Are Deep Learning-based Software bugs different than traditional bugs for bug localization? An empirical study.

Sigma Jahan
Faculty of Computer Science
Dalhousie University
Halifax, Canada
sigma.jahan@dal.ca

Harshit Lakhani
Faculty of Computer Science
Dalhousie University
Halifax, Canada
harshit.lakhani@dal.ca

Meghna Rupchandani
Faculty of Computer Science
Dalhousie University
Halifax, Canada
mg841071@dal.ca

Sharad Kumar
Faculty of Computer Science
Dalhousie University
Halifax, Canada
Sharad.Kumar@dal.ca

***Abstract*—Developers need to know which files should be updated to repair an issue when a new bug report emerges. They may need to examine multiple source code files to identify the issue in a significant software project, which might be time-consuming and expensive. Based on initial bug reports for real-world open-source projects, contemporary IR-based algorithms efficiently locate relevant source code files. But bugs for testing and debugging differ for each subject system (e.g., programming language, application, software). Deep learning-related software bugs are more complex and challenging to debug than traditional bugs. In this paper, firstly, we plan to determine the performance difference between deep learning-based bugs and traditional software bugs using existing IR-based techniques for bug localization. Secondly, we intend to understand the impact of extrinsic and intrinsic bugs on deep learning software-based bugs and whether it correlates with bug localization performance. Lastly, to address the performance of deep learning-based software bugs for bug localization, we propose using a pre-trained language model to determine whether the performance improves compared to IR-based techniques.**

***Index Terms*—bug localization, deep learning, extrinsic bug, CodeBERT**

I. Introduction

Software bugs are increasingly prevalent due to the inherent complexity of software development. The number of bugs in an extensive software system might range from hundreds to thousands. Bug fixing commonly involves locating relevant buggy source code, known as bug localization. However, performing this process manually for many bugs is time-consuming and costly. As a result, any software industry would benefit significantly from practical methods for automatically locating bugs from bug reports. Information retrieval-based bug localization approaches have garnered significant interest in recent years because of their inexpensive computing cost and little external dependencies as they solely require source code and a bug report to perform [1]. There is a significant distinction between AI applications that employ deep learning models and traditional software applications [2, 3, 4]. Considering the demand for deep learning-based software (DLSW), an automatic debugging approach should be created to ensure that deep learning-based software is high-quality [5]. While bug localization is one of the most critical steps in bug management, determining whether deep learning-based software bugs differ from conventional bugs is essential to automate the process for industrial applications. We thus answer three important research questions in our study as follows.

(a) **$\mathbf{RQ_1}$: How does existing IR-based bug localization techniques perform on deep learning-based bugs?**
We conducted experiments on the benchmark dataset [5] using three existing techniques VSM, rVSM and BM25 in bug localization that leverage Information Retrieval (IR). Then we have determined the performance of each technique against the dataset. Later, we compared the result with our proposed methodology. BM25 outperforms rVSM and VSM by scoring MAP@10: 64.71% & MRR@10: 74.96%.

(b) **$\mathbf{RQ_2}$: What is the impact of extrinsic and intrinsic bugs for deep learning software based bugs?**
Depending on the origin, bugs might be intrinsic or extrinsic. Intrinsic and extrinsic code changes have distinct characteristics [6]. We have manually examined on a small subset of dataset (500) for finding out the percentage of extrinsic bugs. We found 19.4% bugs are indeed extrinsic which negatively impact the performance of bug localization methodologies in deep learning software bugs. We have also compared the percentage of extrinsic bugs for the deep learning-based software bugs and traditional python based software bugs. We found that deep learning-based software bugs contains four times more extrinsic bugs than traditional software bugs.

(c) **$\mathbf{RQ_3}$: Does using pre-trained language model on deep learning-based software bugs improve the performance of bug localization?**
Our experiments in RQ_1 use IR-based techniques for bug localization in our dataset [5] using VSM, rVSM and BM25 which might not have been adequate to overcome the challenge of deep learning software-based bugs. We

TABLE I

Methodologies	Similarity with Bug Reports		
	File Similarity	Bug Similarity	Hunk Similarity
BugLocator	Yes	Yes	
AmaLgam	Yes	Yes	
BLuiR	Yes	Yes	
Locus	Yes	Yes	Yes
Blizzard	Yes	Yes	

have implemented pre-trained language model (such as BERT and CodeBERT) [7] on deep learning software-based bugs for bug localization to check whether it improves the performance. Bert scored MAP@10: 29.67% & MRR@10: 38.62% for the experiment which indeed did not work as we anticipated.

II. Related work

A. Software Bug

For testing and debugging, it's crucial to understand the bugs and faults in huge software repositories depending on multiple subject systems. Such as based on programming language, Python stands apart from other popular languages like Java because of features like duck typing and the widespread usage of heterogeneous collections [8]. As a result, the characteristics of bugs found in Python projects based on deep learning are likely to differ from bugs found in other programming languages. One study focused on the deployment of deep learning based software, proposing a taxonomy and obstacles for deep learning based software deployment, as well as mentioning the demand for deep learning based software-specific fault localization approaches for automated debugging [2]. Researchers looked at patterns of code smells in DLSW in another study and observed that the existence of code smells might influence bug occurrence. [9]. Data bugs and logic bugs are the most severe in DLSW, according to Johirul et al. [10], however, this may not be the case for all Python-based applications. In a consequent study, they also showed that deep learning models' bug and repair patterns significantly differ from traditional software [11]. As a result, research concentrating on deep learning-based software bug benchmarks is necessary for creating or developing automatic debugging strategies for deep learning-based software.

B. Bug Localization

In recent years, IR-based bug localization research has made significant progress. Zhou et al. [12] introduced BugLocator, a scheme that leverages textual similarity between bug reports and source code to rank all files in a software repository. Larger files are ranked higher, and term frequency is calculated as a logarithmic function of several terms. Information concerning previously fixed bugs was taken into account throughout the ranking process. BugLocator customarily surpassed models based on VSM, LDA, Smoothed Unigram Model, and LSI, according to the outcomes. Although it has improved the outcomes but the accuracy is still low. Hence, BLUiR, a technique for locating relevant resources for bug fixes, was introduced by Saha et al. [13]. This method also uses the textual similarity between source code and bug reports to rank all files in a software repository. However, the textual similarity is determined using the Okapi-BM25 model [14], a bag-of-words retrieval function. BLUiR also leverages structured information retrieval of code components, including method and class names, which boosts bug localization accuracy and beats buglocator. Wang and Lo [15] developed AmaLgam, an approach for finding related buggy files that incorporates BugLocator [12], which assesses similar reports from bug reporting systems, with BLUiR [13], which takes structure into account. The quality of provided bug reports makes traditional IR-based bug localization challenging. As a result, in another research [16], they developed a Blizzard that leverages the quality element of bug reports by combining context-aware query reformulation into bug localization, overcoming the restriction. But all these researchers used traditional bugs for implementing their methodologies which gave them decent performance, but for deep learning-based software bugs, these methods might not be ideal; hence in this study, we are performing an empirical study to find the performance issue for deep learning-based software bugs in terms of automating bug localization and planning to provide a solution as well.

From most of the existing work, table I, we noticed that they have used similarity between bug reports and file (source code) to determine the location of buggy files hence we decided to use the similarity approach for experimenting on both existing methodologies and our proposed method to determine the performance of bug localization.

III. Methodology

For this project we intend to follow Cross Industry Standard Process for Data Mining (CRISP-DM) [17] as a blueprint which is a data mining methodology.

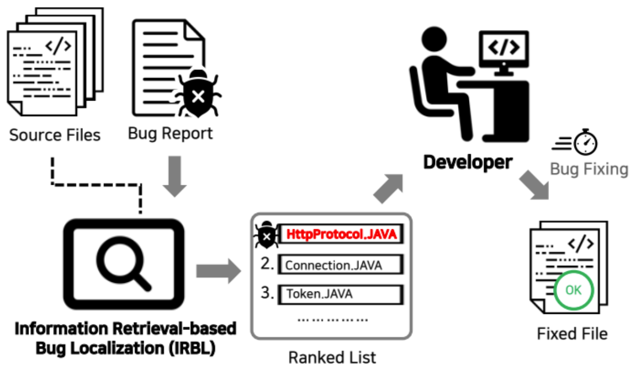


Fig. 1. IR based methodology for bug localization [5]

A. Research Understanding:

Deep learning model-based software bugs significantly differ from traditional software bugs. Our research focuses on deep learning-based software bugs, finding the experimental difference in bug localization, and building a suitable technique concentrating on this specific bug. In this research, the ultimate goal is to save up time and cost for debugging, which benefits software industries from a business perspective.

B. Data Understanding:

We have used a benchmark dataset to conduct our analysis. Our dataset is built from deep learning-based software, which is called Denchmark [5], according to the original research work. Our sampled deep learning based dataset from Denchmark contains 4,301 bug reports from 9 widely used deep learning software projects [5]. They have used GitHub search API to gather bug reports from Deep Learning Software-based projects. These reports include textual descriptions, comments, and HTML tags. To monitor human tagging and offer code information, the dataset also includes HTML tags with the term ”code” in them. To address the concept drift issue, the chosen projects are popular, active, and whose most recent commit is after 2020. After collecting the dataset, we have used stratified sampling in order to choose the sampled dataset from 9 different software projects which had adequate bug reports with available source code.Fig. 2 shows the bug count for our dataset. Based on the shared projects among the themes, we have split them into strata. A different probability sampling technique is used to sample each subgroup once it has been split randomly. Our chosen projects are named as albumentations-team+albumentations, alibaba+pipcook, allegroai+clearml, allenai+allenlp, amaiya+ktrain, apache+incubator-mxnet, apache+tvm, apple+coremltools, and apple+turicreate. Each bug report includes links to the ground-truth files and functions (methods) as well.

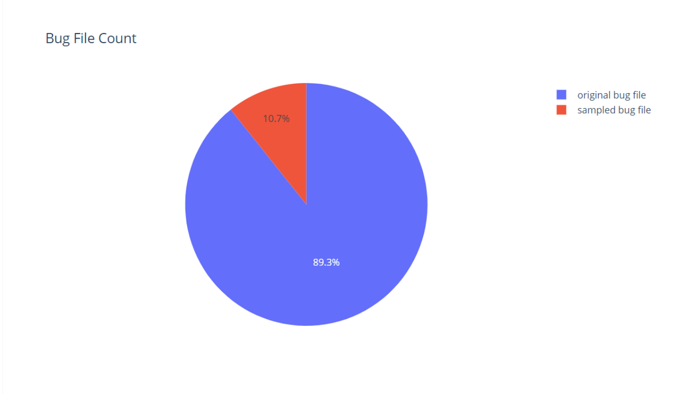


Fig. 2. Bug file count

C. Data Preparation:

There are four phases in the typical bug localization process: corpus development, indexing, query formulation, and retrieval and ranking. After collecting the dataset from two different sources individually, they are cleaned and preprocessed separately. As a result, a variety of measures are to be undertaken. For corpus creation, we built a vector of lexical tokens by doing lexical analysis on each source code file. Some tokens are common to all programs, such as keywords, separators, and operators, and are eliminated. ”Stop words” in English are also

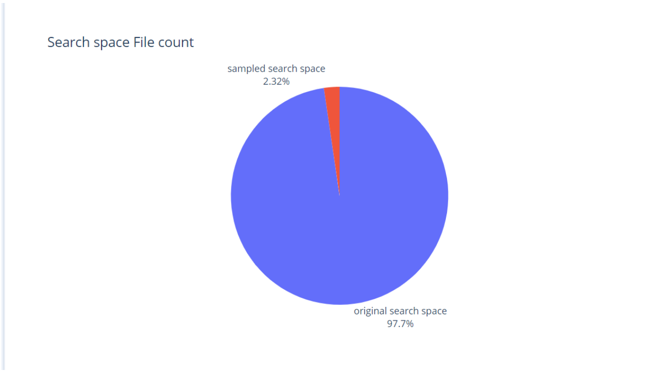
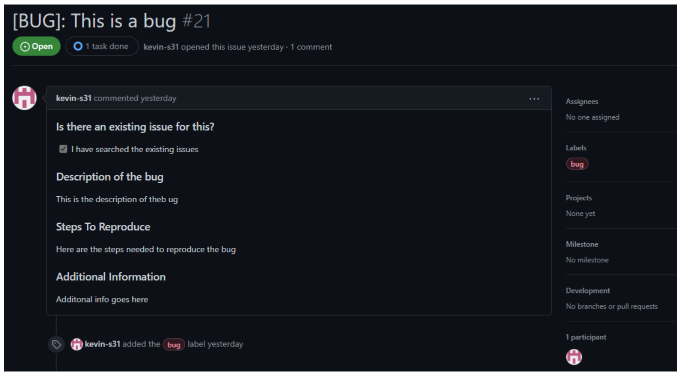


Fig. 3. Search space count

eliminated. Many of the variables in a program are made up of a string of words. Individual tokens are created from composite tokens. The Porter Stemming procedure reduces a word to its root since many tokens have the same root form. All of the files in the corpus are indexed when it is formed. These indexes enable to find files that contain the terms in a query and then rank them according to their relevance. Both dataset are checked for a null value. For the textual data in bug reports, often, the description contains source code or extensive data, which is usually not relevant, hence for bug reports with a description that contains more than 500 characters for those, we have only kept the first 500 characters without white space and discarded the rest of the string. Then, a few natural languages preprocessing are applied to a description, such as removing punctuation, non-alphanumeric character, number, HTML meta tag, URL, and finally converting all texts into lower cases. After that, we have used essential English stop words removal, tokenization, and lemmatization on description to transform the terms into their base form with its context. Our main features of this experiment are bug id, summary and description of the bug reports. Bug reports are often considered as imbalanced datasets hence we have used a thorough natural language pre-processing techniques to reduce the impact of the imbalance nature of the dataset. We have defined each project’s source file search space to locate bugs. The search space for each bug report was chosen as the most current version of the reporting date. Since the ground-truth files are expected to be in the search space for this experiment on deep learning-based defects, we have rejected bug reports in which at least one of the ground truth files or functions was not in the search space. Using the Python AST parser, all functions in the source files are retrieved and specified as a search area for function localization.

Fig. 4 shows bug reports are separated into a summary and a description. Texts with summaries and descriptions make up the queries for standard IR-based bug localization. A bug report is treated as a query by bug localization, which searches the indexed source code corpus for relevant files. The query is formed by extracting tokens from the bug title and description, removing stop words, stemming each word, and forming the query.


[BUG]: This is a bug #21

Open · 1 task done · kevin-s31 opened this issue yesterday · 1 comment

kevin-s31 commented yesterday

Is there an existing issue for this?

- [x] I have searched the existing issues

Description of the bug

This is the description of theb ug

Steps To Reproduce

Here are the steps needed to reproduce the bug

Additional Information

Additonal info goes here

kevin-s31 added the bug label yesterday

Assignees: No one assigned

Labels: bug

Projects: None yet

Milestone: No milestone

Development: No branches or pull requests

1 participant

Fig. 4. Bug Report

Later,

D. Modeling:

For first research question, we plan to use three exiting IR-based techniques [12, 18, 19] on both dataset to generate their performance for comparison. The first technique is based on an TF-IDF and Vector Space Model (rVSM) using text data past bug reports [12]. Second technique is developed on Vector Space Model and cosine similarity using text data from bug reports [18]. The third technique is based on BM25 for locating bugs [19]. For our last research question, we want to implement a language model BERT and CodeBERT, a bimodal pre-trained model for programming language and natural language for locating bugs from bug reports which has the potential to enhance the performance of deep learning based software bugs.

In the realm of information retrieval, the term frequency inverse document frequency (TF-IDF) based vector space model (VSM) and BM25 are the two basic models utilized for document similarity. The fundamental element of IR-based bug localization is the resemblance between a bug report and a source file. The revised VSM (rVSM) was suggested by Zhang et al. because lengthier source files typically have more bugs. In our first research question, we identified which of the three currently available similarity models is most effective for localizing problematic deep learning-based files.

1) VSM:: Each document is expressed as a vector of token weights in the Vector Space Model (VSM), which is commonly calculated as the product of each token's token frequency and inverse document frequency [20]. Cosine similarity is widely applied to gauge how closely the two vectors are related. One research [18] has assessed the effectiveness of the VSM model in locating bugs in an early study.

2) rVSM:: The conventional VSM approach for bug localization might be optimized by using a revised Vector Space Model (rVSM) to rank all source code files based on an initial defect report that considers document length [12]. By incorporating data from pertinent earlier bug reports, the original study [12] in this model additionally altered the resulting ranks.

3) BM25:: The ranking function BM25 (Best Matching 25), which we employed in this experiment as it is another commonly applied technique for duplicate bug report detection, was developed in the Okapi information retrieval system [21], which is an effective textual similarity function [22]. In BM25, documents are ranked using a Bag of Words (BOW) retrieval function. Each term is considered a query term to calculate the probabilistic and statistical dependency of term occurrences across all documents [21]. We chose the parameters of k = 2.0 and b = 0.8 from research [23] because they have shown to deliver stable results. Using BM25, we got scores according to the ranking shown in fig 5. In a later study [19], they have shown BM25 outperforms both VSM and rVSM for bug localization. This approach ranks all bugs

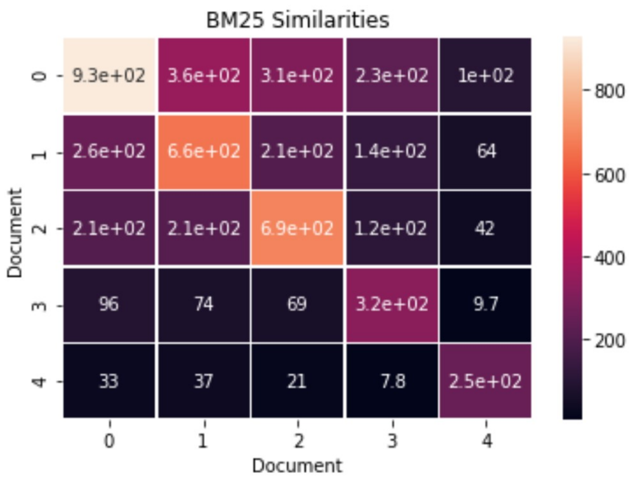


Fig. 5. BM25 Similarity Matrix

given a query bug since it recommends related bugs based on similarity scores. As a result, any number smaller than the overall number of pending bugs for k can be utilized to get the top-k suggestion. In a top-k suggestion, we advise k most similar bugs for each query bug with the highest matching score.

4) Pre-trained language modeling (BERT):: We chose BERT [24] as the pre-trained language modeling to creating the embedding matrix for similarity calculation because it is cutting-edge in semantic modeling and retrieving contextual data. BERT's architecture comprises several layers of transformer encoders that use self-attention, where the idea of attention is to provide different weights to specific words in the sequence [24]. Existing bug localization methods might be significantly enhanced by utilizing a model that leverages attention to highlight specific word associations [25]. We employed a pre-trained BERT model due to data and computing resource limitations. The BERT model produces an embedding matrix of size the number of words in the document by 728 after encoding each word in the input bug report with a vector [24]. An aggregation layer then converts the embedding matrix to a vector. The vector is then fed into a final layer, which uses cosine similarity to provide a relevance score.

5) Similarity measurement:: Cosine similarity measurement [26] is implemented to measure the similarities of documents from the vector representation. The cosine similarity measurement is the most widely used and has been shown in [27] to outperform the other similarity measures. Cosine similarity is a measurement to determine how similar two documents are regardless of their size. This is beneficial because even if two similar documents are separated by the Euclidean distance [21] according to document size, they are likely to be orientated closer together [26]. Cosine similarity has the benefit of being concise, especially for sparse vectors, because only non-zero dimensions need to be regarded. Cosine similarity is used to measure similarities for detecting duplicate bug reports as a common approach [12, 13, 15, 18]

$$Similarity(x, y) = Cos(\theta) = \frac{(x.y)}{(|x||y|)} \tag{1}$$

The cosine similarity is computed using the dot product v1 and v2 for vectors representing two bug reports. Equation above demonstrates the formula of the approach. The smaller the value, the closer the two vectors are to collinearity, and hence the two reports share more weighted words.

E. Evaluation:

We will employ a traditional IR-based approach, a conventional text retrieval that matches a query of keywords with a collection of documents in a corpus. The retrieved documents are sorted in a list style based on their similarity to the query [28]. The fundamental notion is that an automated IR-based technique must place linked files at the top of the sorted list to be effective. In the following sections, we outline the measurements we will use to evaluate our models.
MAP@K: MAP@K considers the order of accurate outcomes in a ranked list. Precision@K generates precision for each buggy file in the list upon occurrence. The average Precision@K for all the buggy files in a ranked list for a particular query is called Average Precision@K. Mean Average Precision@K is the mean of all queries' Average Precision@K [29].
MRR@K: The multiplicative inverse of the rank of the first successfully returns buggy file inside the Top-K results is Reciprocal Rank@K. Mean Reciprocal Rank@K (MRR@K) is a method for averaging such measurements over all queries in a dataset [29].
We have selected top 10 as the rank value to calculate MAP@K and MRR@K. After conducting the experiments, we have evaluated existing IR-based techniques using MAP@K and MRR@K as used by existing work [12, 19, 20]. On the other hand, we have evaluated our techniques using BERT [7] using the same performance metrics for better performance.

IV. Study Findings

In this section, we present the results of our study by answering the three research questions. We first show the performance of existing techniques on our dataset for deep learning based bugs (RQ_1). Then we analyze the data for extrinsic and intrinsic bug reports to show whether this factor is playing any role in bug localization (RQ_2). Finally we apply pre-trained language model BERT to analysis whether it overcomes the challenges of IR-based bug localization for deep learning based bugs in (RQ_3) as follows:

A. RQ_1: How does existing IR-based bug localization techniques perform on deep learning-based bugs?

In this research question, we experimented three existing work (VSM, rVSM and BM25) on our dataset which is deep learning related software bugs for locating buggy files. From fig 6, we can notice that the performance of all three existing techniques are not up to the mark. However, BM25 outperforms (MAP@10: 64.71% & MRR@10: 74.96%) rVSM (MAP@10: 56.40% & MRR@10: 67.81%) and VSM (MAP@10: 46.39% & MRR@10: 54.53%) where the performance of VSM is the poorest among three.

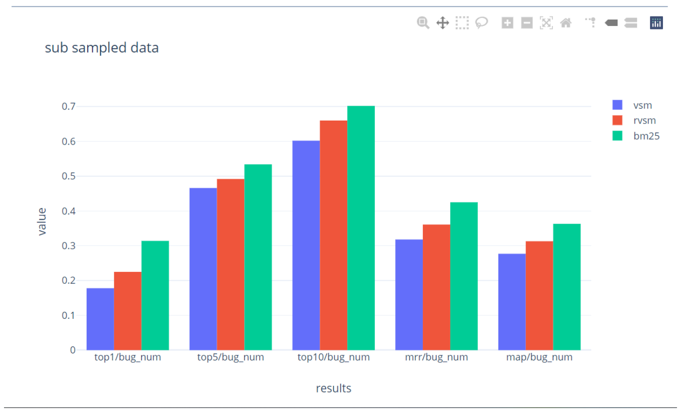


Fig. 6. Result of existing work on sub-sampled data

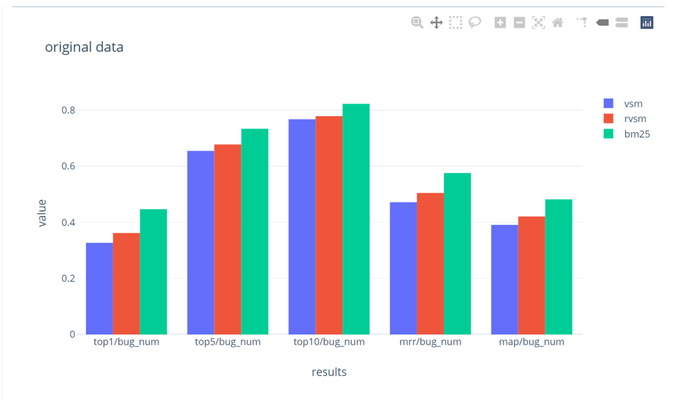


Fig. 7. Result of existing work on original data

Summary of RQ_1: The performances of existing techniques (e.g., VSM, rVSM and BM25) are significantly not up to the standard in terms of detecting the location of buggy files for deep learning based bugs. BM25 outperforms rVSM and VSM by scoring MAP@10: 64.71% & MRR@10: 74.96%.

B. *RQ$_2$: What is the impact of extrinsic and intrinsic bugs for deep learning software based bugs?*

We have adhered to a study paper's recommendation [6] that stated a bug report is extrinsic if it details a defect brought on by a modification to the environment in which the software is operated. Second, it reports a bug because the requirements have changed and a defect spurred on by an outside change to the project's version control system. Finally, it discloses a fault in the project's third-party library. It is considered intrinsic if there is no supporting information to classify a bug report as an extrinsic bug. We have chosen a small subset of bug reports from our original dataset containing 500 deep learning-based bug reports, which satisfies the criteria of 95% confidence interval and 5% margin error to label the extrinsic and intrinsic bugs manually. We got 19.4% extrinsic bugs from a total of 500 bug reports. The notion of extrinsic bugs is relatively new, especially in the case of deep learning bugs. Almost all the bug report management tools such as bug localization, bug classification, or duplicate bug report detection are based on intrinsic bugs; hence this other phenomenon of extrinsic bugs is causing performance degradation for deep learning-based bugs. Removing extrinsic bugs can improve the performance of bug localization in our case. To compare with other types of bugs, we have also manually inspected 500 python bugs that are specifically not deep learning-related bugs to get the ratio of extrinsic and intrinsic bugs. We got only 4.7% extrinsic bugs in python bugs, whereas deep learning-related bugs contain four times more extrinsic bugs. We can conclude by stating that extrinsic bugs are one of the contributing factors to the poor performance of bug localization in deep learning-based software.

Summary of RQ$_2$: Extrinsic bug reports are considered any bug that has occurred for external factors. From manual inspection of a sample deep-learning-based dataset, we got 19.4% extrinsic bugs from deep learning-based software bugs and only 4.7% extrinsic bugs from known python-based bugs. Extrinsic bugs lead to poor performance for traditional IR-based methodologies for bug localization in deep learning-based software.

C. *RQ$_3$: Does using pre-trained language model on deep learning-based software bugs improve the performance of bug localization?*

In the third research question, we used the pre-trained language model BERT to check whether it positively impacts bug localization for deep learning-based software bugs. However, BERT performed poorly for bug localization compared to the other three existing models. BERT has an accuracy of MAP@10: 29.67% & MRR@10: 38.62%, which is the lowest score. We also experimented with CodeBERT, an updated variation of BERT, but CodeBERT performed worse. Our reasoning for the performance would be we have used pre-trained BERT and CodeBERT due to the limitation of massive data and computational resources. Both models are not explicitly pre-trained in the software domain; hence that might be one crucial factor. Another notable point is that pre-trained embedding models do not work well for smaller datasets. In contrast, we have only 4,301 bug reports to experiment on, which is why the performance is also suffering.

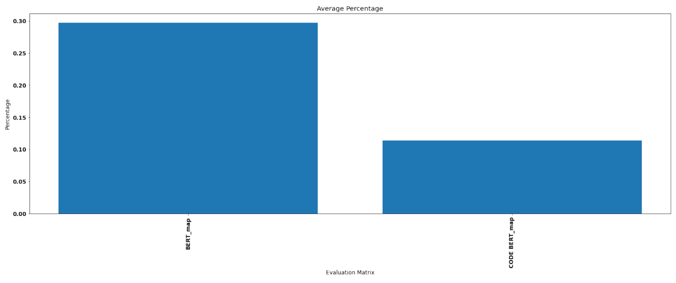


Fig. 8. Result of Bert and CodeBert using MAP@K (K=10)

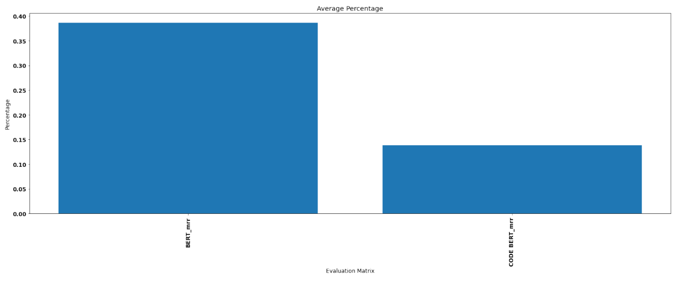


Fig. 9. Result of Bert and CodeBert using MRR@K (K=10)

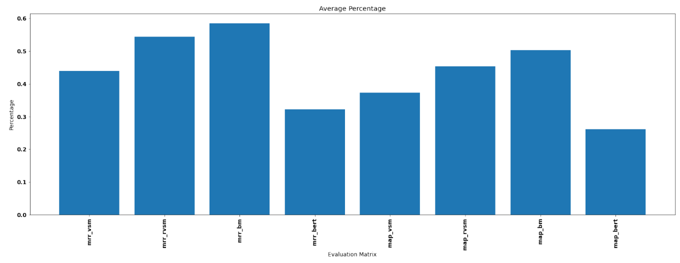
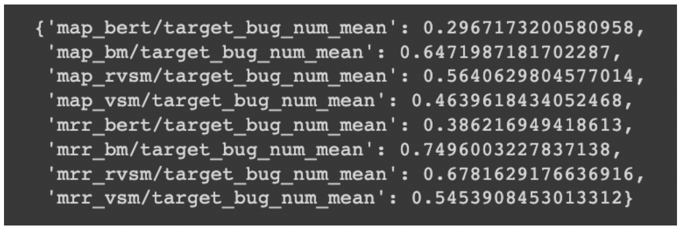


Fig. 10. Result analysis of all the implemented methodologies using MAP@K and MRR@K (K=10)



```
{'map_bert/target_bug_num_mean': 0.2967173200580958,
 'map_bm/target_bug_num_mean': 0.6471987181702287,
 'map_rvsm/target_bug_num_mean': 0.5640629804577014,
 'map_vsm/target_bug_num_mean': 0.4639618434052468,
 'mrr_bert/target_bug_num_mean': 0.386216949418613,
 'mrr_bm/target_bug_num_mean': 0.7496003227837138,
 'mrr_rvsm/target_bug_num_mean': 0.6781629176636916,
 'mrr_vsm/target_bug_num_mean': 0.5453908453013312}
```

Fig. 11. Performance of all the implemented methodologies using MAP@K and MRR@K (K=10)

Summary of RQ$_3$: Pre-trained language modeling such as BERT or CodeBERT did not perform well for bug localization in deep learning-based software bugs mainly because the models are not pre-trained in the software domain, and the dataset was too small for implementing pre-trained language modeling. Bert scored MAP@10: 29.67% & MRR@10: 38.62% for the experiment.

V. Threats to Validity

Threats to internal validity relates to experimental errors and human biases [30]. Traditional bug tracking systems have thousands of reports whose quality cannot be guaranteed. Bug reports often contain poor, insufficient, missing, or even inaccurate information [31]. To address the issue, we applied standard natural language preprocessing and token threshold to them and also check for missing features in each bug report. Another potential source of threat could be the reproduction of existing work. However, we did it carefully using standard libraries and corresponding papers, tuned the parameters, and reported the best results.

Threats to external validity relates to the generalizability of experimental findings. We chose a total of 4301 bug reports and 2643265 files (source code) from nine large-scale open source projects (e.g., apache+tvm, allenai+allenlp, apple+turicreate) that span different application domains and programming platforms. Thus, the threats to external validity might be mitigated.

Threats to conclusion validity is the observations from our study and the conclusions we drew from them could be a source of threat to conclusion validity [32]. In this research, we answer three research questions using 4301 bug reports and 2643265 files (source code) from nine different projects and implementing three existing work. Thus, such threats might also be mitigated.

Threats to construct validity This relates to the use of appropriate performance metrics. We have used standard evaluation metrics such as MRR and MAP to draw any conclusion. Thus, such threats might also be mitigated.

VI. Conclusion and Future Work

Bug localization is vital to detect the problematic software pieces precisely, as this is a high-stakes operation. According to a poll conducted in 2018 [33], bug localization is the most significant duty in Bug Report Management. As a result, bug localization automation is a popular research topic. However, with the growing need for deep learning-based software, existing solutions may not be sufficient. As a consequence of this study, we have explored the performance differences of deep learning-based software defects. We have pointed out the impact of extrinsic bugs in deep learning-based bug reports, and lastly, we proposed to utilize a language modeling (BERT) for bug localization of deep learning-based bugs. This study is designed to assist developers and the software industry directly. After conducting this empirical study, we have a few suggestions and areas to explore in the future, which are mentioned here. Because the performance for deep learning-based bugs is worse than anticipated, we must develop methods to increase IR-based performance for deep learning bugs. First, as the implications of features vary between these bugs, we should first automatically categorize bug reports into deep learning-based and non-deep learning-based bug types. To improve performance, extrinsic bugs from deep learning-based bugs must be eliminated. We should thoroughly study the text characteristics in the deep learning-based bug report to find out why IR-based performance is inferior and why even pre-trained language modeling is not enhancing the performance of bug localization. All of our experiments and dataset are available in the following GitLab repository (https://git.cs.dal.ca/sharad/datascience).

References

[1] D. Binkley and D. Lawrie, "Information retrieval applications in software maintenance and evolution," *Encyclopedia of software engineering*, pp. 454–463, 2010.

[2] Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu, "A comprehensive study on challenges in deploying deep learning based software," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 750–762.

[3] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 291–300.

[4] D. Gonzalez, T. Zimmermann, and N. Nagappan, "The state of the ml-universe: 10 years of artificial intelligence & machine learning software development on github," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 431–442.

[5] M. Kim, Y. Kim, and E. Lee, "Denchmark: A bug benchmark of deep learning-related software," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 540–544.

[6] G. Rodriguezperez, M. Nagappan, and G. Robles, "Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the openstack project," *IEEE Transactions on Software Engineering*, 2020.

[7] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[8] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh *et al.*, "Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies," in *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1556–1560.

[9] H. Jebnoun, H. Ben Braiek, M. M. Rahman, and F. Khomh, "The scent of deep learning code: An empirical study," in *Proceedings of the 17th International*

Conference on Mining Software Repositories, 2020, pp. 420–430.

[10] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 510–520.

[11] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1135–1146.

[12] 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)*. IEEE, 2012, pp. 14–24.

[13] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 345–355.

[14] S. Robertson and H. Zaragoza, *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc, 2009.

[15] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 53–63.

[16] M. M. Rahman and C. K. Roy, "Improving ir-based bug localization with context-aware query reformulation," in *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.

[17] R. Wirth and J. Hipp, "Crisp-dm: Towards a standard process model for data mining," in *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining*, vol. 1. Manchester, 2000, pp. 29–40.

[18] 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*, 2011, pp. 43–52.

[19] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan, "The impact of classifier configuration and classifier combination on bug localization," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1427–1443, 2013.

[20] G. Salton, A. Wong, and C.-S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[21] S. E. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford, "Okapi at trec-3," in *TREC*, 1994.

[22] S. E. Robertson and S. Walker, "Some Simple Effective Approximations to the 2-Poisson Model for Probabilistic Weighted Retrieval," 1994, pp. 232–241.

[23] C.-Z. Yang, H.-H. Du, S.-S. Wu, and I.-X. Chen, "Duplication Detection for Software Bug Reports Based on BM25 Term Weighting," in *2012 Conference on Technologies and Applications of Artificial Intelligence*, 2012, pp. 33–38.

[24] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[25] A. Ciborowska and K. Damevski, "Fast changeset-based bug localization with bert," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 2022, pp. 946–957.

[26] S. Prabhakaran, "Cosine similarity - understanding the math and how it works? (with python)," 2018. [Online]. Available: //shorturl.at/euv69/

[27] J. Deshmukh, K. Annervaz, S. Podder, S. Sengupta, and N. Dubash, "Towards accurate duplicate bug retrieval using deep learning techniques," in *2017 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 115–124.

[28] A. Singhal *et al.*, "Modern information retrieval: A brief overview," *IEEE Data Eng. Bull.*, vol. 24, no. 4, pp. 35–43, 2001.

[29] D. Harman, "Information retrieval evaluation," *Synthesis Lectures on Information Concepts, Retrieval, and Services*, vol. 3, no. 2, pp. 1–119, 2011.

[30] Y. Tian, D. Lo, and J. Lawall, "Automated construction of a software-specific word similarity database," in *Proc. CSMR-WCRE*, 2014, pp. 44–53.

[31] L. Kang, "Automated Duplicate Bug Reports Detection - An Experiment at Axis Communication AB," 2017. [Online]. Available: https://bit.ly/37Ylk5z

[32] M. A. García-Pérez, "Statistical conclusion validity: Some common threats and simple remedies," *Frontiers in psychology*, vol. 3, p. 325, 2012.

[33] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu, "How practitioners perceive automated bug report management techniques," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 836–862, 2018.