2021

# Code Complexity and Version History for Enhancing Hybrid Bug Localization

Ahmed Ali Seyam
*Helwan University*

Abeer hamdy
*The British University in Egypt*, abeer.hamdy@bue.edu.eg

Marwa Farhan
*Helwan University*, marwa.salah@bue.edu.eg

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received March 24, 2021, accepted April 14, 2021, date of publication April 20, 2021, date of current version April 28, 2021.

*Digital Object Identifier 10.1109/ACCESS.2021.3074266*

# Code Complexity and Version History for Enhancing Hybrid Bug Localization

**AHMED ALI SEYAM[1], ABEER HAMDY[2], AND MARWA SALAH FARHAN[2,3]**

[1]The Higher Institute of Computer Science and Information Technology, El Shorouk 11566, Egypt
[2]Faculty of Informatics and Computer Science, The British University in Egypt, Cairo 11837, Egypt
[3]Department of Information Systems, Faculty of Computers and Artificial Intelligence, Helwan University, Cairo 11111, Egypt

Corresponding author: Marwa Salah Farhan (marwa.salah@bue.edu.eg)

**ABSTRACT** Software projects are not void from bugs when they are released, so the developers keep receiving bug reports that describe technical issues. The process of identifying the buggy code files that correspond to the submitted bug reports is called bug localization. Automating the bug localization process can speed up bug fixing and improve the productivity of the developers, especially with a large number of submitted bug reports. Several automatic bug localization approaches were proposed in the literature reviews which are based on the textual and /or semantic similarity among the bug reports and the source code files. Nevertheless, none of the previous approaches made use of the source code complexity despite its importance; as high complexity source code files have higher probabilities to be modified than the low complexity files and are prone to bug occurrences. To improve the accuracy of the automatic bug localization task, this paper proposes a Hybrid Bug Localization approach (HBL) that makes full use of textual and semantic features of source code files, previously fixed bug reports, in addition to the source code complexity and version history properties. The effectiveness of the proposed approach was assessed using three open-source Java projects, ZXing, SWT, and AspectJ, of different sizes. Experimental results showed that the proposed approach outperforms several state-of-the-art approaches in terms of the mean average precision (MAP) and the mean reciprocal rank (MRR) metrics.

**INDEX TERMS** Bug localization, text mining, information retrieval, version history, code complexity, textual similarity.

## I. INTRODUCTION

Software projects are usually released while they still contain bugs; so they set up Bug Tracking Systems (BTS) to receive bug reports and manage bug fixes during the maintenance phase [1]. Bug localization is an essential task during the maintenance phase to locate the buggy files. It is a time-consuming task to be done manually, as the developers should analyze the submitted bug reports and review a large number of source code files. Bug reports are not standardized; they are written using natural language. Sometimes the bug reports contain terms from the associated buggy files. Also, bug reports may contain stack traces [2]. The existence of stack traces in the report is critical as it helps the developer to narrow down the list of suspicious source code files [3]. Moreover, newly submitted bug reports may have textual similarity to previously fixed ones, so comparing the newly submitted reports with the previously fixed ones will help to locate bugs faster. Many approaches were presented for bug localization using Information Retrieval (IR) techniques [4]; where the submitted bug report was treated as a query, to retrieve the top N similar source files sorted according to their similarity score [2]. However, Bug localization based on Textual Similarity is not enough to detect suspicious files. Recent researches focus on exploring additional features that increase the accuracy of bug localization [5]. Hybrid approaches for bug localization have been proposed in the literature, to enhance the performance of locating buggy files. Hybrid approaches consider extra features when calculating the similarity scores between the source code files and bug reports [5]. These features include: (i) semantic similarity between the source code and the bug reports, (ii) textual and/or semantic similarity between the newly submitted bug report and previous reports, (iii) Source code version history,

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

and (iv) stack traces. However, none of the previous hybrid approaches include all these features together. The work proposed by Gharibi *et al.* [15] is considered the most comprehensive hybrid approach, but it does not include the source code version history feature despite its importance. As source files that were modified are likely to contain bugs [6]. Furthermore, most source code files that are modified are getting more complex. As developers usually do the modifications under strict deadlines to avoid the critical cost. So, they do not take into consideration the software design principles and the clean code guidelines; which leads to the occurrence of code smells, including complex code. High code complexity does not indicate the existence of a bug but it indicates a violation of one or more of the software design principles and may lead to the occurrence of bugs [7]. Including the complexities of source code files in the hybrid bug localization approach, may enhance its performance. To our knowledge, none of the previous approaches considered the code complexity feature for enhancing the bug localization process.

### A. RESEARCH OBJECTIVES

The main objective of this paper is to improve the performance of the automatic bug localization task, to increase the productivity of the developers, and to enhance the software quality during the maintenance phase. The paper proposes a hybrid bug localization approach (HBL) that leverages the following extracted features: 1) Textual and semantic similarity features between a newly submitted bug report and the source code files, taking into consideration stack traces in the bug reports, 2) Similarity between the newly submitted bug report and the previously fixed reports, 3) Token matching between bug reports and source code files. The tokens include file names, class names, and method names, 4) The cyclomatic complexity of the source code files, and 5) The version history of the source code files in the (VCS).

The remainder of the paper is structured as follows: Section II briefs the related work in automating the bug localization. Section III presents the architecture of the proposed HBL approach. Section IV discusses the experiments and the results. Finally, the conclusion and future work will be presented in Section V.

## II. RELATED WORK

Information Retrieval (IR) is a widely used technique to automate the bug localization process.''BugLocator'' is one of the efficient early approaches which were proposed by Zhou *et al.* [2]. Buglocator utilized the revised Vector Space Model (rVSM) for representing the bug reports and the source code files. The localization process is based on measuring the textual similarity between the new bug report and each of the source code files and the previously fixed reports. The experiments were performed on four open-source projects with their associated bug reports (e.g. AspectJ, Eclipse, SWT, and ZXing). The results show that BugLocator effectively locates buggy files. For example, it can locate up to 80% of bugs in the Eclipse project and 60 % bugs in the AspectJ project.

Saha *et al.* [8] proposed Bug Localization Using Structure Information Retrieval (BLUiR) where essential terms were extracted from the Abstract Syntax Tree (AST) of the source code such as class names, variables names, methods names, and comments. Also, the description and summary of each bug report were tokenized. BLUiR used Okapi BM25 to calculate the similarities between the bug reports and source code files, The experimental results showed that BLUiR is superior to BugLocator and other bug localization tools [10], [11].

Wong *et al.* [11] proposed ''BRTracer'' which divided the source code into several segments, then measured the textual similarity between each code segment and each bug report. Also, they measured the similarity between the stack traces included in the bug reports and the code segments. BRTracer was assessed over three software projects (e.g. AspectJ, Eclipse, and SWT). The results showed that the performance of BRTracer outperforms the BugLocator even when BRTracer discards the similarity with previous bug reports.

Wang *et al.* [6] proposed the ''Amalgam'' framework for locating buggy files using the version history, similar reports, and the structure of (BLUiR). The experimental results over Four software projects (e.g. AspectJ, Eclipse, SWT, and ZXing) showed an improvement up to 24.4 % in terms of mean average precision (MAP) in comparison to BugLocator and 16.4 % in comparison to BLUiR [2], [8].

Rahman *et al.* [12] proposed an approach for bug localization using the version history component by tracking the source files that frequency changed as it has a higher chance to contain bugs. The approach has modified the (rVSM) of Zhou *et al.* [8] by combining it with the score of the frequency of past fixed source code files to get (MrVSM). The results show on large-scale implementation of three open-source projects (e.g. SWT, ZXing, and Guava) improve 7 % in terms of Mean Reciprocal Rank (MRR) and up to 8 % for (MAP) compared with techniques of (BLUiR), (BugLocator), and (Amalgam) [2], [6], [8].

Wang *et al.* [13] presented an updated version of the Amalgam approach called (Amalgam+) Which locates buggy source code files given a set of bug reports. The proposed approach has been utilizing five features components (e.g. version history component, similar bug report component, Structure component, Stack trace component, and reporter information component). The result of (Amalgam+) over the four open projects (e.g. AspectJ, Eclipse, SWT, and ZXing) show an improvement in term of (MAP) by 6.0%over (Amalgam). Also, compared with state-of-the-art approaches [7], [10], [11], [13].

Zhou *et al.* [14] presented an approach that leverages the feature of part-of-speech in bug reports and the relationship between source code files to enhance the performance of bug localization. The results show over six open source projects (e.g. AspectJ, SWT, ZXing, Eclipse, Birt, and Tomcat ) can

improve the accuracy for all those projects in comparison with (BugLocator) and other techniques of existing bug localization approaches [7], [10], [11], [13].

Youm *et al.* [15] presented Bug Localization using Integrated Analysis (BLIA v1.0). It is a file-level bug localization approach that utilizes the textual properties between the bug reports and source code files. In addition to, the stack traces in the bug reports, the structure information of the source files and code change history features. (BLIA v1.0) was evaluated over three open-source projects (e.g. AspectJ, SWT, and Zxing); and it was found that BLIA v1.0 outperforms some baseline approaches [2], [6], [8], [11] in the terms of (MAP) and (MRR).

Youm *et al.* [16] presented an amended version of (BLIA v1.0) called (BLIA v1.5) which can detect each of the buggy source code files and source code methods. The proposed approach integrated hybrid features from both source code and bug report files. Those features include texts, stack traces, and developer comments from bug reports. Also, structure information and version history were extracted from source code files. BLIA v1.5 was evaluated over three open-source projects (e.g. AspectJ, SWT, and ZXing). The results show that BLIA v1.5 outperforms baseline approaches in the terms of (MAP) and (MRR) [2], [6], [8], [11], [15].

Gharibi *et al.* [17] presented a multi-component bug localization approach which is sometimes referred to as a hybrid bug localization approach. Hybrid approaches leverage various features of the bug reports and the source code such as syntactic and semantic textual features, stack traces to narrow down the search space of source files. The results show the performance of the proposed approach that applied on three open-source projects (e.g. AspectJ, SWT, and Zxing ) can locate appropriate buggy source code files up to 52 % by recommending one source code file and up to 78 % by recommending top ten source code files compared to state-of-the-art approaches [7], [10], [11], [13], [14].

Swe and Oo [18] presented a bug localization approach based on information retrieval. The proposed approach has three components: Structure source code component, fixed bug report component, and combining the score component. The results show the performance of the proposed approach that was applied on three open-source projects (e.g. SWT, AspectJ, and Eclipse) are 42.7 %, 15.6%, and 23.8% in terms of (MAP). Also, in terms of (MRR), the proposed approach achieves 52.4 %, 27.1%, and 33% respectively.

One of the recent studies that tackle the bug localization problem was conducted on the method level instead of the file level. Zhang *et al.* [19] proposed an approach called (FineLocator) that implements bug localization at the method level by using the features of call dependency, semantic similarity, and temporal proximity. The experimental results on the (ArgoUML, Maven, AspectJ, Ant, and Kylin) show that FineLocator can improve the performance of bug localization at the method level at the average by 20%, 21%, and 17% at Top-N, MAP, and MRR respectively.

Wang *et al.* [3] used a supervised topic modeling approach for bug localization. They utilized five features which are the history of bug fixing, terms that appear in bug reports multiples times in related source files, the size of the source file, meta information of bug reports, and stack traces.

Swe and Oo [18] introduced a learning-to-rank approach that leverages the domain knowledge through different parts of source code files includes API, bug-fixing history component, and code change history component. The proposed approach ranking set of source code files given a set of bug reports as input based on the computing weights of each component. The results show that a learning-to-rank approach outperforms the BugLocator and state-of-the-art approaches. In particular, it can extract correct recommendations within the top 10 of ranked source code files up to 70% of bug reports in the Eclipse and Tomcat open-source projects.

Some recent studies leveraged Deep learning techniques to solve the bug localization problem. Deep learning was used to overcome the lexical gap between the terms in bug reports and source code tokens. Lam *et al.* [21] presented an approach that uses Deep Neural Network (DNN) in combination with rVSM to enhance the accuracy of bug localization. Their experimental results showed that the proposed DNN based approach achieves higher accuracy than BugLocator and another two approaches, using 10 fold cross-validation [18], [22].

Xiao *et al.* [22] proposed a DeepLoc approach, which is based on deep learning techniques that take all advantages of semantic information. DeepLoc is an enhanced convolution neural network (CNN) that combine the features of bug–fixing recency and frequency with word-embedding to enhance the lexical gap of similarities between source code files and bug reports. Deeploc is tested and evaluated on over 18,500 bug reports associated with their open-source projects (e.g. Eclipse, AspectJ, SWT, JDT, and Tomcat. The results show that the proposed approach achieves up to 13.4% higher in the term of (MAP) than CNN. DeepLoc outperforms state-of-the-art approaches [2], [20], [22].

Liang *et al.* [24] presented Customized Abstract Syntax Tree (CAST), which exploits deep learning techniques with customized abstract syntax trees of source programs to locate buggy source files effectively. CAST extracts both lexical-semantic from bug reports and source code files. Moreover, CAST enhances the Tree-Based Convolution Neural Network (TBCNN) with Abstract Syntax Tree (AST). The result shows on widely-used open source projects that CAST outperforms state-of-the-art approaches in detecting buggy source files.

However, deep learning-based approaches can only be utilized with large-scale projects. As the models require training on a large amount of data. While the proposed hybrid bug localization approach could be utilized with small, medium, and large scale projects.

### A. RELATED WORK TECHNIQUES ANALYSIS

Table 1 summarizes the main features utilized by the previous hybrid bug localization techniques. Different techniques

**TABLE 1. Comparison among the features utilized by the previous hybrid bug localization approaches.**

| Approach | POS Tagging | Token Matching | VSM Similarity | Stack Trace | Semantic Similarity | Fixed Bug Report | Version History | Call graph | Code Complexity |
|---|---|---|---|---|---|---|---|---|---|
| Swe et al. [18] | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| BugLocator [2] | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| BLUiR[8] | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| Amalgam [6] | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| BRTracer[11] | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ |
| Rahman et al.[12] | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| AmaLgam+ [13] | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ |
| Zhou et al.[14] | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| BLIA v1.5 [16] | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ |
| Gharibi et al. [17] | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |

combine different features of bug reports and source code files to increase the performance of bug localization. The table lists the previous work according to the performance of each technique from the lowest to the highest performance. It should be noted that all these techniques were experimented with the same datasets and used the same performance metrics. As could be observed the more features included in the approach, the better performance of locating the buggy file(s) given a bug report. The following notes could be observed in Table 1: 1) None of the previous techniques leveraged the existence of the code complexity property, 2) The version history component was utilized by four studies, and the authors showed its effectiveness in locating the bug reports. However, no one combined the feature of source code complexity with modified source code files at the version control system (VCS), 3) Although Gharibi *et al.* [15] have combined most of the features (e.g. Token Matching, VSM similarity, Stack Trace, Semantic Similarity, and Previous Fixed Reports), their approach lacks both of the code complexity and version history properties, which are considered important features as the high complexity files have higher probabilities to be modified than the low complexity files and consequently are bug-prone. The paper proposes a hybrid bug localization approach that makes use of the code complexity and version history properties, in addition to all the features included in the work of Gharibi *et al.* [17].

## III. PROPOSED HYBRID BUG LOCALIZATION APPROACH (HBL)

This section presents our novel hybrid bug localization approach (HBL) that benefits from each of the complexity of the source code files and the version history because of the critical issues that happened when the source code changes frequently and become more complex, in addition to other features such as the similarity between the newly submitted bug reports and the previously fixed ones as the previous fixed files are more likely to produce bugs. Also, both the textual and semantic similarity between the newly submitted bug reports and the source code files because the common tokens between source files and bug reports are indicating the location of bugs in source files. The proposed HBL approach, as depicted by Fig.1, includes three-stages, the parsing& pre-processing stage, the Individual feature scores stage, and the combined scores stage. When a new bug report is received, it is parsed and preprocessed then fed to the individual scores stage. On the other hand, source code files are retrieved from the (VCS), parsed and preprocessed, then fed to the individual feature score stage. Then each component in the individual score stage assigns scores to the suspicious source code files. Finally, the individual scores are combined to generate a list of suspicious source files. The following subsections discuss the proposed solution in detail.

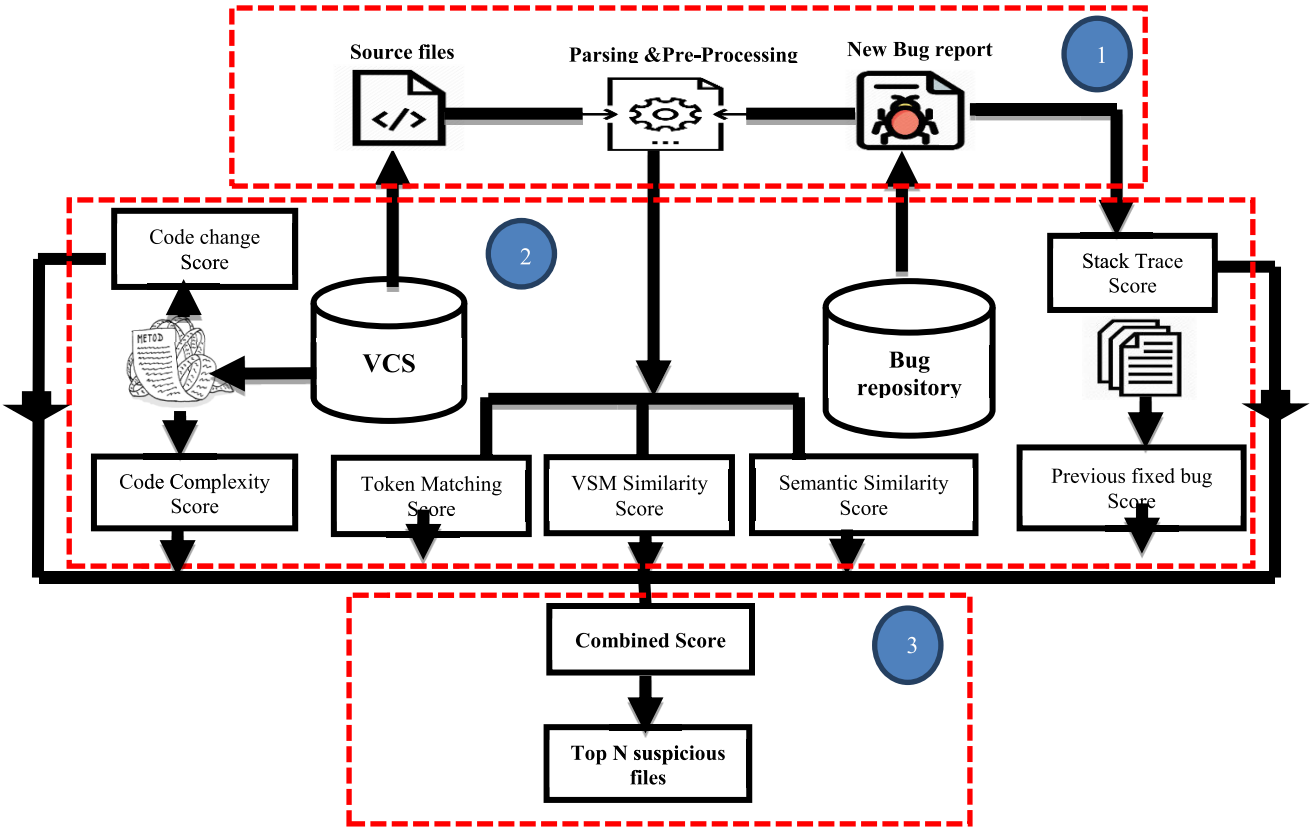


**FIGURE 1.** The overall design of HBL.

### A. PARSING & PRE-PROCESSING STAGE

This is the first stage of the proposed approach. It aims to extract the key tokens from each of the bug reports and the source code and remove the unnecessary tokens for the HBL approach. This stage includes two main steps which are: parsing and pre-processing.

#### 1) PARSING

The source code was parsed to extract the different identifiers such as class names, methods, and variables' names. While, the bug reports were parsed to extract the report summary, description, and open date.

#### 2) PRE-PROCESSING

Natural language processing techniques were applied to each of the source code and the bug reports. Each of the parsed bug reports and the source code was tokenized and stemmed using Porter stemmer. Stemming converts tokens to their base (e.g. "goes" and "gone" are converted into "go"). The Camel Case tokens were split into separate tokens (e.g. "ViewImage" was transferred into "view" and "image"). Furthermore, English stop words were removed such as punctuation marks, numbers, and keywords of programming language. Stemming and stop word removal reduce the number of tokens and remove the noise. Finally, parts of speech (POS) tagger was applied for classifying the tokens into a noun, verb, etc.

### B. INDIVIDUAL FEATURE SCORE STAGE

This stage comes after the parsing and pre-processing stage where both source code files and bug reports have been prepared well. A detailed analysis has been done on the bug reports and source code files to extract each feature from them. Furthermore, give a score for each feature to combine them at the final stage.

#### 1) STACK TRACE SCORE

A bug report is a document that contains all information about an error or critical issues in the software system. Usually, it is submitted by the users of the system. A bug report consists of several fields, but the most important fields are "bug id", "open date", "summary", and "description". Fig.2 is an example of bug reports for the SWT project. The bug report id "75739" that was open in "06-10-2004" contains a problem that was explained in the "summary" and "description" fields, where they share common tokens between them and their associated source code files and methods. Some bug reports contain stack traces, this may help to better localize bug with more accurate and enhance the performance as the stack trace contain the frames that cause the bugs. Therefore,

| Bug ID | 75739 |
|---|---|
| Open date | 2004-10-06 17:02:00 |
| Summary | **Variant** has no **toString**() |
| Description | The **Variant** class has no **toString**() and one cannot call get**String**() in all cases since it throws an Exception if the type is VT_EMPTY. So I suggest: /** * Always returns a **String**. * @param **variant** * @return a **String** */ public static **String toString**() |
| Fixed files | org.eclipse.swt.ole.win32.**Variant**.java |
| Fixed methods | **toString**() |

**FIGURE 2.** Bug report of SWT project without Stack Trace.

the score of stack trace can be calculated as:

$$Stack\_Trace\_Score = \begin{cases} \frac{1}{ranks} & if\ s\ is\ in\ the\ stack\ trace \\ 0 & otherwise \end{cases} \quad (1)$$

$rank_s$ is the score of ranked source files in the stack trace and "0" will be given if the source files not found in the Stack Trace. Fig. 3 is an example of bug reports for the SWT project that contains a Stack Trace.

### 2) PREVIOUS FIXED BUGS SCORE

Previous fixed bugs reports are responsible for produced new bugs as fixed bugs usually try to fix similar source files. Therefore, by using old fixed bug reports, we can find the source code files that cause bugs that are reported in new bug reports. we use the multi-label classification that was proposed by Gharibi *et al.* [17] to get the probability score for relevant source code files for newly bug reports. where the terms of previous fixed bug reports will be an input and their fixed methods will be the labels in the multi-label classification. The previous fixed bug score will be calculated as:

$$Similar_{Score(f_i)} = \sum_{AllS_i Connectedtof_i}^{B} \frac{Similarity\ (B_i, S_i)}{n_i} \quad (2)$$

where $B$ is the new bug received and $S$ is all previous bug report that has been fixed. So the similar score for file $f_i$ calculated by getting all $S_i$ that connected to $f_i$, $n_i$ is the number of files that have been modified to fix $S_i$ [17].

### 3) TOKEN MATCHING SCORE

Token Matching is the process of checking whether a token that belongs to a source code exists among the tokens of a bug report or not. A score is calculated based on counting the number of matched tokens between a bug report and every source code file. A score of zero is given in case of no match between the bug report and a source code file.

### 4) REVISED VECTOR SPACE MODEL (RVSM) SCORE

The vector space model (VSM) has been broadly utilized within the conventional Information Retrieval (IR) field. Most search engines also use similarity measures based on this model to rank Web documents. The model creates a space in which both documents and queries are represented by vectors. The revised Vector Space Model (rVSM) was used in our approach which is proposed by Zhou *et al.* [2]. rVSM represents bug reports and source files as a set of the vector of term weight, then extracts tokens of bug reports from the summary and description. Also, from source files, we extract tokens of the class name, methods name, and noun tokens in comments to build their vectors, the score is calculated by the following equation:

$$w_{i,d} = 1 + log(tf_{i,d}) \times log(\frac{\#src\_files}{df_i}) \quad (3)$$

where $w_{i,d}$ of term $i$ in document d that clarifies the number of appearance term $i$ in document d. $\#src\_files$ is the total source code files in the collection while $df_i$ refer to the document frequency of term $i$ that appear in the document, after that, cosine similarity is been used between bug reports and source code files as:

$$cos\ (b, s) = \frac{\overrightarrow{V}_b \cdot \overrightarrow{V}_s}{|\vec{v}_b| \times |\vec{v}_s|} \quad (4)$$

where $\vec{v}_b$ the vector of term weights for bug reports $b$ and $\vec{v}_s$ are the vector of term weights for source files $s$ while $|\vec{v}_b|$ and $|\vec{v}_s|$ refer to the length of the vectors and calculated as :

$$|\vec{v}_d| = \sqrt{w_{1,d}^2 + w_{2,d}^2 + \ldots + w_{v,d}^2} \quad (5)$$

Then combine the source file's length score with the result of cosine similarity to enhance the result of MAP and MRR according to Zhou *et al.* [8] and calculated as:

$$LenScore_s\ (\#terms) = \frac{1}{1 + e^{-N(\#terms)}} \quad (6)$$

where $lenScore$ for each source code file is calculated by counting the number of terms (i.e. class names, method names, variables, and the comment). Finally, multiplying (4) and (6) we get the final formula of (rVSM) score as:

$$SimilartiyScore = cos\ (b, s) \times LenScore_s\ (\#terms) \quad (7)$$

The $SimilartiyScore$ is calculated for each bug report $b$ according to relevant source files to get similarities between them.

| Bug ID | 77948 |
|---|---|
| Open date | 2004-11-05 09:53:00 |
| Summary | NullPointerExceptionin**CLabel**.**findMnemonic** |
| Description | java.lang.NullPointerException<br>atorg.eclipse.swt.custom.**CLabel.findMnemonic**(**CLabel**.java:194)<br>atorg.eclipse.swt.custom.**CLabel.onMnemonic**(**CLabel**.java:334)<br>atorg.eclipse.swt.custom.**CLabel**$3.keyTraversed**(CLabel**.java:126)at<br>org.eclipse.swt.widgets.TypedListener.handleEvent(TypedListener.java:221)<br>org.eclipse.swt.widgets.EventTable.sendEvent(EventTable.java:82) |
| Fixed files | org.eclipse.swt.custom.**CLabel**.java |
| Fixed methods | initAccessible(), getKeyboardShortcut(),**onMnemonic**() |

**FIGURE 3.** Bug report of SWT project with Stack Trace.

### 5) SEMANTIC SIMILARITY SCORE

The textual similarity is not accurate enough to detect the similarity between a bug report and a source code file due to lexical mismatching; which degrades the performance of locating a bug to relevant source files. So, measuring the semantic similarity between the bug reports and the source files is a necessity. Several approaches were proposed in the literature, that build vector representation to the words. These vectors are called word embedding vectors; which hold the semantic relation between the words.Word2Vec and Glove[1] are the most widely used models for word embedding. Word2Vec uses an artificial neural network model to detect word associations given a large corpus of text. While the glove is utilized the convolutional neural network (CNN), which is one of the architectures of the deep learning models. We used Glove in the proposed approach for generating the embedding vectors related to tokens of each bug report and source code file. Cosine similarity was used to calculate the semantic similarity score. We selected Glove in this work as it is a pre-trained model on a huge corpus, in addition to it is based on deep learning techniques. The Semantic Similarity Score is calculated according to equation (4).

### 6) VERSION HISTORY SCORE

Software projects are frequently subject to change to solve issues or fix bugs. These changes may fix the current issues but may lead to other bugs [25]. Software changes are tracked by a Version Control System (VCS) that keeps all versions of software and theirs commits of changing. We can extract information from a VCS about the commits that have been made, according to the following rules:

- The commit history should contain files that have been changed with a change type that can be added, deleted, modified, or renamed.
- The modification must be done over a specific period of days.

[1] Online: https://spacy.io/usage/vectors-similarity

the commits are analyzed and a score is assigned to the files that have been modified according to equation (8):

$$\boldsymbol{ChangeScore}\,(\boldsymbol{f}, \boldsymbol{k}, \boldsymbol{R}) = \sum_{c \in \boldsymbol{R} \wedge f \in c} \frac{1}{1 + e^{12\left(1 - \left(\frac{k - dc}{k}\right)\right)}} \quad (8)$$

where **R** refers to the relevance of commits for each file **f** that have been done within ***dc***, ***dc*** refers to the consumption time between a relevant commit ***C*** and a bug report during past **k** days [16].

### 7) CODE COMPLEXITY SCORE

Complex source code files indicate a poor software design, due to the violation of fundamental design principles; which degrades the code quality and may cause bugs in the future [26]. Cyclomatic Complexity (CC) is one of the software metrics that is used to measure the complexity of the methods and classes in the source code [7]. Table 2 shows the (CC) values and their corresponding meaning, cost, and effort. As could be observed (CC) values range from 1:10 means that the code is clean and does not suffer any complexity, so the cost and effort of its maintenance are low. While the (CC) values range from 10:20, indicates that the code is quite complex and its maintenance cost and effort is medium. When the (CC) value of 20:40, indicates that the code is very complex and it requires very high effort. IF (CC)

**TABLE 2.** Comparison among the Cyclomatic Complexity numbers and their meaning.

| Cyclomatic Complexity | Status | Cost & Effort |
|---|---|---|
| 1-10 | Source Code is well-structured | Low |
| 10 - 20 | Complex Code | Medium |
| 20 - 40 | Very Complex Code | High |
| More 40 | Not Tested Code | Very. High |

**TABLE 3.** Cyclomatic Complexity and version history properties OF SAMPLE bug reports and their fixed source files from SWT repository.

| Bug ID | Summary | Cyclomatic Complexity | Modified Past Days | Fixed Source File |
|---|---|---|---|---|
| 14654 | [typing] Single line selection on triple-click | 990 | yes | StyledText.java |
| 58185 | Eclipse fails to load old style BMP files | 83 | yes | SWT.java |
| | | 88 | yes | FileFormat.java |
| | | - | no | OS2BMPFileFormat.java |
| | | - | no | WinBMPFileFormat.java |
| 75739 | Variant has no to String | 99 | yes | Variant.java |
| 77948 | NullPointerException in CLabel.findMnemonic | 137 | yes | CLabel.java |
| 78548 | Button Selection fires before MouseUp | 154 | yes | Button.java |
| | | - | no | ToolItem.java |

greater than 40, this means that the code is never passed all test cases [25] and requires code refactoring. In this work, we calculated the (CC) for each source code file that has been modified to solve an issue or fix a bug, then scores were assigned to the files according to equation (9) as follows:

$$\textit{Code\_Complexit\_Score} = \begin{cases} 1 & \textit{if } (CC) \geq 40 \\ 0 & \textit{otherwise} \end{cases} \quad (9)$$

PyDriller[2] was utilized in our experiments to retrieve the modified files and to calculate their Cyclomatic Complexities. Table 3 shows sample bug reports and their fixed source files, retrieved from the SWT repository; as could be observed that the source files that needed fix have high CC values. Moreover, the source files have been modified before submitting the bug report.

### C. COMBINING SCORE STAGE

This is the final stage in our approach where the combined score is calculated through the linear combination of the scores resulting from the feature components using equation (10):

$$\text{Final\_Score} = \sum_{i=1}^{n} wi \times \textit{ScoreStage} \quad (10)$$

where $wi\epsilon$ [0,1] represents the different weights (e.g. $w1, w2, w3, w4, w5, w6, w7$) contributions of the previously discussed seven features in the final score that determines the top n suspicious files.

## IV. EXPERIMENTS AND RESULTS

In this section, firstly we will discuss the dataset that is used in the proposed approach. Also, the evaluation metric. Secondly, the implementation details will be discussed to show the structure of each component in the proposed approach. Finally, the result will be presented to evaluate the proposed approach.

### A. DATASET

In our proposed approach, we use the same benchmark dataset presented by Zhou *et al.* [2] The datasets consist of three open-source projects ZXing,[3] SWT,[4] and AspectJ,[5] and their relevant bug reports. Table 4 shows some statistics related to these projects such as the study period for the bug reports, the number of bug reports, the number of source files, and the number of reports containing stack traces.

**TABLE 4.** Statistics of the benchmark dataset [2].

| Project | Study Period | #Fixed bugs | #source files | #Report with stack trace |
|---|---|---|---|---|
| AspectJ | Jul 2002 – May 2010 | 286 | 6485 | 95 (33.2%) |
| SWT | Oct 2004 – Apr 2010 | 98 | 484 | 4 (4.1%) |
| Zxing | Mar 2010 – Sep 2010 | 20 | 391 | (5 %) |

### B. EVALUATION METRICS

Three evaluation metrics have been used in the literature to assess bug localization systems which are: Top@N, MAP, MRR.

[2]Online: https://pydriller.readthedocs.io/en/latest/intro.html

[3]Online: https://github.com/zxing/zxing

[4]Online: https://www.eclipse.org/swt/

[5]Online: https://www.eclipse.org/aspectj/

1) Top@N

is defined as the number of bug reports whose related source files are ranked in the top N retrieved source files; N was set to equal 1, 5, 10. For a bug report, if the top N source files include at least one related file, we consider it a hit.

2) MEAN RECIPROCAL RANK (MRR)

is a statistic measure for evaluating any system that produces for a query, a list of possible responses ordered by the probability of correctness. The reciprocal rank (RR) of a bug report response is the multiplicative inverse of the rank of the first related source file in the source files: 1 for first place, 0.5 for second place, 0.33 for the third place, and so on. Assume, a set of bug reports BR and the rank of the first relevant source file for bug report $i$ is $firsti$, then MRR is defined as the average of the RR of the results for BR and is given by the equation:

$$MRR = \frac{1}{|BR|}\sum_{i=1}^{|BR|}\frac{1}{firsti} \tag{11}$$

MRR measures the effectiveness of the retrieval process. The higher the MRR value, indicating the better performance of bug localization.

3) MEAN AVERAGE PRECISION (MAP)

Is a metric used in information retrieval applications to evaluate how well are the retrieved results as a response for a query; in our context, the bug report is the query,

MAP for a set of queries is defined as the mean of the average precision scores ($AvgP$) for each query and is calculated using equation 13 as follows:

$$MAP = \sum_{i=1}^{|BR|}\frac{AvgP(i)}{|BR|}, \quad AvgP = \frac{\sum_{k\epsilon S} Prec@k}{m} \tag{12}$$

4) WHERE Prec@k

is the retrieval precision over a set of k source files in a ranked list of $m$ files and it is calculated using equation 14:

$$Prec@k = \frac{\#\ of\ relevant\ source\ files\ in\ top\ k}{k} \tag{13}$$

The higher the MAP, the better the performance of the bug localization system.

### C. SYSTEM TUNING

The version history score, which is calculated using equation 8 includes an important parameter that affects the performance of the bug localization, which is the $k$ parameter. We experimented with different values to the $k$ (50,90,120,150). On the other hand, the y-axis represents the MAP values. Fig. 4 shows the MAP values at the different settings of $k$. As could be observed the maximum value of the MAP is reached at $k = 120$. Fig. 5 shows that the maximum value of the MRR is reached at $k = 120$ too. So, we set the value of $k$ to 120 in all of our experiments.

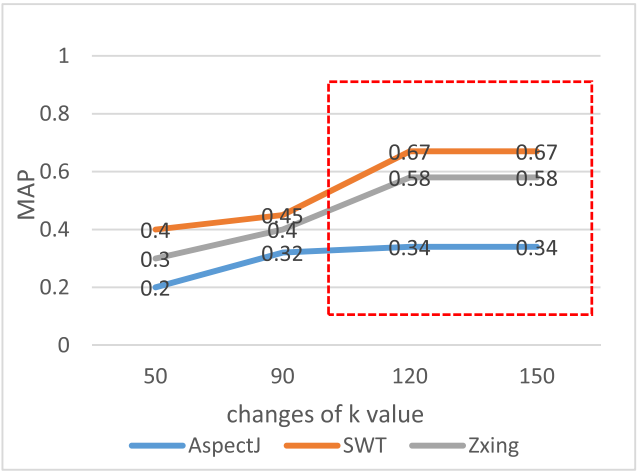


**FIGURE 4.** Impact of the K value on MAP metric.

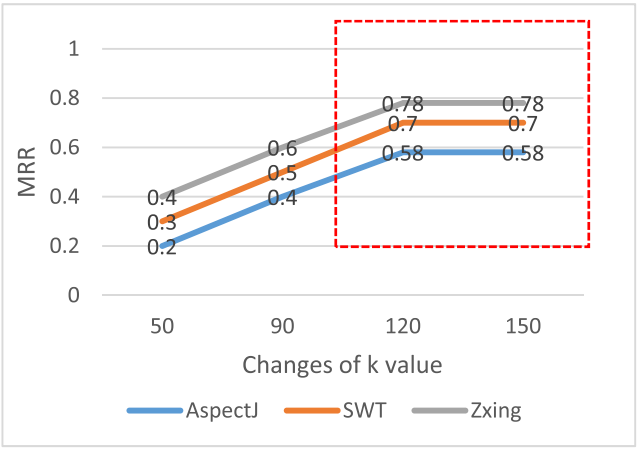


**FIGURE 5.** Impact of the K value on MRR metric.

### D. RESULTS AND DISCUSSION

The following observations were made as a result of experiments conducted by (HBL), Table 5 show statistics related to the three open-source projects. We could observe that in the AspectJ project, there are 84.5 % of total source files suffer from high complexity and modified past days. In the SWT project, there are 96.9 % of total source files suffer

**TABLE 5.** Statistics of the open-source java projects.

| Project | #source files | #Complex & modified source files |
|---|---|---|
| AspectJ | 6485 | 5486 (84.5 %) |
| SWT | 484 | 469 (96.9 %) |
| ZXing | 391 | 201 (51.4) % |

**TABLE 6.** A Comparison between HBL and the other approaches.

| Project | Approach | top@1 | top@5 | top@10 | MRR | MAP |
|---|---|---|---|---|---|---|
| AspectJ | **HBL** | **137 (48%)** | 198 (69.2%) | 218 (76.2%) | **0.58** | **0.34** |
| | Gharibi et al | 133 (46.5%) | 192 (67.1%) | 212 (74.1%) | 0.56 | 0.32 |
| | BLIA v1.5 | 119 (41.5%) | **204 (71.1%)** | **230 (80.6%)** | 0.55 | **0.39** |
| | Amalgam | 127 (44.4%) | 187 (65.4%) | 209 (73.1%) | 0.54 | 0.33 |
| | Rahman et al. | N/A | N/A | N/A | N/A | N/A |
| | BRTracer | 113 (39.5%) | 173 (60.5%) | 197 (68.9%) | 0.49 | 0.29 |
| | BLUiR | 97 (33.9%) | 150 (52.4%) | 176 (61.5%) | 0.43 | 0.25 |
| | BugLocator | 88 (30.8%) | 146 (51.0%) | 170 (59.4%) | 0.41 | 0.22 |
| | Swe et al. | 59 (20.6%) | 95 (33.2%) | 117 (40.9%) | 0.27 | 0.15 |
| SWT | **HBL** | **71(72.4%)** | **88 (89.7%)** | **90 (91.8%)** | **0.78** | **0.67** |
| | Gharibi et al | 67 (68.4%) | 84 (85.7%) | 88 (89.8%) | 0.76 | 0.65 |
| | BLIA v1.5 | 66 (67.3%) | 83 (86.7%) | 88(89.8%) | 0.75 | 0.65 |
| | Amalgam | 61 (62.2%) | 80 (81.6%) | 88 (89.8%) | 0.71 | 0.62 |
| | Rahman et. al. | 47 (48.0%) | 70 (71.4%) | 81 (82.7%) | 0.60 | 0.54 |
| | BRTracer | 46 (46.9%) | 78 (79.6%) | 87 (88.8%) | 0.59 | 0.53 |
| | BLIUiR | 55 (56.1%) | 75 (76.5%) | 86 (87.8%) | 0.66 | 0.58 |
| | BugLocator | 39 (39.8%) | 66 (67.3%) | 80 (81.6%) | 0.53 | 0.45 |
| | Swe et al. | 43 (43.8%) | 63 (64.2%) | 70 (71.4%) | 0.52 | 0.42 |
| ZXing | **HBL** | **14 (70%)** | **17 (85%)** | **18 (90%)** | **0.70** | 0.58 |
| | Gharibi et al | 12 (60.0%) | 15 (75.0%) | 16 (80.0%) | 0.67 | 0.56 |
| | BLIA v1.5 | 11 (55.0%) | 15 (75.0%) | 16 (80.0%) | 0.64 | **0.62** |
| | Amalgam | 8 (40.0%) | 13 (65.0%) | 14 (70.0%) | 0.51 | 0.41 |
| | Rahman et. al. | 9 (45.0%) | 14 (70.0%) | 15 (75.0%) | 0.55 | 0.50 |
| | BRTracer | 10 (50.0%) | 13 (65.0%) | 15 (75.0%) | 0.61 | 0.49 |
| | BLIUiR | 8 (40.0%) | 13 (65.0%) | 14 (70.0%) | 0.49 | 0.39 |
| | BugLocator | 8 (40.0%) | 12 (60.0%) | 14 (70.0%) | 0.50 | 0.44 |
| | Swe et al. | N/A | N/A | N/A | N/A | N/A |

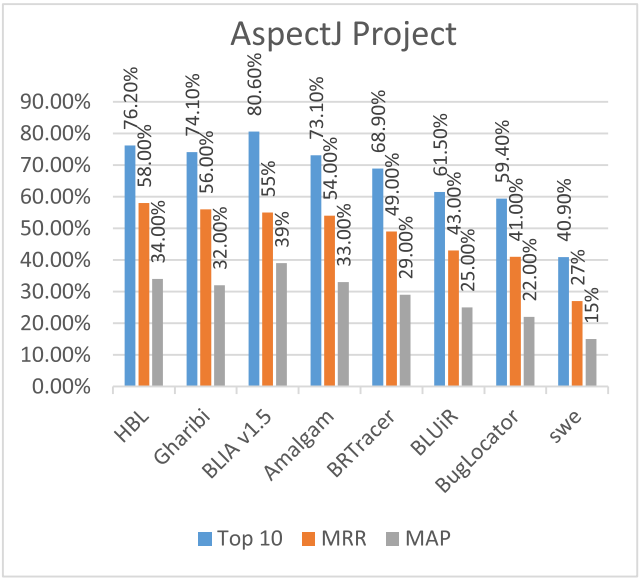


**FIGURE 6.** The comparison result of Top 10, MRR, MAP on AspectJ.

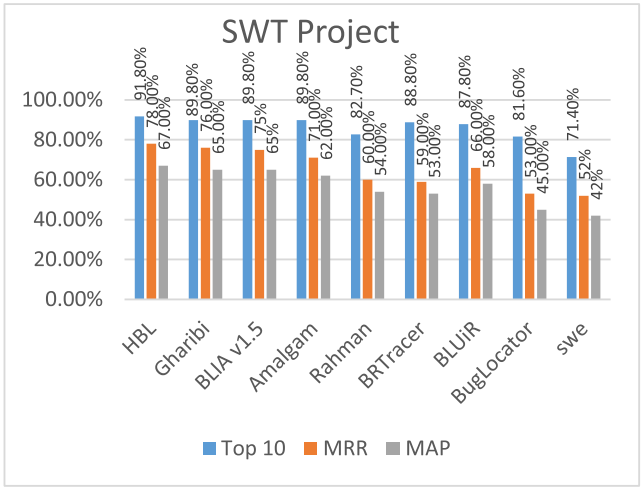


**FIGURE 7.** The comparison result of Top 10, MRR, MAP on SWT.

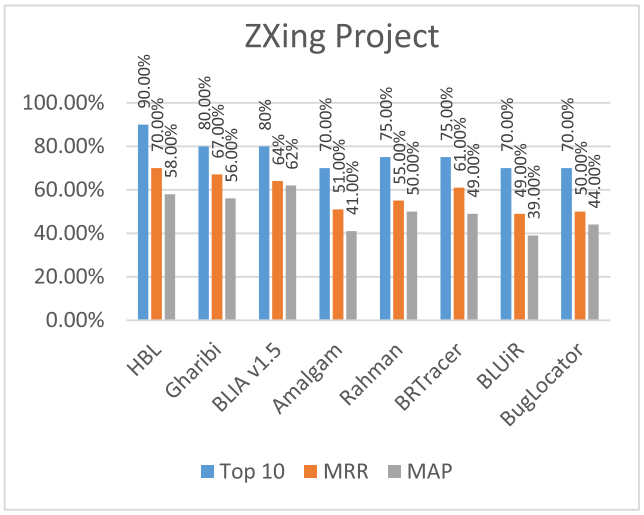


**FIGURE 8.** The comparison result of Top 10, MRR, MAP on ZXing.

from high complexity and modified past days. In the ZXing project, there are 51.4 % of total source files suffer from high complexity and modified past days. As shown in Fig. 6 that explains the comparison of different results among different approaches in the AspectJ project. (HBL) recorded the highest values in terms of the metric of top 10, MRR, and MAP except for the BLIA's v1.5 MAP, top@5, and top@10. Because AspectJ includes low complex source files. Also, the number of modified files past days before submitting the bug report not huge compared to the SWT project. Although BLIA v1.5 outperforms in MAP metric, it requires more time for running because BLIA v1.5 run in two phases, the first phase generates a list of suspicious files, the second phase take the list of suspicious files as input and applying textual similarity between methods in ranked files & newly submitted bug report to rank suspicious methods. (HBL) has archived (76.2 %) in top@10, (58%) in MRR, and (34%) in MAP. Fig. 7 presented the results of different approaches in the SWT project, (HBL) also recorded the highest value among different approaches. As we could observe, the results in SWT achieved the highest among all datasets because 96.9% of the project includes high complexity source files and have been modified past days before submitting a new bug report. (HBL) has achieved (91.8%) in top@10, (78%) in MRR, and (67%) in MAP. Fig.8 presents the results of different approaches in the ZXing project, (HBL) also recorded the highest value among different approaches except for BLIA v1.5's MAP, it outperforms in the MAP metric because 51.4 % of ZXing project only suffer from high complexity and have been modified past days. (HBL) has achieved (90%) in top@10, (70%) in MRR, and (58%) in MAP. The proposed approach has exceeded the different approaches in all evaluation metrics because the utilization of source code analysis that has been used as (HBL) is the only approach that used the feature of code complexity as feature selection that makes improves different approaches. Also, it has used the property of version control system (VCS) which was missed at Gharibi *et al.* [17]. Table 6 summarizes all results for the three projects (e.g. AspectJ, SWT, and ZXing) with all evaluation metrics (e.g.Top@1, Top@5, Top@10, MRR, and MAP), The bold values in Table 6 mean the highest values of each given metric, as shown the proposed approach outperforms the different approaches (e.g. Gharibi *et al.* [17], BLIA v1.5 [16], Amalgam [6], Rahman *et al.* [12], BRTracer [11], BLUiR [8], BugLocator [2], Swe and Oo [18]). In the three open projects of Swe and Oo [18] achieved the lowest performance as it has used two feature selections only (e.g. VSM similarity and Fixed bug report). BugLocator [2] showed a significant outperform over Swe and Oo [18] by enhancing the VSM with a revised Vector Space Model (rVSM) that increases the accuracy of bug localization. BLUiR [8] showed a significant

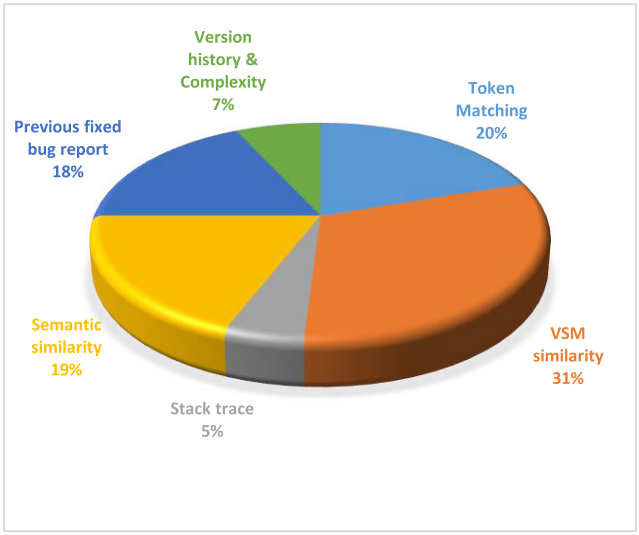


**FIGURE 9.** The average performance of each feature in (HBL).

improvement over Swe and Oo [18], and BugLocator [2] as it enhances the performance of bug localization because it depends on the structure information that is based on code structure. BRTracer [11] outperforms the previous approach (e.g. Swe and Oo [18], BugLocator [2], and BLUiR [8]) as the feature of stack trace added beside the VSM similarity and fixed bug reports. Also, Amalgam [6] outperforms the previous approaches (e.g. BugLocator, BLUiR, and BRTracer) due to the feature of version history that makes significant improvement and achieved high value in all evaluation metrics. BLIA v1.5 [16] has improved the performance by implementing the bug localization on file level and method level. Furthermore, it has added the comments of bug reports that added more information about the bugs. Gharibi *et al.* [17] have improved the performance among the previous approaches because of adding extra property (e.g. POS tagging, Token matching, VSM similarity, Stack Trace, Semantic similarity, and Fixed bug report). (HBL) has achieved the highest performance due to adding the features of source code analysis that merge the property of version history and complex files that indicate that source files are not clean code and suffer from high complexity. Fig. 9 shows the average performance of each separate feature on (HBL) in the metric of Top@10 at all three open projects. VSM similarity achieved (31%) that considered the highest value affected the performance because the common tokens between bug reports and source files have a higher score. Token matching achieved (20%) as it calculates exact matching tokens between bug reports and source code files. Semantic similarity achieved (19%) as it overcomes the problem of lexical mismatching that make the process of bug localization is a tedious task. Previous Fixed bug report plays an important role in enhancing the performance, it has achieved (18%) as the previous fixed bug report may be responsible for producing the same bugs in the future. Version History & Complexity has been achieved (7%), it merges the property of version history and code complexity together which affects the performance of (HBL). Finally, stack trace archived (5%) which considers the lowest value as not all bug reports contain stack traces, so this feature depends on the quality of bug reports.

## V. CONCLUSION AND FUTURE WORK

The software maintenance phase is crucial for the software projects to fix issues and bugs, that might arise after the software release. During the maintenance phase, developers were assigned a large number of bugs that should be fixed as fast as possible to enhance the software quality. The paper proposed a methodology for automating the bug localization process to increase the developer's productivity. The proposed hybrid bug localization approach (HBL) automatically locates the buggy source code files given a set of bug reports. The proposed HBL leverages the textual and semantic features of the source code, previously fixed reports, and the newly submitted ones; in addition to the source code complexity and version history properties. HBL includes three stages; the first stage is the parsing and pre-processing of the source code files and bug reports. the preprocessed source code files and bug reports are passed to the second stage to calculate the individual scores according to the different extracted features. Finally, the third stage combines each feature score to get the final score and generates a list of suspicious files for each source code file. The effectiveness of the proposed approach was assessed using three open-source Java projects, ZXing, SWT, and AspectJ. Experimental results showed that the proposed approach outperforms several state-of-the-art approaches in terms of the Mean Average Precision (MAP) and the Mean Reciprocal (MRR) metrics. Furthermore, the proposed approach is suitable for large-scale and small-scale projects.

This work could be extended such that the buggy methods could be located the same as locating the buggy files.

## REFERENCES

[1] S. A. Akbar and A. C. Kak, "A large-scale comparative evaluation of IR-based tools for bug localization," in *Proc. 17th Int. Conf. Mining Softw. Repositories*, Jun. 2020, pp. 21–31, doi: 10.1145/3379597.3387474.
[2] 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, doi: 10.1109/ICSE.2012.6227210.
[3] 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, doi: 10.1007/s10115-019-01426-2.
[4] Tamanna and O. P. Sangwan, "Review of text mining techniques for software bug localization," in *Proc. 9th Int. Conf. Cloud Comput., Data Sci. Eng. (Confluence)*, Jan. 2019, pp. 208–211, doi: 10.1109/CONFLUENCE.2019.8776959.
[5] Z. Shi, J. Keung, K. E. Bennin, and X. Zhang, "Comparing learning to rank techniques in hybrid bug localization," *Appl. Soft Comput.*, vol. 62, pp. 636–648, Jan. 2018, doi: 10.1016/j.asoc.2017.10.048.
[6] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proc. 22nd Int. Conf. Program Comprehension (ICPC)*, 2014, pp. 53–63, doi: 10.1145/2597008.2597148.

[7] C. R. Gujar, "Use and analysis on cyclomatic complexity in software development," *Int. J. Comput. Appl. Technol. Res.*, vol. 8, no. 5, pp. 153–156, Apr. 2019, doi: 10.7753/IJCATR0805.1002.
[8] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2013, pp. 345–355, doi: 10.1109/ASE.2013.6693093.
[9] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Source code retrieval for bug localization using latent Dirichlet allocation," in *Proc. 15th Work. Conf. Reverse Eng.*, Oct. 2008, pp. 155–164, doi: 10.1109/WCRE.2008.33.
[10] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Trans. Softw. Eng.*, vol. 33, no. 6, pp. 420–432, Jun. 2007, doi: 10.1109/TSE.2007.1016.
[11] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Sep. 2014, pp. 181–190, doi: 10.1109/ICSME.2014.40.
[12] S. Rahman, K. K. Ganguly, and K. Sakib, "An improved bug localization using structured information retrieval and version history," in *Proc. 18th Int. Conf. Comput. Inf. Technol. (ICCIT)*, Dec. 2015, pp. 190–195, doi: 10.1109/ICCITechn.2015.7488066.
[13] S. Wang and D. Lo, "AmaLgam+: Composing rich information sources for accurate bug localization: Composing rich information sources for accurate bug localization," *J. Softw., Evol. Process*, vol. 28, no. 10, pp. 921–942, Oct. 2016, doi: 10.1002/smr.1801.
[14] Y. Zhou, Y. Tong, T. Chen, and J. Han, "Augmenting bug localization with part-of-speech and invocation," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 27, no. 6, pp. 925–949, Aug. 2017, doi: 10.1142/S0218194017500346.
[15] K. C. Youm, J. Ahn, J. Kim, and E. Lee, "Bug localization based on code change histories and bug reports," in *Proc. Asia–Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2015, pp. 190–197, doi: 10.1109/APSEC.2015.23.
[16] 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, doi: 10.1016/j.infsof.2016.11.002.
[17] R. Gharibi, A. H. Rasekh, M. H. Sadreddini, and S. M. Fakhrahmad, "Leveraging textual properties of bug reports to localize relevant source files," *Inf. Process. Manage.*, vol. 54, no. 6, pp. 1058–1076, Nov. 2018, doi: 10.1016/j.ipm.2018.07.004.
[18] K. E. E. Swe and H. M. Oo, "Source code retrieval for bug localization using bug report," in *Proc. IEEE 15th Int. Conf. Intell. Comput. Commun. Process. (ICCP)*, Sep. 2019, pp. 241–247, doi: 10.1109/ICCP48234.2019.8959535.
[19] 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, doi: 10.1016/j.infsof.2019.03.001.
[20] 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. (FSE)*, 2014, pp. 689–699, doi: 10.1145/2635868.2635874.
[21] 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, doi: 10.1109/ICPC.2017.24.
[22] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi, "Improving bug localization with word embedding and enhanced convolutional neural networks," *Inf. Softw. Technol.*, vol. 105, pp. 17–29, Jan. 2019, doi: 10.1016/j.infsof.2018.08.002.
[23] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proc. 38th Int. Conf. Softw. Eng.*, May 2016, pp. 404–415, doi: 10.1145/2884781.2884862.
[24] 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.
[25] A. Cairo, G. Carneiro, and M. Monteiro, "The impact of code smells on software bugs: A systematic literature review," *Information*, vol. 9, no. 11, p. 273, Nov. 2018, doi: 10.3390/info9110273.
[26] G. R. Choudhary, S. Kumar, K. Kumar, A. Mishra, and C. Catal, "Empirical analysis of change metrics for software fault prediction," *Comput. Electr. Eng.*, vol. 67, pp. 15–24, Apr. 2018, doi: 10.1016/j.compeleceng.2018.02.043.

**AHMED ALI SEYAM** is currently a Graduate Student with the Computer Science Program, The Higher Institute of Computer Science & Information Technology. He is also a Master's Student of Software Engineering with Helwan University. He is also a Teacher Assistant with The Higher Institute of Computer Science & Information Technology. His research interest includes software engineering.

**ABEER HAMDY** has been an Associate Professor with the Faculty of Informatics and Computer Science (ICS), The British University in Egypt, since 2009. In 2015, she was appointed as the Coordinator of Software Engineering specialization and the Director of the Quality Office. In 2018, she was also appointed as the Programme Director.

**MARWA SALAH FARHAN** is currently an Associate Professor with the Information Systems Department, Faculty of Informatics and Computer Science, The British University in Egypt, and the Faculty of Computers and Information, Helwan University, Egypt. Her research interests include software engineering semantic web, advanced database management, big data, and data science.