Received 12 February 2023, accepted 5 April 2023, date of publication 10 April 2023, date of current version 13 April 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3265731

RESEARCH ARTICLE

Enhancing Bug Localization Using Phase-Based Approach

AMR MANSOUR MOHSEN[1], **HESHAM A. HASSAN**[2], **KHALED T. WASSIF**[2], **RAMADAN MOAWAD**[1], **AND SOHA H. MAKADY**[2]

[1]Department of Computer Science, Faculty of Computers and Information Technology, Future University in Egypt, Cairo 11835, Egypt
[2]Department of Computer Science, Faculty of Computers and Artificial Intelligence, Cairo University, Cairo 11865, Egypt

Corresponding author: Amr Mansour Mohsen (Amr.Mansour@fue.edu.eg)

This work was supported by Future University in Egypt.

ABSTRACT Software bug localization is an important step in the software maintenance process. Automatic bug localization can reduce the time consumed in the process of localization. Some techniques are applied in the bug localization process, but those techniques suffer from limitations in time and accuracy. This paper proposes a phase-based bug localization approach to overcome these limitations. The approach is composed of three main phases which are raw data preparation, package classification, and source code recommendation. The main input to our approach is a bug report and the source code of the past versions for the target system of interest. From the bug report, various information is utilized: the summary, the description, the stack traces, and the fixed source code files. The raw data preparation phase is used to restructure those inputs. The package classification phase aims to locate the package that would include the source code to be modified as a first step, hence reducing the time needed to locate the source code file due to the lexical mismatch between those files and the bug report data. Bidirectional Encoder Representations from Transformers (BERT), which is a sentence embedding technique, is utilized in the package classification and source code recommendation phases. The experimental results show that our approach outperforms existing approaches according to TOP-N rank and Mean Reciprocal Rank (MRR) evaluation metrics.

INDEX TERMS Bug localization, bidirectional encoder representations from transformers (BERT).

I. INTRODUCTION

The main software processes of building the software are software specification, software design, implementation, software validation, and software evolution [1]. The software evolution phase involves accommodating new requirements as well as fixing the found bugs. Such phase is the longest phase of the software lifecycle and consumes up to 70% of its time. Bug localization is one of the most important tasks in software evolution. Bug localization is the process of finding a bug or error that appeared in the software during operation. Locating the bug manually is an expensive and time consuming process. [2].

Many automatic bug localization techniques are applied to find the bugs in the software. Different software artifacts are utilized to localize bugs as the source code, past bug reports, and test cases. The bug report is a document that illustrates the bugs that appeared in the software [3]. The bug report consists of the bug summary, description, stack traces and bug metadata like who reports the bug and who solved it. Some techniques utilize bug report's text directly in order to find the location of the bug using information retrieval techniques [4], [5], [6], [7]. Such techniques utilized similarity measures between bug reports text and different source code files. The calculated similarity scores for source code files are sorted in descending order to find the most related source code file to the bug report. Other techniques utilize machine learning to find bugs by feeding the information of past solved bugs for training the model [8], [9], [10], [11], [12], [13].

The above mentioned techniques suffer from three main limitations: (1) lack of bug localization artifacts (2) context mismatch, and (3) increased time to locate the bug in large code bases. We discuss those limitations as follows.

The associate editor coordinating the review of this manuscript and approving it for publication was Muhammad Ali Babar.

Limitation 1: (*lack of bug localization artifacts*), several techniques utilize old bug reports to locate the bug. Such techniques would not work if the analyzed project is a new one that does not contain any historical bug reports.

Limitation 2: (*context mismatch*), Information retrieval techniques only consider the text of the source code files to calculate the similarity measures, without considering the context of the analyzed text during the analysis. Such context would improve the classification accuracy if considered. For example, consider source code file1 contains the tokens [“bind”, “hello”, “setVisible”…], and source code file2 contains the tokens [“applet”, “setVisible”, “aspect” …]. Consider that a bug report text contains [“find”, “bug”, “setVisible” …]. If only text similarity is applied depending on an applied technique, the result may be only reference file 1 as the buggy source code file. However the source code file 2 may be the result. The context stated here is the source code file and the word “setVisible” that appeared in two different contexts which may result in misclassification.

Limitation 3: *(bug localization time),* some information retrieval techniques [4] that use similarity measures between the new bug reports text and all source code files text, take a significant time for applying the similarity. For example, JDT software project dataset contains more than 8000 source code files. If we want to apply a text similarity technique [14], the search time between the new bug report and all source code files will be on the 8000 source code files. So the time to search the bug would be large, and needs to be reduced.

To overcome the mentioned limitations of past techniques, a phase-based approach is proposed. The approach consists of three main phases. *Raw data preparation* phase is used for the preparing different software artifacts such as bug report text and source code files. Additionally, overcoming ***Limitation 1*** by combining the source code files text and old bug reports text if available for a specific source code package. Also if the old bug reports are not available, only the available source code files could be used for locating the bugs. *Package classification* which utilizes BERT [15] addresses ***Limitation 3*** by recommending only a software package of a software. The recommendation phase reduces the search time to find the buggy source code file by narrowing the space of search. Source code files recommendation phase outputs the specific buggy source code file by putting into consideration the context of source code files text thus overcoming ***Limitation 2.***

The main contributions of the paper are summarized as follows:

1) Improved bug localization technique that works for projects that do not have historical bug reports through a novel bug localization approach.
2) Introducing a package classification phase to produce the recommended package, in order to overcome the time problem. The package classification phase reduces the search space by recommending the buggy package. The recommended package consists of definite source code files and not all the source code files of the project.
3) We evaluate the performance of our phase-based bug localization approach on two different datasets in terms of accuracy and the needed time for bug localization. Our approach achieves an enhancement considering time and accuracy.

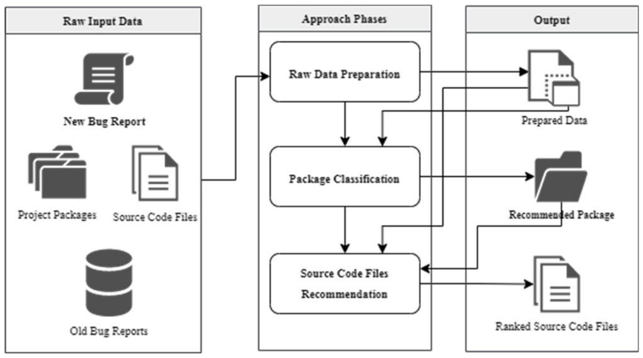


FIGURE 1. The Phase-Based Bug localization Proposed Approach Architecture.

The rest of the paper will be divided as follows: Section II describes the proposed approach. Section III presents the experiments and evaluations that were applied to verify the proposed approach. Section IV presents the threats to validity. Section V shows the related work, and Section VI concludes the paper.

II. THE PROPOSED APPROACH

In this section, the proposed Phase-based bug Localization (PBL) approach will be presented. As shown in Figure 1, the approach consists of three main phases: (1) raw data preparation, (2) package classification, and (3) source code files recommendation for fixing.

The raw data preparation phase involves preparing the artifacts of the project of interest for analysis. For any project, the following artifacts are given as an input to the data preparation phase: (a) the old resolved bug reports, (b) the source code of that project along with its packages’ structure, and (c) the new bug reports that need to be localized and fixed. Such software artifacts would be extracted from some bug tracking system and version control system for the software. The output of the raw data preparation phase is a structured and organized form of bug reports and source code files for further processing.

The package classification phase is a classifier that has been trained on different features. The output of this phase will be a recommended package of the software project that contains the bug. Such step is needed to reduce the time needed to localize the buggy source code files, by first localizing the buggy packages. The time complexity is reduced since the result is a recommended package that would most probably contain the bug. That phase helps the next phase of source code files recommendation in narrowing the search space of source code files to search in.

The source code files recommendation works on recommended package and stack traces for the input bug report if available to recommend files that contain bugs.

To formulate the problem, the symbol *NBR* is used to denote the new bug report. Also, the symbol *RSF* refers to a ranked list of source code files that resulted from our approach. Each software project will be denoted with *SP* and it consists of a list of software packages $PSP \in SP$. Therefore $PSP = \{psp_1, psp_2 \ldots psp_p\}$ where P is the number of software packages. Additionally, a list of source code files will get the notation $PSF \in PSP \in$SP, then PSF $= \{psf_1, psf_2 \ldots psf_F\}$ where F is the number of source code files. Each source code file consists of terms *PSFT* which indicates the classes' names, methods, variables and comments. Another notation which is BRR to indicate a list of old bug reports extracted from the project bug repository. Furthermore the BRR consists of old bug reports, BRR $= \{\mathrm{obr}_1, \mathrm{obr}_{2\ldots}\ \mathrm{obr}_R\}$ where R is the number of bug reports in the system. The unstructured natural text that located in the bug report like bug summary or bug description will be represented with $BRTN \in NBR$ for text in new bug report and *BRTO* for text in the old bug report. *STR* will represent the stack trace located in a bug report either a new or old bug report.

Hence our problem will be formulated as finding a list of ranked source code files *RSF* for a given new bug report *NBR* which contains text NBR and probably a stack trace STR given a list of packages *PSP* for software, *PSF* source code files with its terms *PSFT*. In addition to that a list of old bug reports BRR with its text *BRTO*.

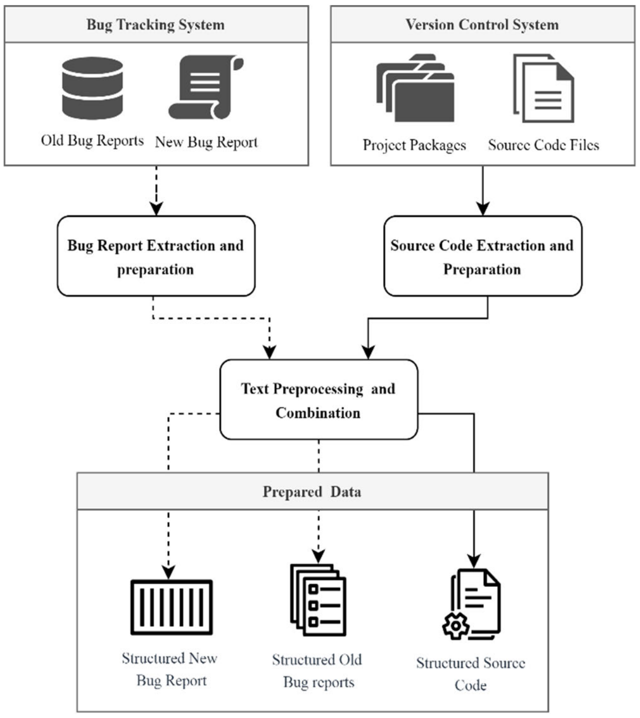
Within the following subsections, we will explain each phase in more details.

A. RAW DATA PREPARATION

In this phase, all inputs to the approach (bug reports and Source code files) will be gathered and preprocessed as shown in Figure 2. The first task is the bug report data extraction and preparations which takes old bug reports as input from the bug tracking system on which the bug reports are published. The second task is the source code files extraction and preparation, by taking the source code files with their different versions from the version control systems as input to put them in a structured form. Afterwards, text processing and text combination task is performed to produce structured bug reports and structured source code files. Such structured information would be the input to the package recommendation phase. We explain each of those three tasks.

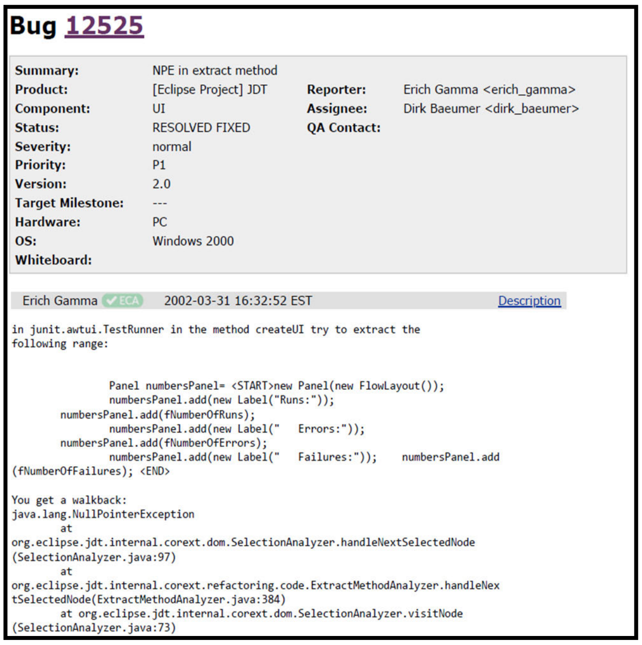
1) BUG REPORTS EXTRACTION AND PREPARATION

Structured and unstructured text information is extracted from the bug report. Project name, project version, data report time, bug report status, and assignee are all structured metadata for the bug report that are already available within specific fields within the bug reports. Figure 3 presents a real bug report extracted from Eclipse bug tracking system for JDT software project. The bug report description is not in an organized form but it is a mix between English text, source code and stack traces. The structured form will separate the English text from the stack traces and from the source code. Such step is needed because each item from the bug report will have special preprocessing for the bug localization process. The unstructured data appears in the bug summary and the bug description. This type of textual data written in the English natural language needs to be in an organized and



Bug Tracking System
Old Bug Reports
New Bug Report
Version Control System
Project Packages
Source Code Files
Bug Report Extraction and preparation
Source Code Extraction and Preparation
Text Preprocessing and Combination
Prepared Data
Structured New Bug Report
Structured Old Bug reports
Structured Source Code

FIGURE 2. Raw Data Preparation Phase.



Bug 12525

Summary:	NPE in extract method		
Product:	[Eclipse Project] JDT	**Reporter:**	Erich Gamma <erich_gamma>
Component:	UI	**Assignee:**	Dirk Baeumer <dirk_baeumer>
Status:	RESOLVED FIXED	**QA Contact:**	
Severity:	normal		
Priority:	P1		
Version:	2.0		
Target Milestone:	---		
Hardware:	PC		
OS:	Windows 2000		
Whiteboard:			

Erich Gamma ✓ ECA 2002-03-31 16:32:52 EST Description

```
in junit.awtui.TestRunner in the method createUI try to extract the
following range:


                Panel numbersPanel= <START>new Panel(new FlowLayout());
                numbersPanel.add(new Label("Runs:"));
        numbersPanel.add(fNumberOfRuns);
                numbersPanel.add(new Label("   Errors:"));
        numbersPanel.add(fNumberOfErrors);
                numbersPanel.add(new Label("   Failures:"));    numbersPanel.add
(fNumberOfFailures); <END>

You get a walkback:
java.lang.NullPointerException
        at
org.eclipse.jdt.internal.corext.dom.SelectionAnalyzer.handleNextSelectedNode
(SelectionAnalyzer.java:97)
        at
org.eclipse.jdt.internal.corext.refactoring.code.ExtractMethodAnalyzer.handleNex
tSelectedNode(ExtractMethodAnalyzer.java:384)
        at org.eclipse.jdt.internal.corext.dom.SelectionAnalyzer.visitNode
(SelectionAnalyzer.java:73)
```

FIGURE 3. A real bug report from JDT project.

TABLE 1. Example of a bug report applying separation by infozilla.

English Natural Text	in junit.awtui.TestRunner in the method createUI try to extract the following range: , You get a walkback:
Stack trace	java.lang.NullPointerException at org.eclipse.jdt.internal.corext.dom.SelectionAnalyzer.handleNextSelectedNode (SelectionAnalyzer.java:97) at org.eclipse.jdt.internal.corext.refactoring.code.ExtractMethodAnalyzer.handleNextSelectedNode(ExtractMethodAnalyzer.java:384) at org.eclipse.jdt.internal.corext.dom.SelectionAnalyzer.visit Node (SelectionAnalyzer.java:73)
Source Code	Panel numbersPanel= <START>new Panel(new FlowLayout()); numbersPanel.add(new Label("Runs:")); numbersPanel.add(fNumberOfRuns); numbersPanel.add(new Label(" Errors:")); numbersPanel.add(fNumberOfErrors);numbersPanel.add(new Label(" Failures:"));numbersPanel.add (fNumberOfFailures);

separated form. The bug description may consist of different types of information like English text, stack traces, and source code files. Such information can all be mixed as one unit of text without any separation within the input bug report. For example, Figure 3 shows an example for such kind of bug reports. The bug description within the bug report in Figure 3, starting from *"in junit.awtui.TestRunner…"* till *"…(SelectionAnalyzer.java:73)"* is composed of three parts: natural text, stack traces, and source code, but those parts are currently one unit of text. Therefore, additional processing is needed to separate such data into three separate parts: text, stack traces, and source code as shown in Table 1. The infozilla tool [16] is applied in our approach to perform the bug report text separation task. Moreover, the stack traces is extracted from the new bug report to be used as an input for the source code classification phase to minimize the search space. However, the stack traces of the old bug reports combined with description English text will be used with old bug report data as a one unit. The reason behind separation is that each separated item will have special further processing to help in finding the bug.

2) SOURCE CODE EXTRACTION AND PREPARATION

First, different software packages and their source code files with different versions are extracted from the version control system of the software project. The preparation phase is then applied by going through different files and putting them in an organized form. The organized form of each source code file is composed of the *project name*, *file name*, *project version*, *file path*, *files' main package*, *commit number,* and *source code text*. After that source code items extraction subtask will take the source code file text as a one unit. Then the output will be the (classes name, method names, identifiers, and code comments) in separated form. The combination subtask will modify the structured form of the old bug reports and source code files by adding the file and package referred to the source code or old bug report.

3) TEXT PREPROCESSING AND COMBINATION

After applying *the bug reports and source code extraction* and *preparation tasks,* that applied by both the bug reports and the source code files are transformed into a structured form. Further preprocessing is needed for these files. First, the text in the bug report passes through natural language processing steps like (Tokenization, Stop words removal, and Stemming) [17]. As a result of this step, we have a bag of words for the bug reports and also source code files. Another subtask is applied here to the source text items which is programming terms splitting. If we have a method name called "*addTwoNumbers*" it will be divided into ["add", "Two", "Numbers"]. This step will enhance the process of text retrieval or matching similarities. Additionally, this sub task will be applied to source code items and comments.

B. PACKAGE CLASSIFICATION PHASE

The output of this phase will mainly be the software package that contains the bug. Such a phase is applied to reduce the needed time for source code files recommendation, and improve the accuracy of the recommended files as well. To reduce the needed time to recommend the source code files that contain the bug, we need to reduce the number of source code files on which the text similarity would be computed. For instance, one text similarity approach [18] had to apply text similarity to more than 8000 files within the JDT dataset in order to locate the source code files to be fixed. To reduce such number of files, our approach will start by recommending one source code package, where such package would have the source code file to be fixed among its files. After locating the recommended package, the text similarity would be applied only to the source code files within that package. For example, if our approach recommends the package "AST view" within the JDT dataset, we would need to run the text similarity analysis only on the 7 files present within such package, instead of the 8000 files present within the JDT dataset. Our approach is expected to improve the accuracy of recommendation. Some source code files or methods may take the same name in different packages, hence limiting the search within the recommended package, will discard such files from the similarity matching, and hence increase the accuracy of source code files recommendation.

Another important advantage of the proposed approach is the ability to work under different circumstances. When the input data for a software project is just source code files, our approach will work. In addition, the presence of old bug reports are also treated with our approach. Source code files and bug reports represents input training record to classify a package first in this phase. Therefore, if the old bug reports is not available, the approach will find the buggy source code files by the utilization of available source code files that are labelled with its package.

The package classification phase as shown in Figure 4 consists of three main tasks: *pre-processing the data* for the BERT model, *applying the BERT model*, and *fine tuning the BERT model*.

BERT is a bidirectional encoder representation of different transformers [15]. The main operation of BERT comes from learning the context of information. All the prepared structured data from the previous phase is fed into the BERT preprocessing phase to prepare the needed format for the BERT model. Such format includes a feature list. After running the BERT model, real number vector will be received, and input into the BERT fine tuning phase. The output of such phase will be a trained model for the recommending the software package. Such trained model can then receive a new bug report as an input, and recommend the package including that bug report. We will discuss each of the above mentioned tasks in the following subsections.

1) BERT PREPROCESSING

BERT preprocessing task will consist of three main subtasks. Which is data labelling, tokenization and padding, and finally numbering as shown in Figure 4.

The Data Labelling subtask will label each training record with its software package. The training records will be in different forms: old structured bug reports, structured source code files. To enhance the process of training, the old structured bug reports will be concatenated to their solved source code files, and will be treated as training records as well. Such concatenation will help the training model to capture different contexts of bug reports terms that appear in different source code files. Each concatenation will be counted as a training record labeled with the classified software package. Moreover, each source code file in different versions of the project will be fed the training model labeled also with its package.

Tokenization and Padding subtask is applied to each of labelled records from the data labelling subtask. The BERT tokenizer [18] will first split the data into small chunks. The padding step is enforced by imposing the size of any input training record to a fixed size for all records.

Numbering subtask is applied using the vocabulary of the training data. Each term on the training data will be have a unique id. Thus, the terms of each record will be converted into numbers to be the input for the BERT model.

2) BERT MODEL PROCESS

The bert model architecture consists of about 12 layers that consist of self-attention mechanisms [19]. The role of self-

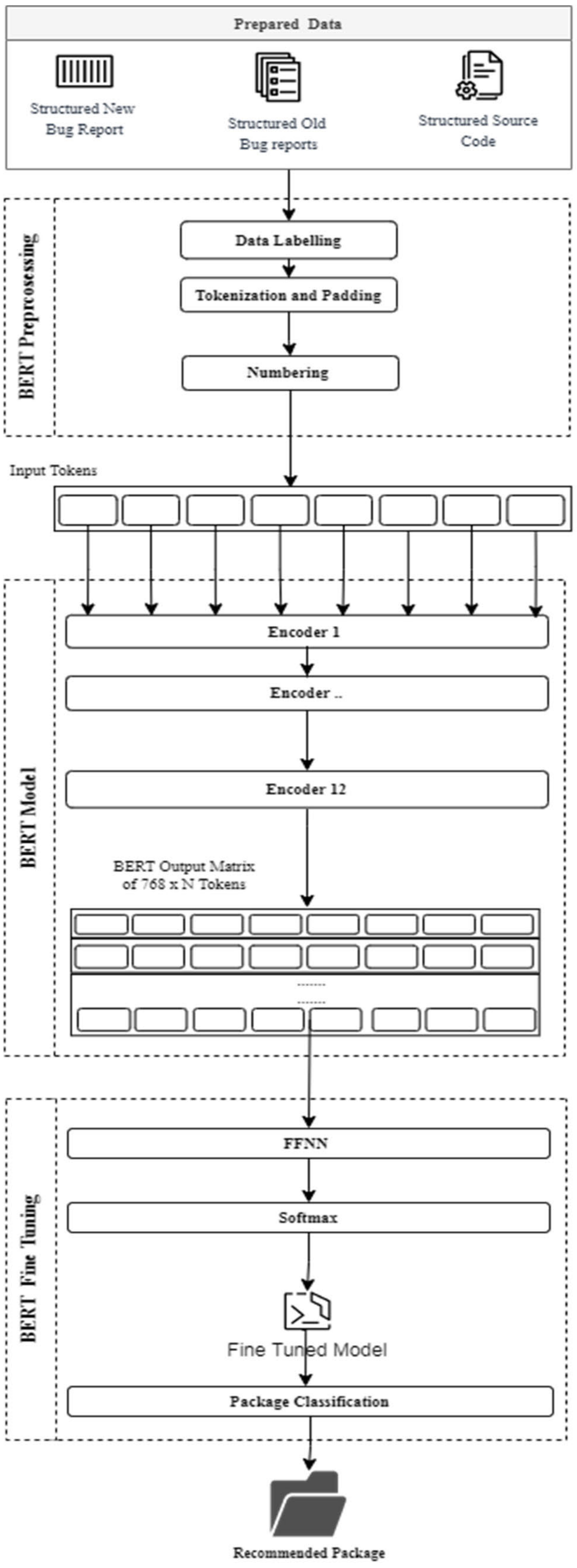


FIGURE 4. Package Classification Phase.

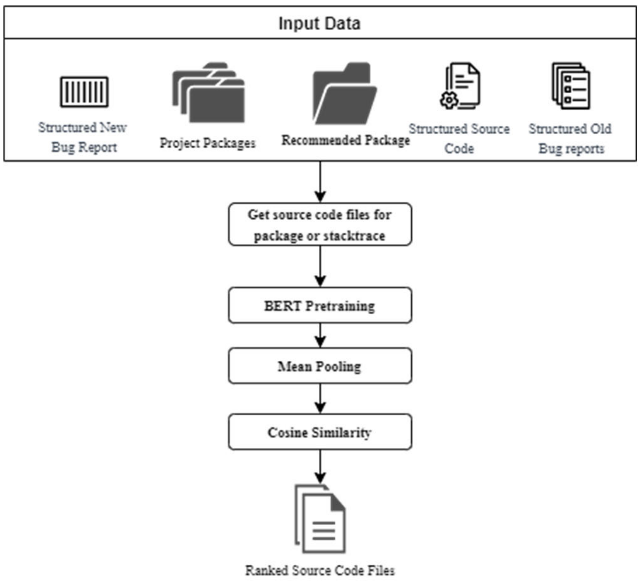


FIGURE 5. **Source code recommendation Phase.**

attention is to enhance each token embedding input by getting different contexts for each token.

Although the directions between different layers of bert is an indication of the consideration of the bert to different bidirectional contexts of each term. The output of the model will be a 768 dimension vector for each token in each training record which represents the sentence embedding.

3) BERT FINE TUNING

The sentence embedding of the bug reports and source code files attached with its package classification will be the input of the fine tuning task. The training data will enter another layer which is the feed-forward neural networks [20] with the softmax function. Each training record will enter the network with its labeled package. Accordingly, the trained output model will be utilized by entering a structured new bug report sentence embedding to classify the new bug report to a specific software package for the next phase.

C. SOURCE CODE RECOMMENDATION PHASE

After the package classification is finished, we got the recommended package that will contain the bug. If the new bug report contains a stack trace, the files in this stack trace will be taken into consideration as well. Figure 5 shows the steps in this phase. Subsequently, the recommended package source files and past related bugs to the specific package will be input into this task. Moreover, a text similarity technique which is sentence BERT [21] will take the new bug report structured text to get the most similar source code file and if attached with old bugs. The output of the source code recommendation phase will be a list of source code files sorted in descending order according to their similarity to new bug report that needs to be fixed.

Sentence embedding's usage is important for the sentence encoding of the input text. In this task, another fine-tuning using BERT applied here for direct text similarity. The text similarity is applied between the new bug report text and all files related to the software package recommended by the previous phase. According to [21] sentence BERT is applied to fine-tune BERT architecture for semantic similarity. As in Figure 5, the input from the source code files and related data from the software package will be entered into the pre-trained BERT. The input will be a pair of records to train the BERT for getting similarities between them. After that, the model will be trained with the text for the embedding.

The Next Step is utilizing the max pooling layer that is located on top of the BERT layer. The role of pooling is to combine each token vector for an input record. Besides, it gives a fixed sentence for the two output sentences. After the two sentences are vectorized, the similarity between these two vectors will be calculated using cosine similarity which is applied as a metric to get the angle [22] between the new bug report and all source code files and data related to a specific package. So the highest score of these results will represent the nearest source code file to the new bug report data.

III. EXPERIMENTS AND EVALUATION

A. RESEARCH QUESTIONS

To evaluate the performance of our Phase-based bug localization approach, the following four research question will be addressed:

RQ1 What is the effect of our approach on bug localization**?**

RQ2 Can the utilization of stack traces affect our approach results?

RQ3 Does the package recommendation phase affect the bug localization time and accuracy?

RQ4 Does our approach outperform state-of-the-art bug localization approaches?

B. EXPERIMENTAL DATASET

The datasets used for our experiments are JDT[1] and Eclipse platform UI[2] dataset which are well known benchmarks for bug localization. The bug localization. JDT stands for Java Development Tools for Eclipse which is an open-source project written in Java language constructed in 2001. Eclipse platform UI contains a set of frameworks in addition to common services for Eclipse. For both datasets, all their source code files across different versions were extracted from Git version control system. Moreover, the projects' bug reports are available on the BUGZILLA bug tracking system. The JDT dataset bug reports that are collected consist of 6274 bug reports and more than 8000 source code files with 17 main packages across more than 12 versions. The Eclipse platform UI dataset consists of more than 6400 bug reports and more than 3400 source code files across more than 14 versions.

C. EXPERIMENT DESIGN AND PROCEDURE

To answer our research questions, we had to pick a dataset that has the needed inputs for our approach: old bug reports from the bug tracking system, and different versions of source code files from version control systems with their packages.

To answer our research questions, we will apply our approach to the JDT and Eclipse datasets mentioned above. For each dataset, we will use 90% of the dataset as training data, and the other 10% as testing data. Hence, some of the previously fixed bugs will be used to assess if our approach will properly recommend their relevant source code files that need to be fixed.

To assess the accuracy of the approach across all the research questions, each dataset will be divided 10 folds cross-validation as applied in some past experiments [23], [24]. The 10 folds will be divided into 9 folds for training and one is assumed to be unfixed for testing. This process will be repeated 10 times as each time one of the 10 folds would be used for testing., and the remaining folds would be for training. After that, the average of the results of all folds is calculated for Top-1, 5, 10, 20 measures.

For RQ1, our approach will be applied to the two datasets. For RQ2, we will apply our approach twice: once with stack traces, and once without stack traces, and compare their accuracy. For RQ3, we will apply our approach twice; once while using the package recommendation phase, and once without using the package recommendation phase, and compare their accuracy and time. For RQ4, we will compare our approach's results against other state-of-the-art approaches that utilized the same dataset, and used the same metrics that we report.

D. EXPERIMENTS EVALUATION METRICS

The experiment performance will be measured using three important evaluation metrics [24] which are TOP-N and MRR as applied by [23].

TOP-N rank metric counts the number of tested bug reports where at least one of the source code files appears in the top-ranked (1 or 5 or 10) files. So, we consider that the tested bug will be successfully localized if at least one of the retrieved source code files is one of the files that were actually fixed within that bug. The true counts for all tested bug reports are counted. Then the true counts will be divided by the total number of bug reports. Therefore, the high the score the high performance of our approach.

Mean Reciprocal Rank (MRR) calculates the inverse position of the source files that contains the bug. The importance of that metric is to measure the retrieved ranked list quality. Equation (1) represents how the MRR is calculated where the symbol Q stands for the bug reports set and rank_i indicates the position of the retrieved files.

$$MRR = \frac{1}{Q}\sum_{i\mathcal{D}0}^{Q}\frac{1}{rank_i} \tag{1}$$

Precision metric is the ratio of correctly predicted positive classes to all items that predicted to be positive as stated in equation 2. Precision tells us how correct is our model, or precise positive predictions. The metric will be utilized only for package classification experiment.

$$\frac{TP}{TP+FP} \tag{2}$$

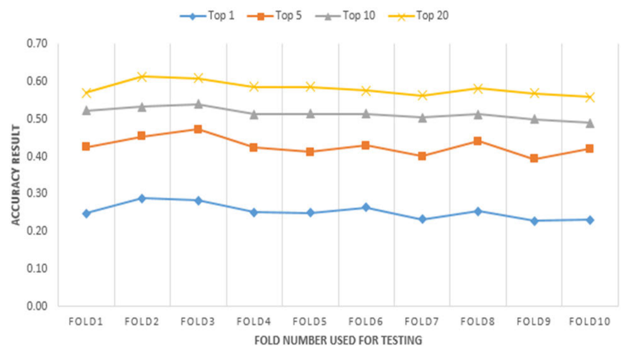


FIGURE 6. Accuracy results of Top-N rank among 10 folds for JDT Dataset for applying our approach.

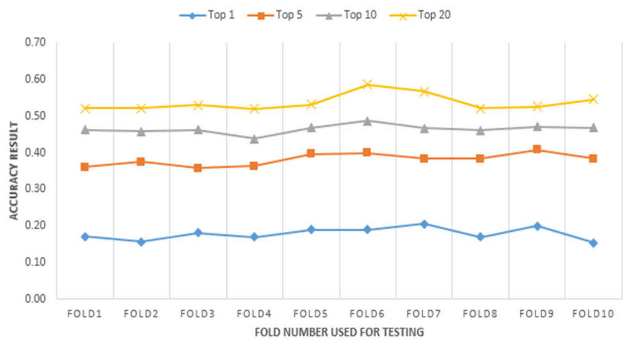


FIGURE 7. Accuracy results of Top-N rank among 10 folds for Eclipse Platform Dataset for applying our approach.

E. EXPERIMENTS RESULTS AND DISCUSSION

Answer to RQ1: What is the effect of our approach on the bug localization?

We applied our approach to the JDT and Eclipse Platform datasets to measure the bug localization performance. The MRR got an average of 0.29 for JDT dataset and 0.27 for Eclipse Platform. Figures 6 and 7 present the Top-N results for each fold to JDT and Eclipse Platform datasets. As stated above that each fold from the 10 folds will be used for testing and the remaining folds for training. Each fold is presented on x-axis and the accuracy results for each fold according to the Top 1, Top 5, Top 10, and Top 20 metrics are presented on the y-axis. Figures 6 and 7 indicate that the results are near among the 10 files, whih implies that our approach's performance is stable for both datasets.

Answer to RQ2: Can the utilization of stack traces affect our approach results?

We applied our approach while utilizing stack traces data on both JDT and Eclipse platform datasets. Within Figures 8

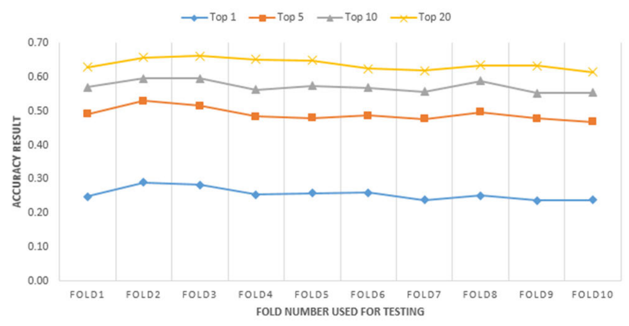


FIGURE 8. **Accuracy results of Top-N rank among 10 folds for JDT Dataset for applying our approach using stack traces.**

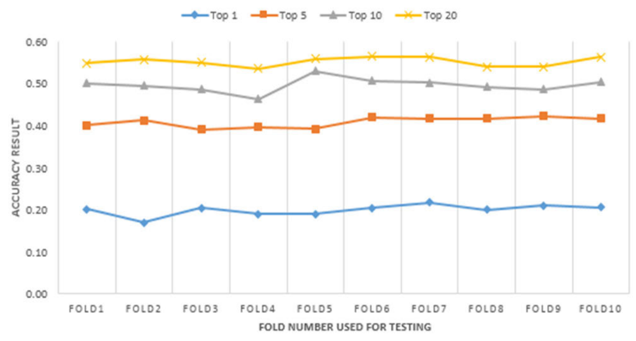


FIGURE 9. **Accuracy results of Top-N rank among 10 folds for JDT Dataset for applying our approach using stack traces.**

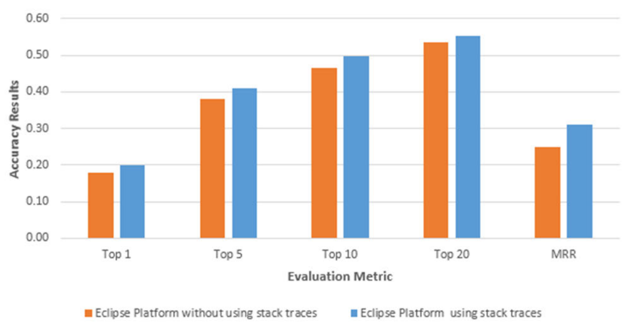


FIGURE 10. **Comparing Results between not utilizing stack traces and utilizing it for JDT Dataset.**

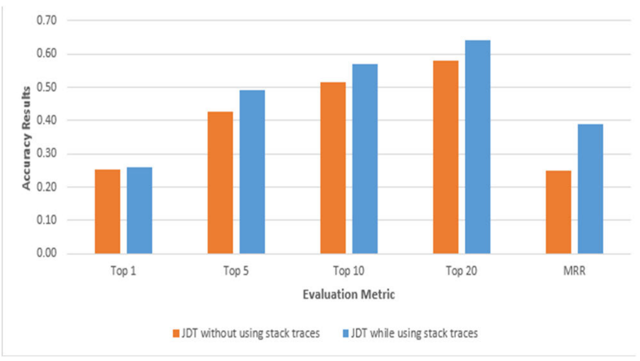


FIGURE 11. **Comparing Results between not utilizing stack traces and utilizing it for Eclipse Platform Dataset.**

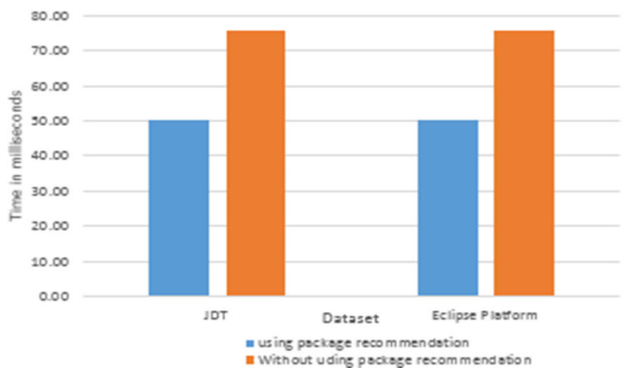


FIGURE 12. **Time results according to utilizing package recommendation phase versus not utilizing.**

and 9, the results of the experiment according to top-n rank for each fold are presented, where each fold is presented on x-axis and the accuracy results for each fold according to the Top 1, Top 5, Top 10, and Top 20 metrics are presented on the y-axis. MRR also achieved a result of 0.33 and 0.31 for JDT and Eclipse Platform respectively.

To assess whether using stack traces would improve the accuracy or not, we applied our approach without using stack traces on both datasets, and compared those results with our previous results when we were utilizing stack traces. Figures 10 and 11 show the difference in results while utilizing stack traces versus while not utilizing them. The results indicate that using stack traces enhanced the accuracy for both datasets across the Top n ranks and MRR metrics.

Answer to RQ3: Does the package recommendation phase affect bug localization time and accuracy?

To answer this question, we have to test the effect of software package recommendation phase we compared the time and accuracy of our approach while applying the package recommendation phase, versus without applying that phase.

Considering the time aspect: Figure 12 shows a bar chart representing the time taken by our approach while applying the package recommendation phase, compared to the time taken by our approach without that phase to locate a list of buggy files. Figure 12 explicitly shows the importance and positive effect of utilizing the package recommendation phase in our approach considering time aspect.

Considering the accuracy aspect: Figure 13 and Figure 14 show the accuracy results when removing the package recommendation phase for both JDT and Eclipse Platform among to 10 folds Top-1, Top-5, Top10, and Top-20 among the 10 Folds for JDT and Eclipse platform datasets.

We compare the accuracy when applying the package recommendation phase versus when removing that phase within Figures 15 and 16 for the JDT and Eclipse Platform datasets respectively. Figure 15 shows that values among Top-1, 5, 10, 20 are (0.19, 0.39, 0.49, 0.54 and 0.31) decreased by

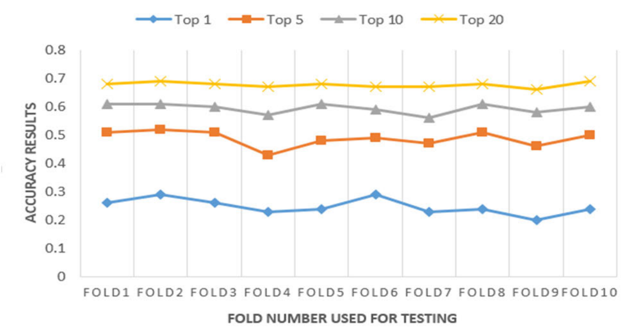


FIGURE 13. Accuracy results for Top-N rank among 10 folds without using package classification for JDT dataset.

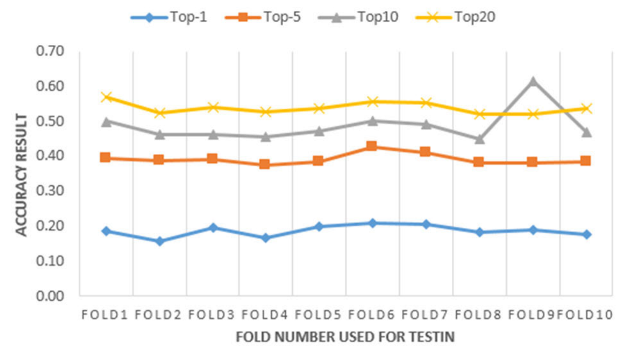


FIGURE 14. Accuracy results for Top-N rank among 10 folds without using package classification for Eclipse Platform Dataset.

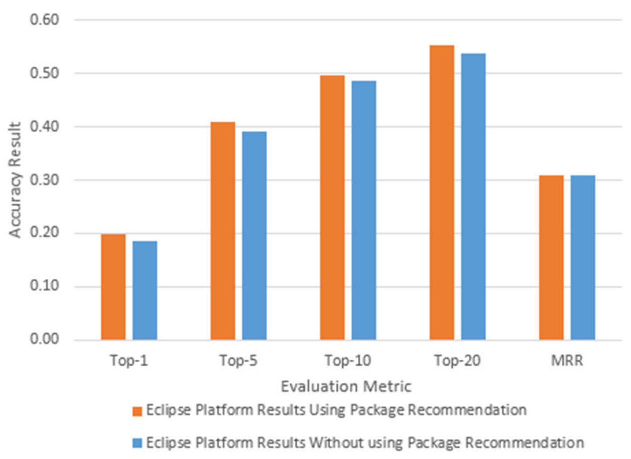


FIGURE 15. Comparing accuracy results using package recommendation versus not utilizing according to Top-N rank among 10 folds according to Eclipse Platform.

approximately 0.01 for removing package recommendation phase versus not removing. However the MRR remains the same which implies that the not utilizing package recommendation phase affected our approach negatively. Figure 16 presents the results applied on JDT dataset, and the results show that Top 1 (0.25) accuracy measure decreased by 0.02, Top 5(0.49) and MRR remains the same. The conducted experiments imply that utilizing the package recommendation phase has a great role in improving the time of our approach, and does not hinder the performance. Also, to the best of our knowledge, no other research works on the layer of package classification.

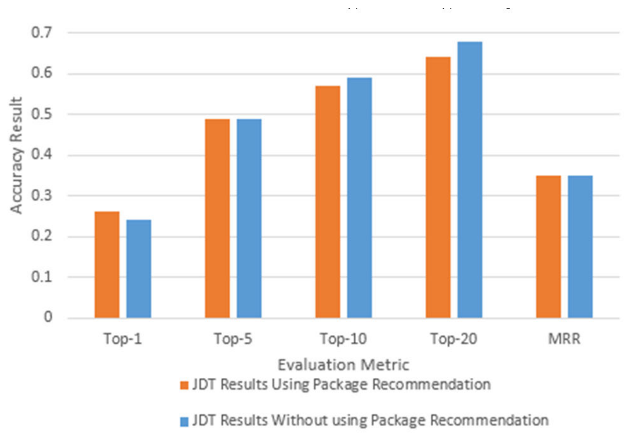


FIGURE 16. Comparing results between accuracy results using package recommendation versus not utilizing according to Top-N rank among 10 folds according to JDT.

Answer to RQ4: Does our approach outperform other bug localization approaches?

To evaluate our approach's effectiveness, we have to evaluate our work according to two important aspects which are time and accuracy against state-of-the-art bug localization approaches. However some of such approaches report the time and accuracy and some of them report accuracy only as follows:

(1) BugLocator [4]: An information retrieval technique used to localize bugs using rVSM for source code and bug reports representation.
(2) NP@hypheCNN [10]: A machine learning technique that utilizes feature from bug reports and source code files. Additionally, a convolution neural networks used for localizing bugs.
(3) STMLocator+ [23]: A supervised topic modelling approach is proposed. The terms in bug reports and source code files is utilized. Also long source code files are phenomena are considered.
(4) cFlow [25]: An approach that learns the features of source codes and bug reports using graph neural networks. A flow based procedure is applied to get the program structure for further learning.
(5) Fast Change [26]:An approach is applied to find the lexical gap between bug reports and source code files
(6) JINGO [27]: An approach that utilizes topic modelling to match co change information in source code files with bug reports by emphasizing topic spaces.

Considering **Top-5** evaluation metric, our approach outperforms all of the above mentioned state of the art approaches that present this evaluation metric which are 5 approaches. Figure 17 shows a comparison between our approach, and the above mentioned approaches in terms of the Top-5 rank,

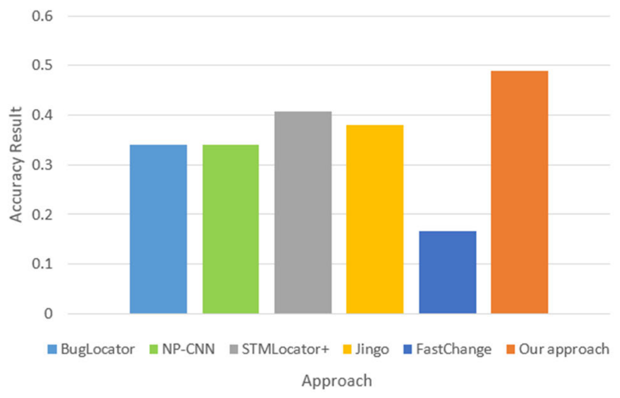


FIGURE 17. Top-5 comparing results between our approach and other state of the art approaches.

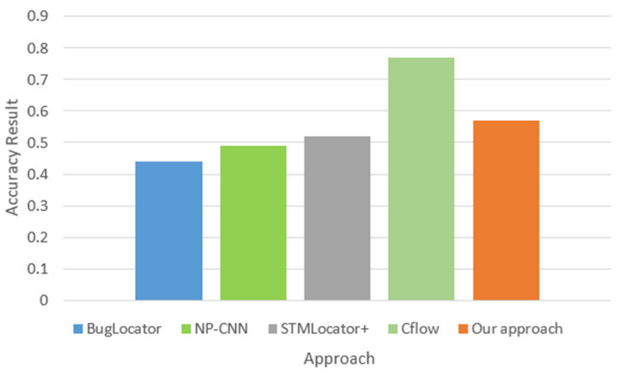


FIGURE 18. Top-10 comparing results between our approach and other state of the art approaches.

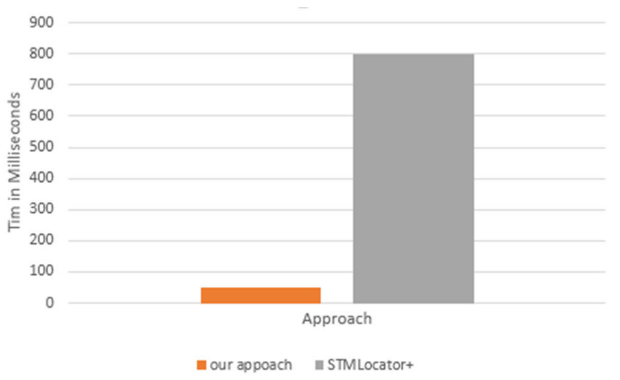


FIGURE 19. MRR comparing results between our approach and other state of the art approaches.

where our approach outperforms all those approaches in that metric.

Considering the Top-10 evaluation metric, Figure 18 compares our approach, and the above mentioned approaches in terms of the Top-10 rank. Figure 18 shows that our phase based approach outperforms three of the approaches. Additionally, our approach got 0.57 and cFlow got 0.77. However cFlow applied their approach on only 5016 bug report from 6274 in JDT dataset.

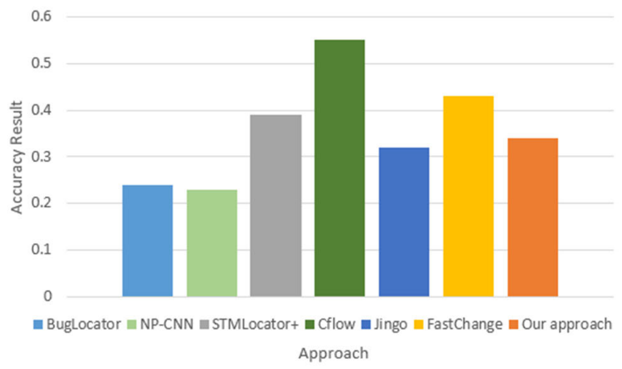


FIGURE 20. Time compared results between our approach and other state of the art approach.

Considering MRR evaluation metric, our approach outperforms all of the mentioned state of the art techniques that utilize this evaluation metric. Figure 19 shows that our phase based approach outperforms three of the approaches by improving BugLocator (0.24) by 10%, NP-CNN (0.23) by 11% and Jingo (0.32) by 2%. Additionally, our approach got 0.34 and cFlow got 0.55. However cFlow applied their approach on only 5016 bug report from 6274 in JDT dataset.

Considering the Time evaluation metric, our work exceeds STMLocator [23] as such technique present the time to find a buggy source code file contains the buggy file. Figure 20 shows that our work recommends the buggy source code file within 50.4 milliseconds for JDT dataset, whereas STMLocator locates the buggy file within 800 milliseconds hence, our approach achieved a time reduction of around reduction of time of about 750 milliseconds.

IV. THREATS TO VALIDITY

In our approach, we apply our work on benchmark datasets as JDT and Eclipse Platform. However, the JDT and Eclipse Platform is an open-source project written in Java language. Hence, we do not know how our approach results would hold for projects written in different programming languages, specifically non object oriented ones. However, our approach steps could generalize across other programming languages that utilizes the same object oriented concepts and the same structural relationships of the Java programming language.

Additional threats to validity is the effect on the number of source code files within a specific package, on the package recommendation phase. In some projects, the software packages could contain source code files with different counts. For instance, one package could contain 600 files, while another package could contain 30 files. Such variance could result in an unbalanced dataset for training. Subsequently, the performance of our approach may be affected.

Another threat to validity is our reliance on the MRR and Top N ranks only. However, we picked those metrics due to their wide usage across various state-of-the-art approaches in the literature. However, other metrics should be considered in our future work

V. RELATED WORK

A. INFORMATION RETRIEVAL APPROACHES

A proposed technique called FineLocator [6] aims to predict bugs at the method level. Their work depends on the syntax trees of the source code. Hence the bug reports and source code artifacts were utilized. A Technique called BLIA indicates bug localziation using integrated analysis [5]. Such technique constructed a score of retrieval of the source code that contains the bug. However, their score depends on the similarity measure between the source code and bug reports.

Another approrach called bug locator [4] worked on the text simlarity between bug report and source code files. Yaojing [7] proposed another technique that takes into consideration the history of fixes for each source code file. Also, the co-occurrence of source code terms with each other among different source code files is utilized. After that a topic modelling technique applied on different benchmark datasets to validate their work. Such technique [7] was evaluated using 10 folds cross validation on the JDT, and PDE platforms. Fan Fang et al [28] attempted to enhance the performance of the information retrieval technique. Such technique targeted the bug report text quality to be informative or not. Another investigation to the combination of different IR technique [29] like VSM and LSI, the authors concluded that the combination results in an improvement results. An IR technique called BoostNShift [30] works with localizing the bug at method level. The authors trying to insert bug report text as a query with the source code to retrieve methods that contains bugs.

In [31], the authors produced an approach for localizing the bugs automatically using ranking. The Cosine similarity is used to measure the similarity between source code files and bug reports. Also, the API information is used in addition to the collaborative filtering enhance the features. Additionally, the names of classes and the frequency of the bug appearing considered to be featured. The average accuracy of 70% achieved all over the top 10 ranked files for 5 benchmark datasets. However, some of the above-mentioned techniques suffers from lexical mismatch between bug reports and source code files which leads to misclassification.

B. MACHINE LEARNING APPROACHES

A proposed tool called Bug2Commit [32] applied the fast text [33] as a word embedding for retrieval of source code files, metadata and stack traces are utilized besides the bug reports. Such technique main difference that approach achieved a good accuracy for a dataset which is not a benchmark but the nature of the software project is different Facebook application. A mathematical model are created by [34] based on past work [4]. The main aim of the mathematical model is to help in enhancing the past technique applied in [4] accuracy with the new solved bug reports. Moreover reducing the time of model by 77.7%.

An integrated model [24] is applied between word embedding and deep neural networks to localize bugs. The rVSM role is to capture the similarity between the source code and bug reports. Additionally the skip gram word embedding model is utilized to get the semantic similarity between bug reports and source code files. The deep neural networks get these features to get the source code files that contains the bug. The authors evaluated their work on 5 known benchmark datasets like SWT, AspectJ, Eclipse and JDT and performing good results. A technique called smell ware based [35]. The code smells which may be results from design issue or problem in source code. The code smells is utilized here to enhance the process of IR based. A proposed approach called DRAST [36] aims at enhancing the generality of the bug localization so it can supports different languages. Random forest and deep learning in addition to a vector space model. This approach achieves an enhancement according to MRR and MAP of about 90%.

Another technique called Blesser [37] that get the source abstract syntax trees and code embedding. They tested their work on 5 C++ source code projects. A two phase's model are used to predict the bugs in the fixed model [13]. The machine learning approach is applied for the bug reports to be predictable or not. The bug report summary, metadata, reporter and bug report description are utilized as features. Then in phase two, the predicted bug report to be fixed for the multi-class classification model. The average accuracy achieved for predicting files is 70 percent.

A system proposed [2] by using a deep learning system to localize bugs. The text terms of Bug reports are utilized in addition to the terms of source code files. The works are evaluated on four datasets (AspectJ, SWT, JDT, and Tomcat) with enhanced accuracy. A topic modeling approach applied to localize bugs in JINGO [27] by utilization of bug reports and source codes. A genetic algorithm is applied in [38] for enhancement of text retrieval. The algorithm trying to find the optimal query to retrieve the true results from the source code files. However, some of above-mentioned techniques would not work if the software project is a new one that does not contain any historical bug reports or any source code projects changes.

VI. CONCLUSION

In this paper, a phase based bug localization is presented to overcome some appeared limitations in previous techniques. Raw data preparation phase role is to prepare different artifacts like bug reports and source code files for next phases. Package classification phase utilizes BERT to capture different contexts of text in different source code files. Such a phase recommends the package that is most probably contains the bug. Finally, the source file recommendation phase produces the buggy source code files.

Our phase based bug localization improves the bug localization time compared to other state-of-the-art techniques. Furthermore, the package classification phase narrows the search space of the code files to search in. Several empirical evaluations were conducted to assess the accuracy of our approach, and the effect of the utilized artifacts. The results show that utilizing the recommendation phase improves the accuracy and time for finding bugs. Additionally, the presence of stack traces artifact shows an enhanced results in the bug localization process versus not.

REFERENCES

[1] I. Sommerville, *Software Engineering*, 9th ed. London, U.K.: Pearson, 2011.

[2] H. Liang, L. Sun, M. Wang, and Y. Yang, "Deep learning with customized abstract syntax tree for bug localization," *IEEE Access*, vol. 7, pp. 116309–116320, 2019.

[3] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei, "A survey on bug-report analysis," *Sci. China Inf. Sci.*, vol. 58, no. 2, pp. 1–24, 2015.

[4] 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 *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 14–24.

[5] K. C. Youm, J. Ahn, and E. Lee, "Improved bug localization based on code change histories and bug reports," *Inf. Softw. Technol.*, vol. 82, pp. 177–192, Feb. 2017.

[6] W. Zhang, Z. Li, Q. Wang, and J. Li, "FineLocator: A novel approach to method-level fine-grained bug localization by query expansion," *Inf. Softw. Technol.*, vol. 110, pp. 121–135, Jun. 2019.

[7] Y. Wang, Y. Yao, H. Tong, X. Huo, M. Li, F. Xu, and J. Lu, "Bug localization via supervised topic modeling," in *Proc. IEEE Int. Conf. Data Mining (ICDM)*, Nov. 2018, pp. 607–616.

[8] H. Xuan, T. Ferdian, L. Ming, L. David, and S. Shu-Ting, "Deep transfer bug localization," *IEEE Trans. Softw. Eng.*, vol. 47, no. 7, pp. 1368–1380, Jul. 2021.

[9] Y. Xiao, J. Keung, Q. Mi, and K. E. Bennin, "Bug localization with semantic and structural features using convolutional neural network and cascade forest," in *Proc. 22nd Int. Conf. Eval. Assessment Softw. Eng.*, Jun. 2018, pp. 101–111.

[10] H. Xuan, L. Ming, and Z. Zhi-Hua, "Learning unified features from natural and programming languages for locating buggy source code," in *Proc. IJCAI*, 2016, pp. 1–7.

[11] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2014, pp. 689–699.

[12] N. An, T. Anh, A. Hoan, and N. Tien, "Combining deep learning with information retrieval to localize buggy files for bug reports (N)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 476–481.

[13] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? A two-phase recommendation model," *IEEE Trans. Softw. Eng.*, vol. 39, no. 11, pp. 1597–1610, Nov. 2013.

[14] S. Qaiser and R. Ali, "Text mining: Use of TF-IDF to examine the relevance of words to documents," *Int. J. Comput. Appl.*, vol. 181, no. 1, pp. 25–29, Jul. 2018.

[15] S. Das, N. Deb, A. Cortesi, and N. Chaki, "Sentence embedding models for similarity detection of software requirements," *Social Netw. Comput. Sci.*, vol. 2, no. 2, pp. 1–11, Apr. 2021.

[16] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *Proc. Int. Work. Conf. Mining Softw. Repositories*, May 2008, pp. 27–30.

[17] J. Atwan, M. Wedyan, Q. Bsoul, A. Hammadeen, and R. Alturki, "The use of stemming in the Arabic text and its impact on the accuracy of classification," *Sci. Program.*, vol. 2021, pp. 1–9, Nov. 2021.

[18] H. Chouikhi, H. Chniter, and F. Jarray, "Arabic sentiment analysis using BERT model," in *Proc. Int. Conf. Comput. Collective Intell.*, 2021, pp. 621–632.

[19] N. M. Duc Tuan and P. Quang Nhat Minh, "Multimodal fusion with BERT and attention mechanism for fake news detection," in *Proc. RIVF Int. Conf. Comput. Commun. Technol. (RIVF)*, Aug. 2021, pp. 1–6.

[20] J. Xin, R. Tang, Y. Yu, and J. Lin, "BERxiT: Early exiting for BERT with better fine-tuning and extension to regression," in *Proc. 16th Conf. Eur. Chapter Assoc. Comput. Linguistics: Main Volume*, 2021, pp. 1–14.

[21] J. Seo, S. Lee, L. Liu, and W. Choi, "TA-SBERT: Token attention sentence-BERT for improving sentence representation," *IEEE Access*, vol. 10, pp. 39119–39128, 2022.

[22] B. Li and L. Han, "Distance weighted cosine similarity measure for text classification," in *Proc. Int. Conf. Intell. Data Eng. Automated Learn.*, 2013, pp. 611–618.

[23] Y. Wang, Y. Yao, H. Tong, X. Huo, M. Li, F. Xu, and J. Lu, "Enhancing supervised bug localization with metadata and stack-trace," *Knowl. Inf. Syst.*, vol. 62, no. 6, pp. 2461–2484, Jun. 2020.

[24] S. Cheng, X. Yan, and A. A. Khan, "A similarity integration method based information retrieval and word embedding in bug localization," in *Proc. IEEE 20th Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Dec. 2020, pp. 180–187.

[25] Y.-F. Ma and M. Li, "The flowing nature matters: Feature learning from the control flow graph of source code for bug localization," *Mach. Learn.*, vol. 111, no. 3, pp. 853–870, Mar. 2022.

[26] A. Ciborowska and K. Damevski, "Fast changeset-based bug localization with BERT," in *Proc. 44th Int. Conf. Softw. Eng.*, May 2022, pp. 946–957.

[27] A. Ciborowska, M. J. Decker, and K. Damevski, "Online adaptable bug localization for rapidly evolving software," 2022, *arXiv:2203.03544*.

[28] F. Fang, J. Wu, Y. Li, X. Ye, W. Aljedaani, and M. W. Mkaouer, "On the classification of bug reports to improve bug localization," *Soft Comput.*, vol. 25, no. 11, pp. 7307–7323, Jun. 2021.

[29] S. Khatiwada, M. Tushev, and A. Mahmoud, "On combining IR methods to improve bug localization," in *Proc. 28th Int. Conf. Program Comprehension*, Jul. 2020, pp. 252–262.

[30] A. Razzaq, J. Buckley, J. V. Patten, M. Chochlov, and A. R. Sai, "BoostNSift: A query boosting and code sifting technique for method level bug localization," in *Proc. IEEE 21st Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2021, pp. 81–91.

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

[32] V. Murali, L. Gross, R. Qian, and S. Chandra, "Industry-scale IR-based bug localization: A perspective from Facebook," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng., Softw. Eng. Pract. (ICSE-SEIP)*, May 2021, pp. 188–197.

[33] M. Liao, B. Shi, X. Bai, X. Wang, and W. Liu, "TextBoxes: A fast text detector with a single deep neural network," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017.

[34] Z. Yang, J. Shi, S. Wang, and D. Lo, "IncBL: Incremental bug localization," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2021, pp. 1223–1226.

[35] A. Takahashi, N. Sae-Lim, S. Hayashi, and M. Saeki, "An extensive study on smell-aware bug localization," *J. Syst. Softw.*, vol. 178, Aug. 2021, Art. no. 110986.

[36] S. Sangle, S. Muvva, S. Chimalakonda, K. Ponnalagu, and V. Gopalan Venkoparao, "DRAST—A deep learning and AST based approach for bug localization," 2020, *arXiv:2011.03449*.

[37] W. Zou, E. Li, and C. Fang, "BLESER: Bug localization based on enhanced semantic retrieval," 2021, *arXiv:2109.03555*.

[38] S. Wang, D. Lo, and J. Lawall, "Compositional vector space models for improved bug localization," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Sep. 2014, pp. 171–180.

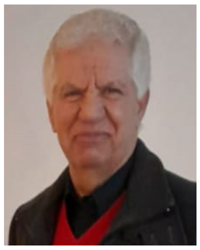
[39] X. Huo and M. Li, "Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code," in *Proc. IJCAI*, 2017, pp. 1909–1915.

[40] C. Sun, X. Qiu, Y. Xu, and X. Huang, "How to fine-tune BERT for text classification?" in *Proc. China Nat. Conf. Chin. Comput. Linguistics*, 2019, pp. 194–206.

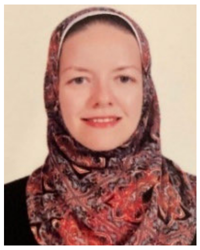
AMR MANSOUR MOHSEN received the B.Sc. degree in computer science from the Faculty of Computers and Artificial Intelligence, Cairo University, and the master's degree in computer science from the Faculty of Computers and Information, Cairo University, in 2016. He has been an Assistant Lecturer with the Computer Science Department, Faculty of Computers and Information Technology, Future University in Egypt, since 2012. His research interests include artificial intelligence, text mining, opinion mining, and software engineering.

HESHAM A. HASSAN is a professor at the faculty of computers and artificial intelligence, Cairo University as the head of the computer science department. He worked as an IT Consultant with the Central Laboratory of Agricultural Expert System, National Agricultural Research Center. He has published over 140 research papers in international journals and conference proceedings. His research interests include knowledge modeling, sharing and reuse, intelligent information retrieval, intelligent tutoring systems, software engineering, cloud computing, and service-oriented architecture (SOA).

KHALED T. WASSIF received the B.Sc. degree (Hons.) in accounting from the Faculty of Commerce, Cairo University, in 1983, the Postgraduate Diploma degree in computer and information science from the Institute of Statistical Studies and Research, Cairo University, in 1986, and the master's and Ph.D. degrees in artificial intelligence from Cairo University, in 1991 and 1998, respectively. He is currently a Professor with the Faculty of Computers and Artificial Intelligence, Cairo University. His research interests include machine learning, data mining, web mining, case-based reasoning, big data, and knowledge engineering. He has supervised or co-supervised 12 students on their Ph.D. dissertations and M.S. theses. He has published 30 research papers in international journals and conference proceedings. He is a reviewer in the international *Egyptian Informatics Journal* and the *Egyptian Computer Journal*.



RAMADAN MOAWAD received the B.Sc. degree in Electric Engineering and the M.Sc. degree in computer engineering from Military Technical College, and the Ph.D. degree in Software Engineering from the ENSAE College, France. He taught several courses in CS and CE in several institutions including the American University in Cairo, the Military Technical College, Cairo University and the Arab Academy for Science and Technology. He joined Future University in 2011 and currently working as Vice-Dean of FCIT. He published over 60 papers in different journals and conferences locally and internationally. His research interests include software engineering and software quality assurance. He has reviewed several papers in IEEE Transactions in *Software Engineering Journal* and many other international and national journals.



SOHA H. MAKADY received the B.Sc. and M.Sc. degrees (Hons.) in computer science from the Faculty of Computers and Information, Cairo University, Egypt, in 2002 and 2005, respectively, and the Ph.D. degree in software engineering from the University of Calgary, Canada, in 2015. She is currently an Assistant Professor with the Faculty of Computers and Artificial Intelligence, Cairo University. She has supervised two M.Sc. students and one Ph.D. student. She is currently supervising three M.Sc. and Ph.D. students. She has ten refereed research papers in international journals and conference proceedings. Her research interests include software evolution, software architecture, and software testing.