Model to rank all source code files based on the textual similarities with the bug report. rVSM is an optimization of the classical VSM model for fault localization that takes into account factors such as TF-IDF giving lower rankings to larger files. The idea is bug reports may contain keywords either in the description or the stack trace that can be found in source code files.

The second method we will explore is the complete BugLocator, which is a combination of method 1 (direct comparison) along with an indirect comparison between new bug reports with fixed bug reports. The indirect comparison will rank each source code file based on textual similarities between the new bug report and old bug reports that occurred due to a bug in the source code file. The idea is that similar bugs tend to be fixed by similar files. After calculating the direct and indirect similarity scores, they will be combined together to create a final score.

After BugLocator was published, many other IRFL methods have been proposed to try and improve the results of BugLocator. One of those methods is BLUiR [2] proposed by R. K. Saha, which uses structured information retrieval to distinguish the source code's rich structure into code constructs such as class names, methods, variable names and comments. BugLocator treats the source code files as flat text which simplifies the system, but sacrifices the opportunity to use the structural information to improve the fault localization. Ripon Saha found that their proposed method of fault localization were equal or an improvement from BugLocator. We want to see if we can replicate the same results in our implementation.

II. Background

A. Simple revised Vector Space Model

The simple revised Vector Space Model was proposed by J. Zhou, H. Zhang, and D. Lo in a 2012 paper [1]. The paper mainly focused on the BugLocator, but they also discussed the experimental results of the revised Vector Space Model (rVSM). rVSM is a optimized version of the classical Vector Space Model (VSM). VSM is a model for representing text documents as vectors and finding the similarities between them [3]. However, for the use case of finding similarities between source code files and bug reports, VSM has a major flaw. Classical VSM favours small documents over longer documents during ranking because long documents are represented in a way that indicates that they have poor similarity values [1]. However, according to previous studies, larger source code files are statistically more probable to contain bugs [1].

B. BugLocator

The BugLocator was also proposed in the same paper as rVSM. However, the results of BugLocator were more greatly explored. BugLocator consists of multiple components, it is a weighted sum of a direct similarity and and indirect similarity. The direct similarity is the rVSM method mentioned previously and the indirect method is a textual comparison between the new bug report and all other bug reports that have been fixed. Comparison scores to source files are the sum of similarity scores between the new bug report and all other bug reports that were fixed by the given source file. Finally, after calculating both direct and indirect similarity scores, a weighted sum of the two using a weight factor was calculated to give the final score.

C. BLUiR

BLUiR was proposed in an attempt to improve the results from BugLocator [2]. The success of IR based fault localization is dependent on how well the program can correctly identify and match keywords between the source files and the bug reports. The way the source code files and bug reports are processed could significantly impact the accuracy of the program. BugLocator treats source code files as flat text files containing no information about the codes structure. The BLUiR method aims to keep the structural information of the source code files to improve the fault localization. With queries , bug reports are separated into two fields, the concise summary field and the more verbose description field. For the source code files, the files are parsed into four categories, classes, methods, variables, and comments. We the calculate the direct similarities between the bug reports and the source code files.

III. Method

For our implementation of BLUiR we will mostly follow the exact steps outlined by R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry [2]. However in their paper, they proposed a different TF-IDF formula that they believed was better than the one that BugLocator used. For our implementation of BLUiR, we will not be using their TF-IDF formula, but instead use the same one that BugLocator used. This section will outline the steps necessary for the structured IR-based approach for fault localization. The purpose of this is to better compare the two models by removing other variables that could affect the results.

A. BLUiR Architecture

Fig. 1 shows the overall architecture of BLUiR. BLUiR takes the source code files and builds a Abstract Syntax Tree (AST) for each node using the javalang library for python. We will traverse throught the AST and extract the different program constructs such as class, methods, variables, and comments. BLUiR then continues to tokenize each section by removing stop words, stemmed the words and removing punctuation. For the bug reports, since the data is already separated into summary and description, we would only need to tokenize the two sections.

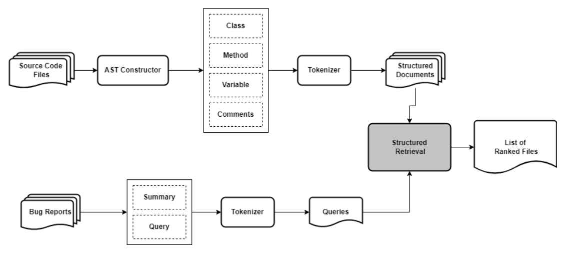

Fig. 1. BLUiR Architecture.

Once the bug reports and source code files are tokenized, we can begin to rank them by similarity.

B. Ranking Similarity with Structural Information

The BugLocator model presented earlier does not consider source code structure, so each term in a source code file is considered having the same relevance to the given query. Therefore, important information like class names and method names often get lost in the relatively large number of variable names and comments terms. So, if a source code file with class name ”A” also contains 10 variable names having the term ”A” within the camel casing, then the class name does not have a greater weight. Therefore, if there is a bug report related to a class ”A”, it will rank files that have the term ”A” more than 11 times higher regardless if those instances were located in only the comments or the variable names. BLUiR using different code constructs to overcome this problem. Given the four document fields (classes, methods, variables, comments) and two query representations (summary, description) we calculate similarities by calculating the cosine similarities of all eight combinations of document fields and query representations then summing them together to get a final score. The benefit is that terms appearing in multiple document fields are assigned a greater weight since the contribution of each field is summed together.

IV. Experiment

A. Objectives

In this experiment we will attempt to apply three different IRFL techniques, a revised vector space model, the BugLocator which is a combination of the revised vector space model and an indirect method of calculating similarity, and BLUiR which is a structured based IRFL technique. The dataset we will be using is Bench4BL. Bench4BL is openly accessible to the public, and to the best of our knowledge, is the most inclusive and up-to-date data set for fault localization. The main objectives of our experiment are:

1) Successfully implement the revised vector space model, BugLocator, and BLUiR.
2) Replicate the results that the original papers who proposed these solutions produced.
3) Compare the results of the three methods.

4) Determine which of the three methods perform the best for our dataset.

B. Procedure

1) Preprocessing for Method 1 and 2: Before calculating any textual similarities, we need to first clean the data to remove any useless clutter in the data. The unprocessed java source code files have many unnecessary symbols such as asterisks, brackets, and algebraic operations. We remove them by importing sklean's ENGLISH_STOP_WORDS and adding on basic java keywords and operations. We then want to remove all punctuation from the source code text files. Finally we turn all the words into lowercase and use PorterStemmer to stem the words. Stemming is the process of reducing a word down to it's base word.

For the bug reports, both the summary and description section will be preprocessed the same way the source code files were preprocessess, then after the preprocessing the summary and description fields will be combined to create a "query" field. This is what we will be using to calculate similarities with.

Finally before we are able to use the textual data to calculate similarities, we first need to vectorize the data using TF-IDF, which transforms the textual data into a vector of weights.

2) Method 1 - Revised Vector Space Model: To understand rVSM, we need to first understand VSM. In VSM, the similarity score between a document d and a query q is given by a cosine similarity.

$$s(d,q) = cos(d,q) = \frac{\overrightarrow{V_d} \cdot \overrightarrow{V_q}}{|\overrightarrow{V_d}||\overrightarrow{V_q}|} \quad (1)$$

rVSM is a optimization of classical VSM. A major flaw of VSM is that it favours smaller files over larger ones [1]. However, statistically it is more likely that larger files contain more bugs, so rVSM accounts for this. For rVSM, the similarity is calculated using the following equation:

$$rVSM(d,q) = g(\#terms) \times s(d,q) \quad (2)$$

$$g(\#terms) = \frac{1}{1+e^{-N(\#terms)}} \quad (3)$$

3) Method 2 - BugLocator: For BugLocator, we need to take into account bugs that have been fixed before, this is called the indirect similarity ranking. We first construct a three-layer homogeneous graph shown in Fig. 2. The first layer is the new bug report, the second layer is previously fixed bugs, and the third layer represents all source code files. The connections between the third layer and the second layer represent all the source files in the third layer that caused the bug in the second layer.

The indirect similarity score for each source code file is as follows:

$$SimiScore = \sum_{\substack{all\ bugs \\ resulting \\ from\ F_j}} (s(B,S_i)/n_i) \quad (4)$$

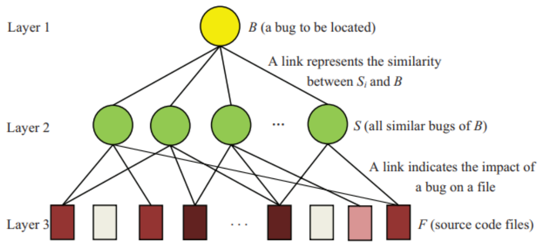
Layer 1
B (a bug to be located)
A link represents the similarity between S_i and B
Layer 2
...
S (all similar bugs of B)
A link indicates the impact of a bug on a file
Layer 3
...
F (source code files)

Fig. 2. Indirect Similarity Graph.

, where S_i is a bug that connects to F_j, and n_i is the total number of connections to layer 3 S_i has to layer 2.

Finally the direct and indirect similarity equations are normalized and combined using the equation:

$$FinalScore = (1-\alpha) \times N(rVSMScore) + \alpha \times N(SimiScore) \quad (5)$$

4) Preprocessing for Method 3: To preprocess the soruce code files in method 3, we need to build an Abstract Syntax Tree (AST) to break down the source code files into classes, methods, variable, and comments. An AST is a tree representation of code and is a fundamental way of how compilers work [4]. To build an AST, we are going to be using the python library "javalang", which is an AST library specifically for java source code files. To create the AST, we will call the function javalang.parse.parse which will return an AST object. After creating the AST, we just need to filter through the tree to get all the necessary components (classes, methods, variables, comments). After we separate the source code into the separate components, we run the same preprocessing steps from method 1 and 2. The preprocessing steps for the bug reports also stays the same, expect that we no longer combine the summary and description fields.

5) Method 3 - BLUiR: Similarly to BugLocator, we calculate similarity scores using (1). Except instead of doing it once, we perform separate consine similarity calculations on each of the eight (query representations, document fields) combinations then sum the similarity scores across all eight combinations.

$$s'(\vec{d},\vec{q}) = \sum_{r\in Q}\sum_{f\in D} s(d_f, q_r) \quad (6)$$

where r is a query representation and f is a document field.

C. Measurements

All methods in this project are measured using Mean Reciprocal Rank and Mean Average Precision.

1) Mean Reciprocal Rank: Mean Reciprocal Rank (MRR) is a statistic used for evaluating a process that produces a list of possible responses [1]. To calculate reciprocal rank, we take the reciprocal of the first correctly ranked file. Then mean reciprocal rank is the mean of all the reciprocal ranks. The mean reciprocal rank is calculated by:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{7}$$

The higher the MRR is, the better the fault localization performance.

2) Mean Average Precision: Mean average precision (MAP) is the mean of all average precision's. Average precision is defined by the equation:

$$AvgP_i = \sum_{i=1}^{M}\frac{P(j)\times pos(j)}{number\,of\,positive\,instances} \tag{8}$$

, where j is the rank, M is the number of instances retrieved, pos(j) indicates whether the result at the rank is relevant, P(j) is used to determine the precision at any given cutoff point, and is calculated by:

$$P(j) = \sum_{i=1}^{M}\frac{number\,of\,positive\,instances\,in\,top\,j}{j} \tag{9}$$

The higher the MAP is, the better the fault localization performance.

D. Research Questions

RQ1: How many bugs can be successfully located by BugLocator

RQ2: Which of the three methods will perform the best?

RQ3: Can we improve BugLocator by replacing the rVSM direct similarity score with BLUiR direct similarity score?

E. Results

1) RQ1: Table I shows a few of the projects that BLUiR was tested on and the results acheived by BLUiR.

TABLE I
BLUiR Performance

Project	Top 1	Top 5	Top 10	MRR	MAP
SPR	78 (21.8%)	182 (50.8%)	228 (60.7%)	0.35	0.27
IO	37 (54.4%)	61 (89.7%)	64 (94.1%)	0.67	0.60
LANG	79 (48.8%)	122 (75.3%)	136 (90.0%)	0.61	0.53
DATAMONGO	35 (15.0%)	111 (47.6%)	146 (62.7%)	0.30	0.23

For SPR, BLUiR successfully located 78 (21.8%) bugs in the top 1, 182 (50.8%) bugs in the top 5, and 228 (60.7%) bugs in the top 10. BLUiR also achieved a MRR of 0.35 and a MAP of 0.27. SPR was the second largest we tested dataset, so it makes sense that it found a lot of bugs, but the MRR and MAP performance scores were quite low. For IO, which is a significantly smaller dataset, BLUiR found 37 (54.4%) bugs in the top 1, 61 (89.7%) bugs in the top 5, 64 (94.1%) bugs in the top 10 and the MRR and MAP scores were 0.67 and 0.6 respectively. BLUiR performed much better on IO, especially for finding bugs within the top 10 ranked files. However, this could also be due to IO being a smaller dataset.

2) RQ2: We will be comparing the results from rVSM, BugLocator, and BLUiR (will be referred to as method 1, method 2, and method 3) to see which performed the best out of the three methods. But first, we need to determine which alpha value performed the best for BugLocator. Fig. 3 and Fig. 4 show the MRR and MAP of BugLocator for each project at different alpha value weights.

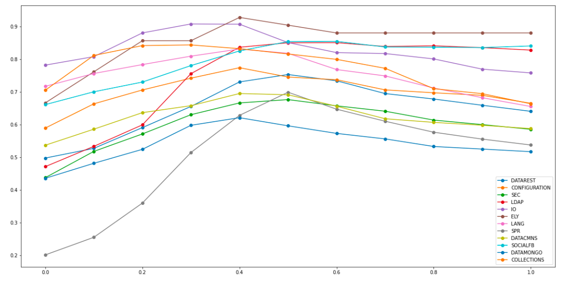

Fig. 3. BugLocator MRR at different alpha values.

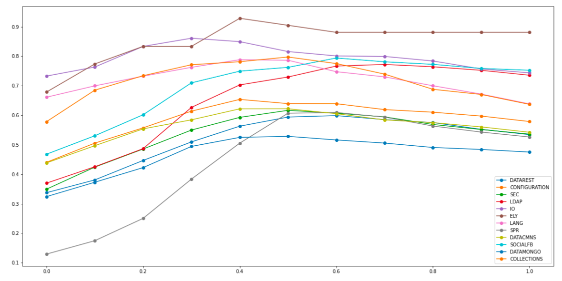

Fig. 4. BugLocator MRR at different alpha values.

Using this data, we can see that the best performing alpha value is either 0.4 or 0.5. For our comparison, we will use alpha=0.4 to compare BugLocator to the other methods. We will compare our three methods in MRR and MAP to see which will perform the best.

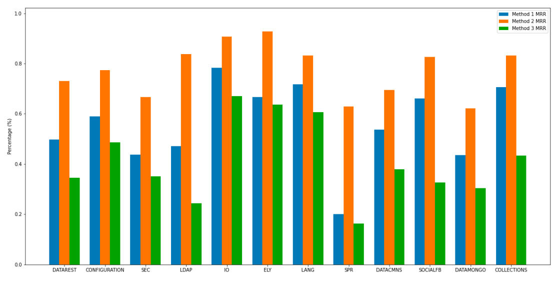

Fig. 5. MRR of all three methods.

Fig. 5, and Fig. 6, compares the three methods in terms of MRR and MAP. There seems to be a clear ranking among the three methods; method 2 clearly performs the best, then method 1, and method 3 is the worst of the three. For all

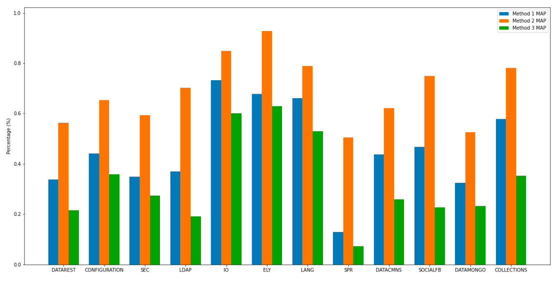

Fig. 6. MAP of all three methods.

projects, method 2 performs better than the rest on both MRR and MAP for every project.

3) RQ3: Since BLUiR compares bug reports directly to source code files, it would classify as a direct similarity. We want to find out if using BLUiR as a direct similarity for BugLocator (instead of using rVSM) can improve the performance. Theoretically, it is possible that rVSM and the indirect similarity method of BugLocator rank the files too similarly, meaning they don't provide each other with any new information. We want to test this out by replacing rVSM with BLUiR in BugLocator. We will call this BugLocator 2.0.

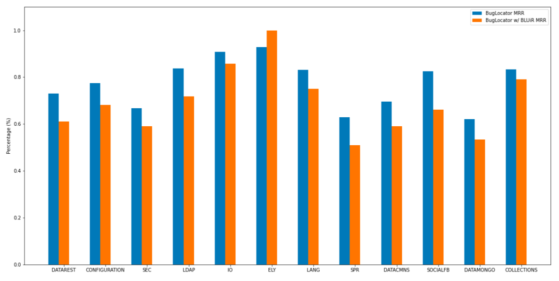

Fig. 7. Comparing MRR of BugLocator 2.0 against original.

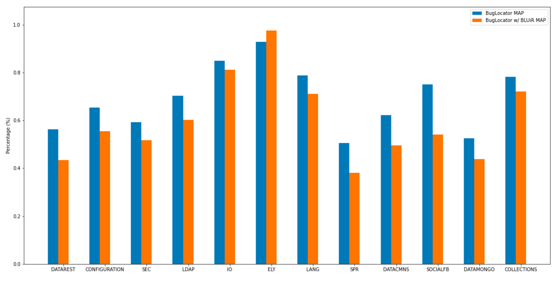

Fig. 8. Comparing MAP of BugLocator 2.0 against original.

In Fig. 7 and Fig. 8, we can see that the original BugLocator outperforms BugLocator 2.0 in both MRR and MAP for all projects except one. In ELY, BugLocator 2.0 actually performs better than the original BugLocator. This means our hypothesis that rVSM could rank files too similarly to BugLocator is wrong.

V. Discussion

Our results show that there is a clear hierarchy of which methods perform better than others. BugLocator performed the best and BLUiR performed the worst. This was the consistent for almost every project and every metric (there were only a few cases where a worse method slightly outperformed a better method in a single particular project).

We were able to successfully implement BLUiR but our results did not align with the results by R. K. Saha [2] who in their study, found that BLUiR performed better than BugLocator. This could be due to many things, for one, I made a few modifications to their procedure that could have resulted in the different outcome. In their study, they used a different TF-IDF formulation that I did. I chose to use the same TF-IDF formula that BugLocator used. They also used a summation of TF-IDF instead of cosine similarity to calculate their similarity scores. However, I think the main reason for the lower performance is the smaller data we got from the preprocessing using the AST. The AST separated the source code files into classes, methods, variables, and comments but left out the import fields. With the library we were using, it did not read anything outside of the class structure so we were not able to get the imports of the java file. I think this information is important because it tells the program which other classes and libraries will be used in the particular file.

We explored the idea of using BLUiR with BugLocator and found that it is possible for BLUiR to improve BugLocator. We hypothesized that rVSM may rank files too similarly to BugLocator's indirect similarity methods, so using a different direct similarity method could improve BugLocator. We found that BLUiR was able to improve BugLocator in a single project, meaning that it is possible for BLUiR to improve BugLocator using a weighted sum of the three methods (rVSM, BLUiR, and indirect method). This is something we can explore in the future, it may be possible to use BLUiR to make up for some weaknesses of BugLocator, but we would have to experiment with it to find out.

VI. Related Work

In a study by S. Rao and A. Kak [5], they discovered that IR based fault localization techniques are at least as effective as stativc and dynamic bug localization tools developed in the past. They also experimented on more sophisticated models such as LDA, LSA, and CBDM and found that these more sophisticated models do not outperform simpler models such as unigram and VSM for IR based fault localization on large software systems.

In a study by Lee et al [6], they looked at what the impact of version matching is on performance of IRFL techniques. The IRFL models they tested were BugLocator, BLUiR, BRTracer, AmaLgram, BLIA, and Locus. BugLocator and BLUiR are both models we used in our study, but the also used BRTracer, AmaLgram, BLIA, and Locus which were very interesting models. BRTracer is a model that leverages stack trace information to localize bugs, AmaLgram uses version history, BLIA is a combination of BLUiR and BRTracer, and Locus uses code change information to help narrow down the bug localization. These are all interesting models and their study found that using version history did improve the performance of fault localization.

In a study by S. Cheng, X. Yan, and A. Khan [7], they attempted to improve the accuracy of fault localization by solving the lexical mismatch between natural language in the bug reports and the programming language in the source files. They propose a similarity integration method which uses similarities calculated by information retrieval and word embedding. Word embedding is a technique that maps words in a vector space such that words that are more similar are closer to each other. To further improve the fault localization, they used a deep neural network to get correlations between bug reports and source code files. Their experimental results shows that their solution outperforms several existing bug localization approaches such as BugLocator.

In a study by S. Akbar and A. Kak [8], they attempt to divide the history of fault localization tools into three generations and compare each generation in terms of performance. The first generation methods they evaluated were TF-IDF and DLM, the second generation methods were BugLocator and BLUiR, and the third generation methods they tested are MRF.SD and MRF.FD. MRF.SD is a method that measures probability distribution of frequencies of pairs of consecutively occurring terms appearing in a file to determine a relevancy score. MRF.FD is similar but considers the frequencies of all pairs of terms. They discovered that the third generation methods outperform both second generation methods and the second generation methods performed equally to the first generation methods in large scale software systems.

In a study by Le et al [9], they proposed a mult-modal bug localization approach named Adaptive Multi-modal bug Localization (AML). This is different that other approaches that are one-size-fits-all. Their proposed approach can adapt itself to better localize each bug report by tuning it's weights learned from a training set. They found that AML performed better than the best baseline models by 48%, 31%, and 28% when comparing top 1, top 5, and top 10 rankings.

In a study by F. Song and W. Croft, they tested a new method of performing information retrieval using a range of data smoothing techniques including Good-Turning estimate, curve fitting functions, and term pairing, creating a bigram model. They discovered that the term pairing improved the performance of their information retrieval. This is something that is different from our experiment and could test out in the future.

In a study by Saha et al [11], they studied the effectiveness of information based fault localization on C. Up to this point bug localization methods have only been tested on object oriented programming languages such as Java. This gave a richer perspective on previous bug localization studies. They found that IR-based fault localization works comparably in C to Java code, however they found that BLUiR worked significantly less effectively on C code than Java code due to less structure. Testing in projects of different languages is not something we tried in our experiment and we could do to improve our findings.

In a study by S. Miryeganeh [13], they explored the use of doc2vec word embedding to improve bug localization. The use of word embedding techniques existed in other studies, but this study is unique that it tries to utilize a global word embedding technique. All other IR-based fault localization techniques I've seen previously had only used local information retrieval to conduct fault localization. This study utilizes the large amount of open source project data available on the internet to train a neural network to be able to vectorize any given source file of bug report. This is an idea we did not explore in our study; our study is completely local and does not take into account any global data to help localize our bugs.

In a study by Takahashi et al [13], they used the unique idea of code smells to help improve fault localization. Code smells are patterns in code that could indicate the likelihood of a bug appearing in the code, this is a term commonly used in static analysis. Other studies have assumed that all source code files are equally likely to contain the bug, however using code smells we could determine probabilities of source code files containing bugs before doing and fault localization on them. This is an interesting idea that we did not experiment with in our study.

In a study by O. Chaparro, J. Florez, and A. Marcus [14], they conducted IR-based fault localization by breaking up the bug reports into separate components, similar to what BLUiR did with the source code files. Chaparro and Florez separated the bug reports into the title, observed behaviour, expected behaviour, and step to reproduce. They tried different combinations of these four components on fault localization techniques (such as BugLocator, BRTracer, and locus), and found that the combination of title, observed behaviour, and steps to reproduced performed the best in performance.

VII. Conclusion

Locating bugs is important, difficult, and expensive, particularly for large-scale software projects. Several different methods of locating bugs have been proposed and tested against each other. We took a look at some of the best performing methods for locating bugs and compared them against each other. We took a look specifically at information retrieval based fault localization techniques. We took a look at rVSM, BugLocator, and BLUiR. First we implemented rVSM and BugLocator, then we looked at BLUiR which was originally proposed in response to BugLocator to try and improve the performance of fault localization. In our experiment, we found the BLUiR did not perform better than BugLocator in our dataset. However, we also looked at the possibility of combining both BugLocator and BLUiR to create a better and improved BugLocator 2.0. Our results showed that our proposed BugLocator 2.0 did not improve the original BugLocator but we hypothesized that it was possible to improve BugLocator by combining them together using a weighted sum. This is something we can take a look at in a future study. Our study concluded that BugLocator significantly performed better than all other information based fault localization technique that we tested.

References

[1] ZHOU, Jian; ZHANG, Hongyu; and LO, David. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. (2012). ICSE 2012: 34th International Conference on Software Engineering, Zurich, June 2-9. 14-24. Research Collection School Of Information Systems

[2] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, “Improving bug localization using structured information retrieval,” 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013.

[3] G. Salton, A. Wong, C. S. Yang, ”A vector space model for automatic indexing,” Communication of the AMC, 11th ed., vol. 18. New York: Association of Computing Machinery.

[4] D. Kundel, “Introduction to abstract syntax trees,” Twilio Blog, 11-Jun-2020. [Online]. Available: https://www.twilio.com/blog/abstract-syntax-trees. [Accessed: 14-Dec-2021].

[5] S. Rao and A. Kak, “Retrieval from software libraries for bug localization,” Proceeding of the 8th working conference on Mining software repositories - MSR ’11, 2011.

[6] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. L. Traon, “Bench4BL: reproducibility study on the performance of IR-based bug localization,” Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2018.

[7] S. Cheng, X. Yan, and A. A. Khan, “A Similarity Integration Method based Information Retrieval and Word Embedding in Bug Localization,” 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS), 2020.

[8] S. A. Akbar and A. C. Kak, “A Large-Scale Comparative Evaluation of IR-Based Tools for Bug Localization,” Proceedings of the 17th International Conference on Mining Software Repositories, 2020.

[9] T.-D. B. Le, R. J. Oentaryo, and D. Lo, “Information Retrieval and Spectrum Based Bug Localization: Better Together,” School of Information Systems Singapore Management University, 2012.

[10] F. Song and W. B. Croft, “A general language model for information retrieval,” Proceedings of the eighth international conference on Information and knowledge management - CIKM 99, 1999.

[11] R. K. Saha, J. Lawall, S. Khurshid, D.Perry, ”On the Effectiveness of Information Retrieval Based Bug Localization for C Programs”

[12] Miryeganeh, S. N. (2019). Applications of Text Mining Techniques on Automated Software System Verification (Unpublished master’s thesis). University of Calgary, Calgary, AB.

[13] A. Takahashi, N. Sae-Lim, S. Hayashi, M. Saeki, ”An extensive study on smell-aware bug localization” Tokyo Institute of Technology.

[14] O. Chaparro, J. M. Florez, and A. Marcus, “Using bug descriptions to reformulate queries during text-retrieval-based bug localization,” Empirical Software Engineering, vol. 24, no. 5, pp. 2947–3007, 2019.