第 49 卷 第 4 期
2022 年 4 月

湖南大学学报（自然科学版）
Journal of Hunan University (Natural Sciences)

Vol. 49 No. 4
April 2022

Open Access Article https://doi.org/10.55463/issn.1674-2974.49.4.12

An Enhanced Information Retrieval-Based Bug Localization System with Code Coverage, Stack Traces, and Spectrum Information

Shakirat Aderonke Salihu*, Oluwakemi Christiana Abikoye

Department of Computer Science, University of Ilorin, Ilorin, Nigeria

Abstract: Several strategies such as Vector Space Model (VSM), revised Vector Space Model (rVSM), and integration of additional elements such as stack trace and previously corrected bug report have been utilized to improve the Information Retrieval (IR) based bug localization process. Most of the existing IR-based approaches make use of source code files without filtering, which eventually increases the search space of the technique, thereby slowing down the bug localization process. This study developed an enhanced IR-based bug localization model as a viable solution. Specifically, an enhanced rVSM (e-rVSM) is developed based on the hybridization of code coverage, stack traces, and spectrum information. Combining the stack trace and spectrum information as additional features can enhance the accuracy of the IR-based technique by boosting the bug localization process. Code coverage analysis was conducted to remove irrelevant source files and reduce the search space of the IR technique. Then the filtered source files are preprocessed via tokenization and stemming from selecting relevant features and removing unwanted words. The preprocessed data is further analyzed by finding similarities between the preprocessed bug reports and source code files using the e-rVSM. Finally, scores for each source code and suspected buggy files are ranked in descending order. The performance of the proposed e-rVSM is tested on two open-source projects (Zxing and SWT), and its effectiveness is assessed using TopN rank (where N = 5, 10), Mean Reciprocal Rank (MRR), and Mean Average Precision (MAP). Findings from the experimental results revealed the effectiveness of e-rVSM in bug localization. In particular, e-rVSM recorded a significant Top 5 (80.2%; 65%) and Top 10 (89.1%; 75%) rank values on SWT and Zxing dataset respectively. Also, the proposed e-rVSM had MRR values of 80% and 54% on the SWT dataset and MAP values of 61.22% and 47.23% on the Zxing dataset.

Keywords: information retrieval, bug localization, vector space model.

具有代码覆盖率、堆栈跟踪和频谱信息的增强型基于信息检索的错误定位系统

摘要：向量空间模型、修订的向量空间模型以及堆栈跟踪和先前更正的错误报告等附加元素的集成已被用于改进基于信息检索的错误定位过程。大多数现有的基于信息检索的方法都使用未经过滤的源代码文件，这最终增加了该技术的搜索空间，从而减慢了错误定位过程。本研究开发了一种增强的基于信息检索的错误定位模型作为可行的解决方案。具体来说，基于代码覆盖率、堆栈跟踪和频谱信息的混合开发了一个增强的修正向量空间模型。将堆栈跟踪和频谱信息组合为附加功能可以通过促进错误定位过程来提高基于信息检索的技术的准确性。进行代码覆盖分析以删除不相关的源文件并减少信息检索技术的搜索空间。然后，过滤后的源文件通过标记化进行预处理，并源于选择相关特征和删除不需要的单词。通过使用电子修订的向量空间模型发现预处理的错误报告和源代码文件之间的相似性，进一步分析预处理的数据。最后，每个源代码和可疑错误文件的分数按降序排列。提议的增强型修正向量空间模型的性能在两个开源项目（志兴和标准小部件工具包）上进行了测试，并使用前 N 排名（其中 N = 5、10）、平均倒数排名和平均平均值评估其有效性精确。实验结果揭示了增强的修正向量空间模型在错误定位中的有效性。特别是，增强的修正向量空间模型分别在标

Received: February 13, 2022 / Revised: March 27, 2022 / Accepted: March 29, 2022 / Published: April 30, 2022

About the authors: Shakirat Aderonke Salihu, Oluwakemi Christiana Abikoye, Department of Computer Science, University of Ilorin, Ilorin, Nigeria

Corresponding author Shakirat Aderonke Salihu, salihu.sa@unilorin.edu.ng

准小部件工具包和志兴数据集上记录了显着的前 5(80.2%; 65%)和前 10 名(89.1%; 75%)排名值。此外，提议的增强型修订向量空间模型在标准小部件工具包数据集上的月经常性收入值为 80%和 54%，在志兴数据集上的平均精度值为 61.22%和 47.23%。

关键词：信息检索、错误定位、向量空间模型。

1. Introduction

Software bugs are instances of unanticipated behavior that affect the software's performance. It may also be viewed as an unsuitable procedure in software that would generally provide inaccurate results. Software engineers often devote a significant amount of time and resources to developing quality and reliable software, yet despite this, software systems are still vulnerable to software bugs or defects [1-3]. Moreover, software bugs may not be present or noticed immediately, but as update continues and complexity increases, there is a likelihood of the presence of software bugs. Therefore, software bug reports are critical for every software development project. A software user may warn the software development team about unexpected consequences of using their product by filing a bug report [4-8]. Software bug reports are widely employed in various research domains, including bug prediction, bug localization, and bug triaging; they are submitted via the Bug Tracking Systems (BTS) [9-11]. Software debugging entails two stages: bug localization and bug fixing [12]. To formally address software bugs, software developers and engineers set up software bug repositories to collect software bug reports from users [13].

Bug localization is the process by which a bug is located within the source code files. Although it is necessary for the software development process, particularly software maintenance, it is time-consuming and incredibly costly [5, 7, 14, 15]. As reported in several studies, software testing and debugging activities take up 75% of the software development cost. In comparison, software maintenance activities consume roughly 90% of the software development life cycle (SDLC) [16-18]. For example, Lucia et al. [19] assessed 374 bugs from Rhino, AspectJ, and Lucene software repositories in their study. It was discovered that 84-93% of software bugs are found in the first 20% of the source code files. Also, in the respective study of Zhou et al. [20] and Anvik et al. [21], it was disclosed that 300 bug reports are filed daily. In the case of large software products, the number of bug reports in the repository may be so high that it will be difficult for the software development team to address this large number of bug reports in the least amount of time.

Furthermore, developers often do bug localization manually, which is tedious. As a result, reliable ways to automatically detect software bugs from bug reports are necessary [7, 9, 14, 22, 23]. Several studies have proposed automated bug localization techniques to overcome this issue, which take as input bug reports and use textual information from these reports' summary and description fields to find the buggy source code files. Many of these approaches are information retrieval (IR)-based, and they usually work by computing similarities between a reported bug and source code files. Then, the source code files are ranked based on their similarities to a reported bug [24-27].

IR-based bug localization techniques have gained significant attention due to their minimal external dependencies [28-30]. Some of the existing IR-based bug localization techniques such as Vector Space Model (VSM) [31, 32], revised Vector Space Model (rVSM) [20], Modified revised Vector Space Model (MrVSM) [33], and Smoothed Unigram Model (SUM) [34] has been reported to be effective in bug localization. VSM was initially used for automatic bug localization by finding the similarities between bug reports and source code. Zhou et al. [20] extended VSM to rVSM, which considered the complexity of the source codes in finding similarities. Wong et al. [35] introduced the magnifier parameter, β, to control how much weight is given to complex or large source codes. Wang et al. [11], Youm et al. [36], and Dao et al. [37], in their respective studies, proposed the integration of execution information and enhancement features into VSM in other to improve the accuracy of the automatic bug localization process.

Despite the effectiveness of these methods, there is still a need for continuous development of sophisticated methods with high bug localization accuracy. From current studies, some features have been identified as potential factors that can improve the effectiveness of IR-based bug localization methods. Specifically, features such as version control history [33], source code metrics [36], code churn [38], stack traces [35] and program execution details (coverage, slicing and spectrum information) [11, 37, 39]. Collectively, the software team requires these features (information) to fix the bug. Nonetheless, most of the existing bug localization solutions usually consider the whole source code files without filtering, which often included irrelevant source code files that were not

covered by any failure run. It naturally expands the search space of IR-based techniques. Furthermore, several of the notable IR-based bug localization methods that utilized rVSM used a constant magnifier parameter, β, across all projects, which did not give ideal results due to the differences in the open-source projects' features.

This study proposes an enhanced IR-based bug localization with an automatic generation of magnifier parameter β for the following: large source code files; code coverage to filter the irrelevant source files; integration of stack traces and spectrum information as an additional feature boosting the accuracy of the model.

Specifically, the significant contributions of this work are highlighted below:

1. Integration of code coverage analysis into the bug localization model for filtering the source code files to reduce the search space of IR-based models

2. Automatic generation of the magnifier parameter β for large source code files in the rVSM model.

3. Integration of stack traces analysis and spectrum information to enhance bug localization accuracy of the model.

The remainder of the paper is structured as follows: Section 2 addresses major related works, and Section 3 explains the research methodology used in this study. Section 4 presents and discusses the experimental outcomes. Finally, Section 5 wraps up the research and suggests prospective future studies.

2. Related Works

This section reviews previous efforts relevant to the various methodologies for bug localization. They are provided in terms of the approaches used in the investigations.

VSM was first developed as an information retrieval tool that can help find similarities between two documents and rank them based on their similarity scores. Poshyvanyk et al. [40] utilize an IR technique (Latent Semantic Indexing - LSI) for feature location named PROMISER. The developed method experimented on three bugs in eclipse and five in Mozilla, which is very small. In a related study, Lukins et al. [41] used Latent Dirichlet allocation (LDA) for automated bug localization; the approach was based on grouping similar phrases into a subject and then identifying terms that belong to that topic. Similarly, Nguyen et al. [5] built a model called BugScout, an automated solution that assists developers in reducing the time they spend searching for problems. It was an LDA-based strategy in which some of the technical phrases in the bug report were utilized as topics in the textual contents of bug reports and source code files. It correlates bug reports and related buggy files by their similar topics. It also included several improvement features, such as past bug reports, to improve the model's performance. However, LDA has the challenge of determining a suitable number of topics to employ. Besides, its ability to evolve or change with topics over time is a significant drawback [42, 43].

In enhancing VSM, Zhou, et al. [20] proposed a revised VSM (rVSM) that assigns weight based on the difficulty of a code. They based their IR techniques on previously corrected bug reports to maximize performance. The rationale underlying their approach is that VSM assigns the same weight to simple and complex code, even though complex code contains more errors than simple code. For their weights determination, another equation was developed and added to the standard VSM. The approach was tested on four open-source projects, with the results outperforming previous works. Also, Saha et al. [39] improved the VSM by considering the bug report and source code structures; more weight or importance was assigned to titles in the bug report than a summary, and the approach was based on the fact that the title of the report can also help in localizing bugs. Wong et al. [35] integrated rVSM with stack traces and codes segmentation. Their experimental findings showed that the two heuristics techniques integrated with rVSM amplified its accuracy. Furthermore, they posited that an additional parameter is introduced to control the weight given to each source file with large size or complex code. However, the suggested parameter cannot be automatically determined or generated.

Some studies have also been conducted that employed the dynamic technique. Ye et al. [44] employed word embeddings for improving automated bug localization. Their research aimed to close the lexical gap by projecting natural language statements and code snippets as meaning vectors in single representation space. The word embeddings are initially trained on API to get more similarities. Takahashi et al. [45] utilized open-source code smells as added information to improve the efficiency of IR-based problem localization. Their technique indicated that code smells may be utilized in conjunction with IR to locate bugs. Le et al. [26] considered both static and dynamic approaches for bug localization alongside suspiciousness words in the bug report as an additional feature to enhance its performance. Zhang et al. [25] developed a spectrum-based bug localization using the PageRank algorithm. Given the original program spectrum information, the PageRank Algorithm is used to recompute the spectrum information by considering the contributions of various tests. In a study by Dao et al. [37], execution traces were employed to improve the performance of IR approaches. The three-execution information was employed differently on three separate IR- methods. Their research highlighted how execution information might be integrated with the IR approach to improve performance.

Nowadays, Machine Learning (ML) and Deep

Learning (DL) methods have also been employed for bug localization. Sangle et al. [46] combined ML and DL methods with rVSM for bug localization. For improving its accuracy, the proposed method employed a Multilanguage project composed of Java and C-language and certain extra features such as previously addressed bug reports. Similarly, Mahajan and Chaudhary [47] employed the textual contents of bug reports and LDA, together with vectorization, to rank problematic files. They used an approach that combines LDA and vectorization to score the ranking files. Qiu et al. [48] developed a Just-In-Time defect identification and localization (JITO). Their proposed method was created as a plugin in an Integrated Development Environment (IDE) to assist developers in identifying and locating bugs. Cheng et al. [49] designed a model that combines IR technology, word embedding, and Deep Neural Network (DNN). The IR technique was used to determine the exact similarity between the bug report and the source files; the terms in the bug report and the source files of different code tokens are linked by word embedding. The DNN technique integrated the extracted features to determine the correlation between the bug report and the source files. Xiao et al. [50] proposed an approach based on employing a convolutional neural network (CNN), random ensemble forests (RF), and multi-grained scanning to extract semantic and structural features from the word vectors derived from bug reports and source files. Tantithamthavorn et al. [51] studied the impact of the choice of IR-based classifier configuration on the model's performance and the required effort to examine the source code entities before locating a bug at the method level. Lam et al. [52] combined DNN with rVSM. In this approach, IR was used to collect the feature on the textual similarity between bug reports and source files, while DNN was used to learn to relate the terms in bug reports to potentially different code tokens and terms in source files. Also, in a study by Loyola et al. [53], their model was based on learning feature representations from source changes extracted from the project history at both syntax and code change dependency perspectives to support bug localization. A well-structured end-to-end architecture was incorporated into the system to integrate feature learning and ranking between bug reports and source code changes.

Despite the reported effectiveness of these existing methods, there is still a need for more sophisticated ones as the implication for prompt and accurate bug localization is vital to software development processes. Consequently, an enhanced rVSM is proposed in this study. Specifically, the suggested technique would use a mathematical equation to automatically compute the value of the magnifier parameter, β, resulting in source code files from various projects, even if they include the same number of words, not having the same magnifier value. This enhancement is expected to give an improved result compared with Wong et al. [35], where the parameter was manually determined and all the projects used have the same value of the parameter, which is considered inappropriate due to the different characteristics of the projects. Furthermore, the proposed method improves the localization process by using code coverage before using the IR technique to filter the source files. In addition, stack traces and spectrum information are merged to improve the performance of the proposed IR technique. However, these two elements have not been combined to improve the accuracy of the automatic bug localization process.

3. Methodology

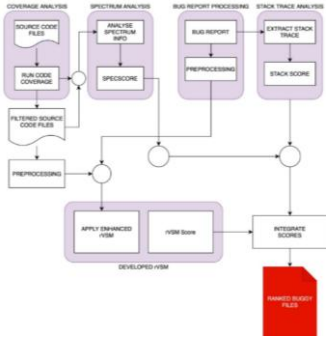
This section outlines and describes the proposed e-rVSM method, tested datasets, and performance assessment metrics.

3.1. Enhanced Revised Vector Space Model (e-rVSM)

In the proposed e-rVSM, the entire process is divided into four sections: code coverage analysis, data preprocessing, development of e-rVSM, stack trace, and spectrum information analysis. The code coverage analysis is carried out before the application of the e-rVSM; this is to ensure a reduction in the search space as only source files identified to be failed will be passed to the IR. Each of the stages in the model is discussed in detail in the subsequent sections.

3.2. Code Coverage Analysis

The code coverage refers to all classes and methods covered by the program execution. The coverage can be achieved through code coverage analysis via code coverage tools, where coverage is run to filter the source files. This analysis helped determine source files that failed and those that were passed. That aims to help reduce the search space for the Information Retrieval technique. Consequently, only failed source files are passed to the IR-based model. The procedure for the code coverage analysis is presented in fig. 1b. Code coverage tools incorporated into an Integrated Development Environment (IDE) for running and report generation are required for carrying out this analysis.



(a)

```
INPUT: Source files collected from open-source project
OUTPUT: Passed source files and Failed source files
Step 1: Import Source code files
Step 2: Coverage Process
2.1   import Source files RD
2.2   let TC be the total number of Test Cases
2.3     let NF be total number of failed test cases
2.4     let NP be total number of passed test cases
2.5   Run the source file RD using jcrasher
2.6    For each source file
2.7      list all classes C
2.8      For each class C
2.9         do
2.10           Generate test cases TC
2.11           Run TC using Junit
2.12           if (TC fails )
2.13                   increment NF by 1
2.14                   return failed (Class names C)
2.15           else
2.16                   increment NP by 1
2.17                   return Passed (Class names C)
2. 18          end do
2. 19          generate report
2. 20     end for
2. 21  end for
```

(b)

Fig. 1 (a) The framework of the proposed e-rVSM; (b) Code coverage algorithm

Several tools can be used, including Jcrasher, Java Code Coverage (JaCoCo) for Java applications, etc. In this study, Jcrasher and JUnit are used to analyze. These tools (Jcrasher and Junit) try to reveal defects by causing the program to throw undeclared runtime exceptions. In addition, the error reporting phase was made to be automatically generated as a CSV file.

3.3. Preprocessing of Source Codes and Bug Reports

Preprocessing is one of the critical issues that must be properly carried out when performing the IR process. There are several ways of preprocessing data. However, three preprocessing stages were used to preprocess the source code files and bug reports for this proposed model.

These stages are Tokenization, Removal of stop words, identifier splitting, and stemming:

1) Tokenization involves breaking down the text into terms for each source code file and bug reports. This process was achieved using a java class called string tokenizer. That enables the bug report, and the source code files to be seen as a composition of terms.

2) Removal of stop words: Stop words refer to punctuation, question marks, and any unwanted words or character. Removing stop words that were used includes removing java keywords and removing words such as 'must', 'is', 'so', 'are', and a single character.

3) Stemming involves reducing a word to its root form to enable similar words to be represented using the same term. Stemming is usually used for words in Information Retrieval (IR) systems so that words with almost the same meaning are grouped as the same concept. That was achieved through the use of a porter stemming algorithm that is based on rules and cases. The stemming process is presented based on the rules generated to track words to be stemmed. For instance, 'Complete', 'Completed', 'Completing' are reduced to 'Complet'.

3.4. Development of e-rVSM

In this proposed IR- model, source code files are considered a corpus, and bug reports as the query. The IR technique takes the bug report as a query and creates a model to search the source code files based on the query from the bug report. The similarity score or the level of relevance between the source code files and the bug report is computed. In traditional or classical VSM, the relevance score between a document represented as d and a query q is computed as the cosine similarity between their corresponding vector representations as described in Equation 1:

$$\text{similarity}(q,d) = \cos(q,d) = \frac{\overrightarrow{U_q} \cdot \overrightarrow{U_d}}{|\overrightarrow{U_q}|\,|\overrightarrow{U_d}|} \quad (1)$$

where $\overrightarrow{U_q}$ and $\overrightarrow{U_d}$ are a vector of term weights for the query q and document d respectively. $\overrightarrow{U_q}$. $\overrightarrow{U_d}$ represents the inner product of the two vectors. The term weight w is computed based on the term frequency (*tf*) and the inverse document frequency (*idf*). Thus, the tf and idf are defined in Equation 2.

$$tf(t,d) = \log(f_{td}) + 1, \quad idf(t) = \log(\frac{\#D}{n_t}) \quad (2)$$

where f_{td} denotes the number of occurrences of a term *t* in document *d*, n_t denotes the number of documents that contain the term *t*, and #D represents the total number of documents (source code files) in a particular open-source project. Equation 2 is used by rVSM to define *tf* and *idf*. Therefore, each term weights *w* in the document vector $\overrightarrow{U_d}$ and its $|\overrightarrow{U_d}|$ are calculated in Equation 3.

$$w_{t\in d} = tf_{td} \times idf_t = (\log f_{td} + 1) \times \log(\frac{\#D}{n_t}) \quad (3)$$

$$|\overrightarrow{U_d}| = \sqrt{\sum_{t\in d}((\log f_{td} + 1) \times log(\frac{\#D}{n_t}))2} \quad (4)$$

Similarly, the vectors of term weights for the query $\overrightarrow{U_q}$ and $|\overrightarrow{U_q}|$ were obtained in Equations 5 and 6.

$$w_{t\in q} = tf_{tq} \times idf_t = (\log f_{tq} + 1) \times \log(\frac{\#Q}{n_t}) \quad (5)$$

$$|\overrightarrow{U_q}| = \sqrt{\sum_{t\in q}((\log f_{tq} + 1) \times log(\frac{\#Q}{n_t}))2} \quad (6)$$

It is to be noted that classical or traditional VSM does not favor large documents when ranking files. In other words, large documents are often poorly

represented, whereas, in rVSM, large source code files are given more consideration. Thus, $L(U_d)$ is defined in Equation 7 to model the document length in rVSM, which is based on logistic function.

$$L(U_b) = \frac{1}{1+e^{-Norm(\#t \in U_b)}} \quad (7)$$

Equation 7 is used to ensure that large documents are given more consideration or higher scores during ranking. To control how much favor is given to large files size represented by the number of terms, this proposed model (e-rVSM) intends to enhance the rVSM model by automatically deriving the value of the magnifier parameter *βeta* introduced by Wong et al. [35]. It is calculated automatically using Equation 8

$$\beta eta = \left(\frac{\sum_{i=1}^{N} x_{i\,(\#terms)}}{KN}\right) \times 100\% \quad (8)$$

where N represents the total number of source code files in each open-source project, *#terms* represent the number of terms in the source code files, *and K >= 10*. With the parameter *βeta* introduced, Equation 8 now becomes Equation 9.

$$L(U_d) = \frac{1}{1+e^{-\beta eta \times Norm(\#t \in U_d)}} \quad (9)$$

Norm is the normalization of the values. The Norm is defined as follows*: $(n - n_{min}) / (n_{max} - n_{min})$*

The scoring equation for the enhanced rVSM is shown in Equation 10.

$$\begin{aligned} \text{rVSMScores}(q, d) &= L(U_d) \times \cos(q, d) \\ &= \frac{1}{1+e^{-\beta eta \times Norm((t \in U_d)}} \times \frac{1}{\sqrt{\sum_{t \in q}((\log f_{tq}+1) \times log(\frac{\#Q}{n_t}))2}} \times \frac{1}{\sqrt{\sum_{t \in d}((\log f_{td}+1) \times log(\frac{\#D}{n_t}))2}} \\ &\times \sum_{t \in q \cap d}((\log f_{tq} + 1) \times (\log f_{td} + 1) \times log(\frac{\#D}{n_t})\,2 \end{aligned} \quad (10)$$

3.5. Stack Trace Analysis

The stack trace information was retrieved from the bug report. For analyzing a bug report with stack traces, regular expressions are used. The regular expression defined followed these three patterns:

1) STACK_TRACE_INFO_REGEX = "\\((. *?)\\)": This detects all bug reports with an open and close bracket.

2) "([^\\s]+(\\.(?i(java))$)" – It extracts bug report with a bracket, space and .java

3) "\\s+([a-zA-Z][\\.\\w]" - This compares the source files detected with the source files in the corpus. If such file exists then it is extracted and allocated score based on Equation 11.

Given a bug report *q*, let *st* represent the stack trace in *q*, *ST* be the set of all stack traces; S_{stack} represents the set of files in the stack trace, $S_{stackimport}$ represents the set of imported files from S_{stack} and *k* represents the total number of all S_{stack} and $S_{stackimport}$ in the corpus. Then, with the regular expression earlier defined to extract the stack traces, the stack trace score was calculated using Equation 11.

$$StackScore(b, st) = \begin{cases} \frac{k}{ST} b \in S_{stack} \\ \frac{k}{ST} b \in S_{stackimport} \\ 0\ otherwise \end{cases} \quad (11)$$

3.5.1. Spectrum Information Analysis

The spectrum suspiciousness score is calculated using Tarantula, one of the techniques for extracting spectrum information. It is rated efficient in generating spectrum suspiciousness scores and, thus, most popular. Equation 12 depicts the Tarantula equation used to calculate the suspiciousness score of the program element denoted as:

$$SpecScore(d) = \frac{\frac{NF(d)}{NF}}{\frac{NF(d)}{NF} + \frac{NP(d)}{NP}} \quad (12)$$

NF(d) denotes the Number of Failed test cases executing the program element, *NF* denotes the total number of failed test cases, *and NP(d)* refers to the number of passed test cases that execute the program element. *NP* represents the total number of passed test cases. The values of *SpecScore* and *StackScore* are also normalized using the same normalized equation for rVSMScore.

3.5.2. Integration of Scores

All the three scores generated from rVSM are denoted as rVSMScores in Equation 10. Stack trace score represented as *StackScore* in Equation 11 and spectrum suspiciousness score represented as *SpecScore* in Equation 12 is integrated to rank the buggy files. That will be achieved by finding the average of the three scores as expressed in Equation 13. The average score is used for ranking the source code files suspected to be buggy in descending order.

$$IntScore(d) = \frac{\sum f(rv_i + st_j + sp_k}{\sum f} \quad (13)$$

where rv_i is the rVSMScores, st_j is the StackScore, and sp_k is the SpecScore.

3.6. Description of Open-Source Dataset

The datasets used to evaluate the proposed models are source code files and bug reports from open-source SWT and Zxing (Zebra Crossing) projects. These datasets are publicly available and thus, downloaded from their respective websites — this study based its choice on the Java code platform and the android application. Also, the preference for the selected datasets extends to open-source projects that have been previously used in existing research. For example, SWT is part of the Eclipse foundation project and consists of Seven Hundred and Thirty-Six (736) source files and Ninety-Eight (98) bug reports. In contrast, Zxing consists of Four Hundred and Twelve (412) source code files and Twenty-Seven (27) bug reports. A brief description of the datasets is presented in Table 1.

Table 1 Studied open-source software projects

Project	Description	No. Source Files	No. of Bug Reports
SWT v3.1	A standard widget toolkit for Java was designed to provide efficient and portable access to the GUI facilities of the OS on which it is implemented.	736	98
Zxing	A barcode image processing library is licensed under apache and supports 1D Product, 1D Industrial, and 2D barcodes.	412	27

3.7. Performance Evaluation Metrics

In terms of performance evaluation, bug localization models based on the proposed and other methods were analyzed using TopN, Mean Average Precision (MAP), and Mean Reciprocal Rank (MRR) values. These metrics are often used in existing SDP studies to assess the performance of bug localization methods[11, 35, 46, 54].

1) Top N Rank: It is a metric used to calculate the number of bug reports that have their buggy files found and ranked within the top N, where N can be 1, 5, or 10.

2) Mean Average Precision (MAP): It is an IR metric used in evaluating the ranking approaches; it calculates the average precision values among a set of queries. It emphasizes all the ranked buggy files. The higher the value of MAP, the better the performance of the approach. The Mean Average Precision is computed by taking the mean of the average precision scores across all queries.

3) Mean Reciprocal Rank (MRR): The reciprocal rank of a query is the reciprocal of the position of the first buggy files in the result that is ranked to be suspicious. MRR is the mean of the reciprocal ranks of the results of a set of queries Q.

4. Results and Discussion

The code coverage analysis on SWT Source Files was based on classes in the source file. SWT is an open-source from the Eclipse foundation. It is a standard widget toolkit for JAVA, and 736 source files are retrieved and compiled from this open-source project. The files are first checked to see if all the necessary executable .jar files needed for the running by the Jcrasher are contained in the open-source folder before it is imported to the NetBeans environment for coverage analysis.

From these source files, Jcrasher generates a list of 6069 classes and several test cases for each class. It then makes the Junit reproduce their error-revealing behavior. The error listings are reported in the form of CSV files. As presented in Table 2, in the column meant for error, any value greater than 0 signifies that the class has an error. Out of these classes generated, only 1691 are found to contain error values greater than 0. That implies that these classes contain errors and are therefore filtered out from the project.

A similar procedure is repeated for the Zxing dataset. During this process, the Jcrasher automatically retrieved all the classes in the open-source and generated test cases for each. These source files contain 4034 classes, while 651 classes were found to contain error values greater than 0. Similarly, these classes contain errors and are removed from the project. Table 2 and Table 3 show the code coverage from SWT and Zxing.

Table 2 Samples of code coverage analysis for SWT

S/N	Source Files	Classes	No. TC Generated	Error/No. TC Failed
1	DND.java	DNDTest2	3	3
2	DND.java	DNDTest3	9	9
3	ImageData.java	ImageDataTest1	34	1
4	ImageData.java	ImageDataTest107	500	1
5	ImageData.java	ImageDataTest108	170	36
6	ImageData.java	ImageDataTest122	500	15
7	ImageData.java	ImageDataTest123	61	12
8	ImageData.java	ImageDataTest191	500	21
9	ImageData.java	ImageDataTest192	470	30
10	ImageData.java	ImageDataTest196	378	2
11	ImageData.java	ImageDataTest2	396	3
12	ImageData.java	ImageDataTest200	81	9
13	ImageData.java	ImageDataTest215	500	17
14	ImageData.java	ImageDataTest22	329	32
15	ImageData.java	ImageDataTest4	315	1
16	ImageData.java	ImageDataTest7	453	5
17	ImageData.java	ImageDataTest91	500	3
18	ImageData.java	ImageDataTest92	500	28
19	ImageData.java	ImageDataTest93	299	18
20	Compatibility.java	CompatibilityTest18	4	1
21	Compatibility.java	CompatibilityTest19	303	60
22	PngDeflater.java	PngDeflaterTest2	4	2
23	Library.java	LibraryTest3	3	3
24	Library.java	LibraryTest4	6	5

Continuation of Table 2

25	XPCOMInit.java	XPCOMInitTest2	216	109
26	XPCOMInit.java	XPCOMInitTest3	216	109
27	XPCOMInit.java	XPCOMInitTest4	1	1

Table 3 Samples of code coverage analysis for Zxing

S/N	Source Code Files	Test Cases	No. of Test Runs/Generated	Error/No. of Failed TC
1	ParsedReaderResultTestCase.java	ParsedReaderResultTestCaseTest5	1	1
2	SMSParsedResult.java	SMSParsedResultTest1	500	32
3	SMSParsedResult.java	SMSParsedResultTest10	500	35
4	SMSParsedResult.java	SMSParsedResultTest100	500	21
5	SMSParsedResult.java	SMSParsedResultTest101	500	39
6	SMSParsedResult.java	SMSParsedResultTest102	500	37
7	SMSParsedResult.java	SMSParsedResultTest103	500	35
8	SMSParsedResult.java	SMSParsedResultTest104	500	30
9	SMSParsedResult.java	SMSParsedResultTest105	500	28
10	SMSParsedResult.java	SMSParsedResultTest106	500	50
11	SMSParsedResult.java	SMSParsedResultTest107	500	35
12	SMSParsedResult.java	SMSParsedResultTest108	500	36
13	SMSParsedResult.java	SMSParsedResultTest109	500	29
14	SMSParsedResult.java	SMSParsedResultTest11	500	36
15	SMSParsedResult.java	SMSParsedResultTest110	500	36
16	SMSParsedResult.java	SMSParsedResultTest111	500	23

4.1. Summary of the Code Coverage Analysis Result

Table 4 summarizes the code coverage analysis for the two open-source projects used for this study. The number of source files, classes generated, errors detected, and the number of failed source files retrieved for the two open sources are indicated in Table 4.

Table 4 Summary of the code coverage analysis

S/N	Open-Source Projects	No. of Source Files	No. of Classes	No. of Errors	No. Failed Source Files	% Code Coverage
1.	SWT	736	6069	1691	106	90%
2.	Zxing	412	4034	651	52	85%

From Table 4, SWT has 736 source code files; 6069 classes were extracted by the Jcrasher, out of which 1691 contain an error and have about 90% code coverage, while 106 source files were filtered out as failed source files. Zxing has 412 source code files, 4034 classes were extracted, and 651 of them have error values greater than 0. The percentage code coverage is 85%, and the number of failed source files filtered out is 52. Therefore, 106 and 52 source files are filtered out as failed source files for SWT and Zxing, respectively.

It can be deduced from the coverage that those passed source files are irrelevant to failure and has been removed, thereby reducing the search space of the IR technique. These failed source files will be passed to the preprocessing stage of the IR technique.

4.2. Preprocessing of Filtered Source Code Files and Bug Reports

The preprocessing of the source files filtered after the code coverage analysis and bug reports from the two open-source projects were done through Tokenization, Removal of stop words, identifier splitting, and stemming. Also, the number of terms in each source file was noted, and the number of occurrences of such terms.

As a result, all the failed source files from code coverage and bug report passed through the preprocessing stage. Fig. 2 shows samples of a source file and bug report after preprocessing.


Term Key(s):cancel, inform, dragEnter, Operat, complet, String, ERROR, CANNOT, INIT, DRAG, successfulli, ERROR, INVALID, DATA, Unknown, Throw, press, DROP, MOVE, eclips, happen, ad, pass, past, IllegalArgumentExcept, Kei, SELECTION, CLIPBOARD, FEEDBACK, SELECT, left, except, sub, cut, DROP_TARGET_MOVE, kei, result, current, releas, NLS, IBM, term, defin, contain, set, intern, right, FEEDBACK, SCROLL, store, materi, SWT, throw, v, DROP, LINK, invok, remov, Event, oper, DRAG, SOURCE, KEY, dd, reserv, correct, ERROR, CANNOT, SET, CLIPBOARD, dl, specif, dt, swt, either, DragEnter, legal, Eclips, implement, INIT, DRAG, MESSAGE, us, DropTargetListen, Failur, given, accompani, initi, usual, last, NON, look, DragSourc, http, Class, sinc, drop, cursor, allow, string, origin, Contributor, non, fatal, button, appli, param, client, SWTError, modifi, org, perform, distribut, target, chanc, RuntimeExcept, refer, chang, FEEDBACK, NONE, DND, mark, base, CANNOT, SET, CLIPBOARD, MESSAGE, sourc, select, Undefin, properti, visibl, program, type, clipboard, three, DropAccept, displai, dnd, href, drag, enter, epl, applic, creat, item, method, made, INVALID,DATA,MESSAGE, termin, scroll, Drop, system, field, Object, DROP_DEFAULT, begin, FEEDBACK_INSERT_AFTER, tabl, other, valu, widget, Sampl, DragSetData, DragLeav, copi, requir, hresult, mechan, ESC, DragStart, event, map, getData, avail, Licens, setData, dragFinish, appropri, Drag, ECS, report, Clipboard, FEEDBACK, EXPAND, logic, msg, argument, constant, data, It, Corpor, Copyright, hit, whenev, SWTException, shown, www, html, DragOperationChang, occur, tree, format, thrown, choos, Public, DragSourceListen, expand, transfer, DROP_NONE, cleanup, effect, updat, DropTarget, code, DROP, TARGET, KEY, CLIPBOARD, link, insert, ERROR_CANNOT_INIT_DROP, error, platform, DragEnd, lang, make, provid, on, move, OR, recover, OS, DragOver, choosen, Error, Data, control, FEEDBACK, INSERT, BEFORE, boundari, illeg, mous, dragOperationChang, API, INIT, DROP, MESSAGE, Limit, DROP, COPY, user

Term Value(s):1, 2, 2, 6, 1, 14, 2, 1, 2, 1, 2, 1, 1, 6, 3, 2, 2, 2, 2, 2, 1, 1, 1, 4, 1, 1, 1, 3, 4, 2, 1, 10, 2, 1, 1, 1, 4, 2, 1, 1, 2, 1, 5, 1, 2, 1, 1, 3, 9, 8, 1, 6, 1, 2, 2, 2, 2, 6, 5, 1, 1, 1, 1, 1, 2, 7, 2, 1, 1, 2, 5, 1, 1, 10, 2, 6, 2, 1, 6, 22, 7, 2, 2, 1, 1, 2, 1, 1, 5, 3, 1, 5, 3, 6, 3, 1, 15, 1, 1, 1, 2, 1, 13, 2, 2, 2, 9, 5, 1, 2, 1, 1, 2, 2, 1, 1, 1, 1, 18, 1, 1, 2, 2, 6, 2, 1, 2, 1, 1, 8, 1, 1, 2, 2, 1, 1, 4, 1, 29, 1, 1, 1, 1, 3, 1, 11, 3, 1, 1, 2, 1, 2, 3, 1, 2, 2, 2, 7, 1, 1, 2, 1, 1, 12, 2, 3, 12, 10, 2, 1, 2, 3, 3, 3, 2, 1, 1, 1, 5, 2, 3, 1, 1, 1, 1, 2, 1, 1, 7, 1, 15, 33, 1, 1, 1, 2, 2, 14, 5, 1, 3, 1, 1, 3, 3, 2, 2, 1, 1, 1, 6, 3, 6, 1, 2, 1, 1, 2, 2, 2, 1, 1, 4

Sum Value(s):694

Fig. 2 Source code files after preprocessing

4.3. e-rVSM Results

4.3.1. SWT

The preprocessed source code files and bug reports served as input into the IR. The total number of terms and occurrences are used, and their weights are calculated using Equation 8, as discussed in the methodology section. Fig. 3 shows the experimental result of the rVSM.

A bug report served as a query into the 106 documents in the corpus. Each of these bug reports followed the equations of IR and then had an rVSM score for the source files in the corpus. In other words, a bug report has a rVSMscore for 106 files. As presented in Table 5, the value has been normalized using the normalization equation within the range of 0.0 to 1.0.

4.3.2. Zxing

The preprocessed source code files and bug reports served as input into the IR. The total number of terms and their corresponding occurrences are used; the weight for each one of them is calculated. Fig. 4 shows the result of the rVSM.

A bug report served as a query into the source files referred to as documents or corpus. Each bug report then has a rVSMscore for the source files in the corpus. In other words, all the 27 bug reports have distinct rVSMscore for 52 files used for Zxing.

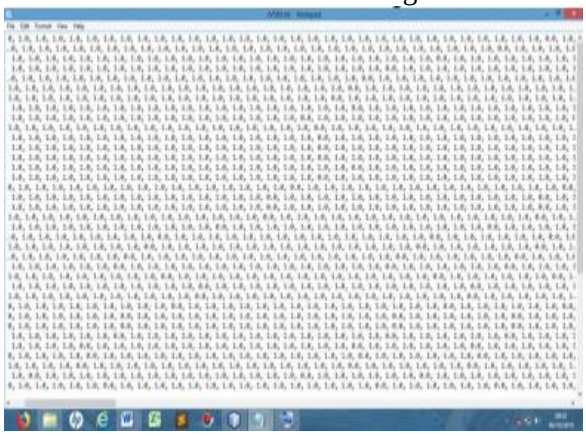


Fig. 3 rVSM scores for SWT

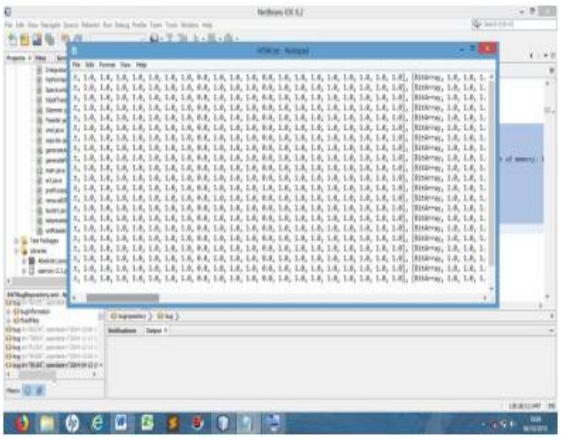


Fig. 4 rVSM scores for Zxing

4.4. Stack Trace Analysis Results

The regular expression generated detects and extracts stack trace files and those imported to the stack trace. Values of the stack trace for the source files were generated using Equation 11. This analysis was carried out for both SWT and Zxing. Fig. 5 and Fig. 6 show a sample of the results of stackscore for SWT and Zxing.

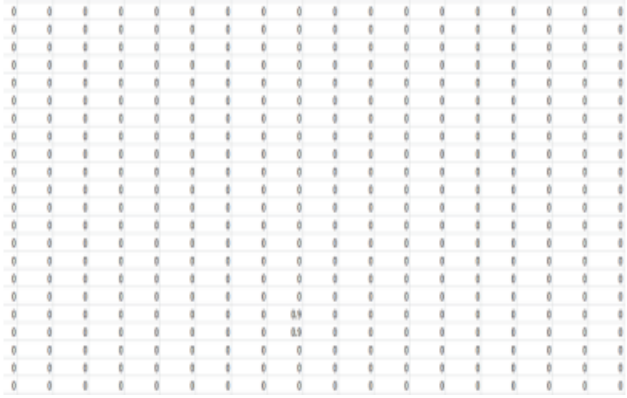


Fig. 5 Stackscore for SWT

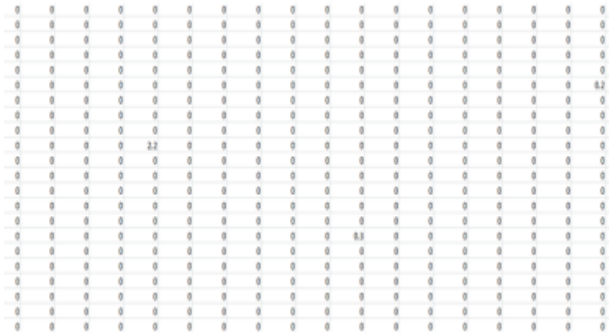


Fig. 6 Stackscore for Zxing

4.5. Spectrum Analysis Results

4.5.1. SWT Source Files

The spectrum suspiciousness score for 106 source files in SWT was calculated as discussed in the study's methodology. After the calculations, each of the source files has a Specscore. Some of the results of Specscore for SWT are shown in Table 5.

Table 5 Samples of specscore for SWT

S/N	Source files	SpecScore
1.	DND	0.1
2.	ImageData	1.0
3.	Compatibility	0.1
4.	PngDeflater	0.1
5.	Library	0.1
6.	XPCOMInit	0.5
7.	nsEmbedString	0.3
8.	nsIAppShell	0.1
9.	nsIAuthInformation	0.1
10.	nsIBadCertListener2	0.2
11.	nsIBaseWindow	0.1
12.	nsICancelable	0.1
13.	nsICategoryManager	0.1
14.	Accessible	1.0
15.	Color	1.0
16.	FontData	0.3

Continuation of Table 5

17.	Font	0.1
18.	ImageDataLoader	0.1
19.	Path	0.9
20.	Region	1.0
21.	StyledText	0.9
22.	Transform	0.1
23.	COMObject	0.3

4.5.2. Zxing Source Files

The spectrum suspiciousness score for 52 source files is calculated using the equation for spectrum analysis (See Equation 12). After the calculations, each source file has a score denoted as Specscore. Some of the results of this Specscore for Zxing are shown in Table 6.

Table 6 Samples of Specscore for Zxing

Source files	SpecScore	Source files	Spec Score
PlanarYUVLuminanceSource.java	1.0	BinaryBitmap.java	0.2
BitMatrix.java	1.0	DecoderResult.java	0.1
SMSParsedResult.java	0.8	InvertedLuminanceSource.java	0.9
SMSParsedResults.java	0.8	MultiFormatReader.java	0.2
RGBLuminanceSource.java	0.2	BlockedParsedResult.java	0.1
BitArray.java	0.1	Decoder.java	0.1
GenericGF.java	0.1	DecoderConfig.java	0.1
PlanarYUVLuminanceSources.java	1.0	DataMatrixWriter.java	0.1

4.6. Integration of the Scores

All the scores from rVSMScores, Stackscore, and SpecScore were integrated to rank the source files in descending order. Table 7 presents samples of files with scores integrated and ranked files, respectively, using SWT. A bug report with ID 108792 has its source files integrated and ranked in descending order, as shown in the tables.

Table 7 Samples of IntScore result for a bug report in SWT

Bug ID	Source Files	rVSM Score	Stack Score	Spec Score	Int Score
108792	Path	1.0	0.3	0.9	0.73333
	Accessible	1.0	0.0	1.0	0.66667
	Color	1.0	0.0	1.0	0.66667
	ImageData	1.0	0.0	1.0	0.66667
	Region	1.0	0.0	1.0	0.66667
	PngDeflater	1.0	0.9	0.1	0.66667
	StyledText	1.0	0.0	0.9	0.63333
	ImageDataLoader	1.0	0.6	0.1	0.56667
	nsIBaseWindow	1.0	0.6	0.1	0.56667
	XPCOMInit	1.0	0.0	0.5	0.50000
	FontData	1.0	0.0	0.3	0.43333
	COMObject	1.0	0.0	0.3	0.43333
	nsEmbedString	1.0	0.0	0.3	0.43333
	nsIBadCertListener2	1.0	0.0	0.2	0.40000
	Font	1.0	0.0	0.1	0.36667
	Transform	1.0	0.0	0.1	0.36667
	DND	1.0	0.0	0.1	0.36667
	Compatibility	1.0	0.0	0.1	0.36667
	nsIAppShell	1.0	0.0	0.1	0.36667
	nsIAuthInformation	1.0	0.0	0.1	0.36667
	nsICancelable	1.0	0.0	0.1	0.36667
	nsICategoryManager	1.0	0.0	0.1	0.36667

Table 7 shows some of the results of returned files after integrating the scores and ranked in descending order for bug report with ID: 108792. From this, it can be deduced that the predicted buggy files for this bug report have Path.java as the first file in the returned ranked files with the highest score.

For the Zxing dataset, three scores were integrated to form the IntScore as the final score for all the source files in this project. Table 8 presents some samples of files that have been ranked for Zxing. In addition, a bug report with ID 508 has its source files integrated and ranked, as shown in Table 8.

Table 8 Result of ranked files for a bug report in Zxing

Bug ID	Source files	rVSM Score	Stackscore	Spec Score	IntScore
508	PlanarYUVLuminanceSource.java	1.0	0.0	1.0	0.666667
	PlanarYUVLuminanceSources.java	1.0	0.0	1.0	0.666667
	BitMatrix.java	1.0	0.0	1.0	0.666667
	InvertedLuminanceSource.java	1.0	0.0	0.9	0.633333
	SMSParsedResult.java	1.0	0.0	0.8	0.600000
	SMSParsedResults.java	1.0	0.0	0.8	0.600000
	DecoderResult.java	1.0	0.3	0.1	0.466667
	DataMatrixWriter.java	1.0	0.3	0.1	0.466667
	BinaryBitmap.java	1.0	0.0	0.2	0.400000
	GenericGF.java	1.0	0.1	0.1	0.400000
	MultiFormatReader.java	1.0	0.0	0.2	0.400000
	RGBLuminanceSource.java	1.0	0.0	0.2	0.400000
	BitArray.java	1.0	0.0	0.1	0.366667
	Decoder.java	1.0	0.0	0.1	0.366667
	DecoderConfig.java	1.0	0.0	0.1	0.366667
	BlockedParsedResult.java	1.0	0.0	0.1	0.366667

In Table 8, a bug report with ID 508 was ranked in descending order based on the IntScore. The returned ranked files show that PlanarYUVLuminanceSource.java was the first among the buggy files ranked.

4.7. Performance Evaluation of the Developed Model

The model was evaluated using the three widely used IR-metrics: Top N rank (where N= 5 or 10), Mean Average Precision (MAP), and Mean Reciprocal Rank (MRR) (See Section 3.3). Before each of the parameters' values in the equation can be deduced, the changelog files for the bug report must be known. Since the bug reports used for this study have been resolved, their bug tracking system is checked to see the files that have been modified to resolve such bug reports. This task is carried out for about one hundred and forty-five bug reports used. For instance, Fig. 5 shows a sample of a changelog file for a bug report

with ID 108792 in the SWT source project.

In Fig. 7, the changelog file for the bug report was *StyledText.java*. This process is repeated for all the bug reports, and their position in the returned ranked files is noted; this is applied to calculate performance metrics. Table 9 presents the values of these metrics for the two open-source projects.

Table 9 Overall performance evaluation of the developed model

S/N	Open-Source Projects	Top 5 (%)	Top 10 (%)	MRR	MAP
1.	SWT	80.20	89.10	80.00	61.22
2.	Zxing	65.00	75.00	54.00	47.23

In Table 9, the Top N value is set to 5 and 10. For SWT open-source, the Top 5 and Top 10 have 80.20% and 89.10%, respectively, while the MRR and MAP are 80.00% and 61.22%. At the same time, Zxing has 65.00% for the Top 5 and 75.00% for the Top 10. The MRR and MAP for Zxing are 54.00% and 47.23%, respectively.

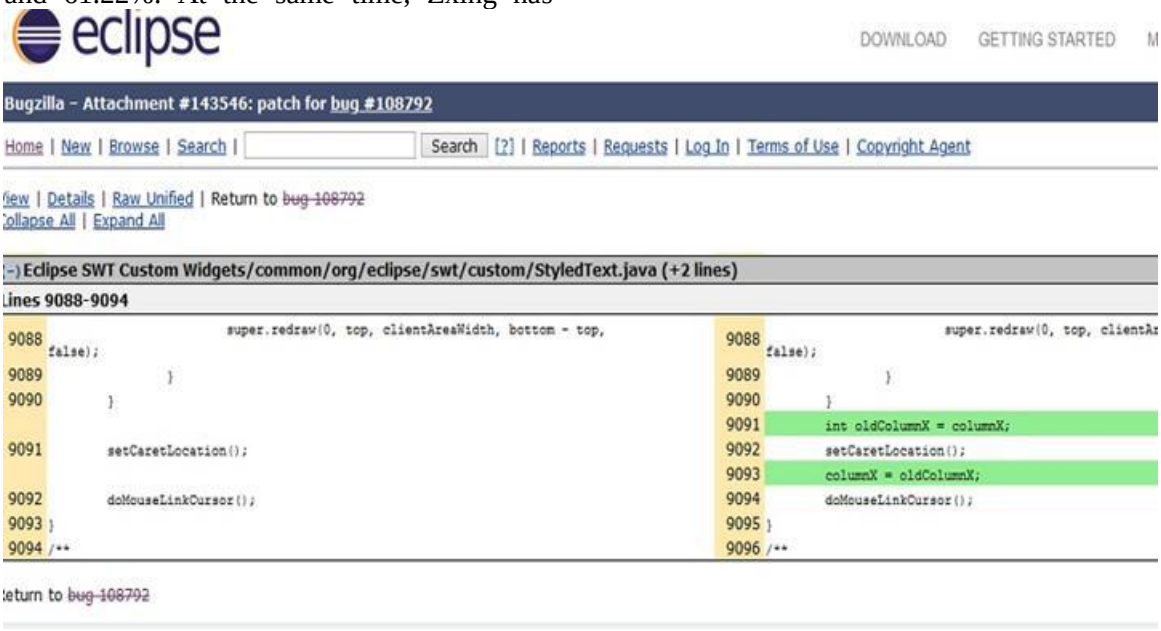


Fig. 7 Sample changelog file for bug ID 108792

4.8. Comparative Analysis of Proposed e-rVSM with Existing Methods

The overall comparison of the proposed e-rVSM with some existing bug localization methods is carried out in this section. The existing works used for comparison, as shown in Table 10, consist of BugLocator [20], an automated IR-based bug localization that used rVSM and the previously fixed bug report. BLUiR [39] is also a bug localization technique based on rVSM and BRTracer [35] developed using rVSM, segmentation of bug reports, and stack traces. From Table 10, it can be deduced that the model outperformed these existing bug localization methods. For SWT its Top 5, Top 10, MRR and MAP has 80.20%, 89.10%, 80.00% and 61.22% respectively. These values outperformed the existing works of BugLocator, BLUiR, and BRTracer. Also, Zxing has 65.00%, 75.00%, 54.00%, and 47.23% for its Top 5, Top 10, MAP, and MRR, respectively.

Table 10 Overall performance evaluation of the model

Source projects	Techniques	Top 5 (%)	Top 10 (%)	MRR	MAP
SWT	BugLocator [20]	69.30	79.30	50.20	44.50
	BLUiR [39]	75.00	86.00	66.00	58.00
	BRTracer [35]	79.60	88.80	59.50	53.00
	The e-rVSM	80.20	89.10	80.00	61.22
Zxing	BugLocator [20]	60.00	70.00	50.00	44.00
	BLUiR [39]	64.50	70.00	49.00	39.00
	BRTracer [35]	N/A	N/A	N/A	N/A
	The e-rVSM	65.00	75.00	54.00	47.23

From Table 10, it can be concluded that e-rVSM developed with the combination of stack traces and spectrum information outperform BugLocator, BLUiR, and BRTracer. Fig. 8 depicts the graphical representation of the comparison.

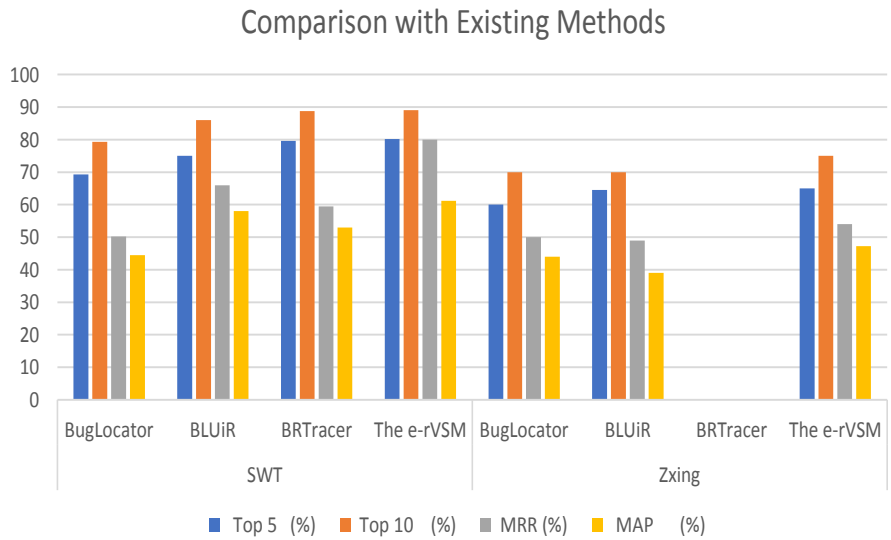


Fig. 8 Graphical illustration of performance of the proposed and existing methods

5. Conclusions and Future Work

This study proposes and develops a novel model for IR-based bug localization. The new model incorporates code coverage analysis to filter source code files and enhance the rVSM model with an automatic magnifier parameter to represent significant source code file scores appropriately. In addition, the accuracy of the IR model was boosted with the stack trace analysis and spectrum information. Findings from the experimental result of the proposed model on open source projects (SWT and Zxing) indicated the effectiveness of the proposed IR-based technique for bug localization. The successful combination of code coverage, stack trace analysis, and spectrum information improved the accuracy of bug localization. In addition, the code coverage analysis introduced before applying the IR technique helped reduce the search space and minimized the time for the bug localization process.

As a limitation of this study, it is imperative to explore and investigate other aspects of bug localization. For example, machine learning (ML) techniques and more enhancement features to the VSM model will be investigated. Also, more and different open source projects such as Eclipse and AspectJ would be used for experimentation.

References

[1] BALOGUN A. O., BASRI S., ABDULKADIR S. J. and HASHIM A. S. Performance analysis of feature selection methods in software defect prediction: a search method approach. *Applied Sciences*, 2019, 9(13): 2764. https://doi.org/10.3390/app9132764

[2] XIAO Y., KEUNG J., BENNIN K. E., and MI Q. Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology*, 2019, 105: 17-29 https://doi.org/10.1016/j.infsof.2018.08.002

[3] BALOGUN A. O., BASRI S., MAHAMAD S., CAPRETZ L. F., IMAM A. A., ALMOMANI M. A., ADEYEMO V. E., and KUMAR G. A Novel Rank Aggregation-Based Hybrid Multifilter Wrapper Feature Selection Method in Software Defect Prediction. *Computational Intelligence and Neuroscience*, 2021, 2021: 5069016. https://doi.org/10.1155/2021/5069016

[4] BETTENBURG N., PREMRAJ R., ZIMMERMANN T., and KIM S. Extracting structural information from bug reports. Proceedings of the 2008 international working conference on mining software repositories, New York, 2008, pp. 27-30. https://doi.org/10.1145/1370750.1370757

[5] NGUYEN A. T., NGUYEN T. T., AL-KOFAHI J., NGUYEN H. V., and NGUYEN T. N. A topic-based approach for narrowing the search space of buggy files from a bug report, Proceedings of 26th IEEE/ACM International Conference on Automated Software Engineering, Lawrence, 2011, pp. 263-272. https://doi.org/10.1109/ASE.2011.6100062

[6] SHI Z., KEUNG J., BENNIN K. E., and ZHANG X. Comparing learning to rank techniques in hybrid bug localization. *Applied Soft Computing*, 2018, 62: 636-648. https://doi.org/10.1016/j.asoc.2017.10.048

[7] MILLS C., PANTIUCHINA J., PARRA E., BAVOTA G., and HAIDUC S. Are bug reports enough for text retrieval-based bug localization? Proceedings of IEEE International Conference on Software Maintenance and Evolution, Madrid, 2018, pp. 381-392. https://doi.org/10.1109/ICSME.2018.00046

[8] HERBOLD S., TRAUTSCH A., and LEDEL B. Large-scale manual validation of bugfixing changes. Proceedings of the 17th International Conference on Mining Software Repositories, Pittsburgh, 2020, pp. 611-614. https://doi.org/10.1145/3379597.3387504

[9] PINGCLASAI N., HATA H., and MATSUMOTO K.-I. Classifying bug reports to bugs and other requests using topic modeling. Proceedings of 20th Asia-pacific software engineering conference, Bangkok, 2013, pp. 13-18. https://doi.org/10.1109/APSEC.2013.105

[10] GU Y., XUAN J., ZHANG H., ZHANG L., FAN Q., XIE X., and QIAN T. Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence. *Journal of Systems and Software*, 2019, 148: 88-104. https://doi.org/10.1016/j.jss.2018.11.004

[11] WANG Y., HUANG Z., FANG B., and LI Y. Spectrum-based fault localization via enlarging non-fault region to

improve fault absolute ranking. *IEEE Access*, 2018, 6: 8925-8933. https://doi.org/10.1109/ACCESS.2018.2796849
[12] KIM J., PARK J., and LEE E. A new hybrid algorithm for software fault localization. Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication, New York, 2015, pp. 1-8. https://doi.org/10.1145/2701126.2701207
[13] HASSELBRING W., CARR L., HETTRICK S., PACKER H., and TIROPANIS T. Open source research software. *Computer*, 2020, 53(8): 84-88. https://doi.ieeecomputersociety.org/10.1109/MC.2020.2998235
[14] SINGH S., & SINGH S. K. A novel approach for bug localization for Exception Handling and Multithreading through mutation. Proceedings of Annual IEEE India Conference, New Delhi, 2016, pp. 1-6 https://doi.org/10.1109/INDICON.2015.7443160
[15] KILINÇ D., YÜCALAR F., BORANDAĞ E., and ASLAN E. Multi-level reranking approach for bug localization. *Expert Systems*, 2016, 33(3): 286-294. https://doi.org/10.1111/exsy.12150
[16] ALAZZAWI A. K., RAIS H. M., BASRI S., ALSARIERA Y. A., CAPRETZ L. F., BALOGUN A. O., and IMAM A. A. HABCSm: A Hamming Based t-way Strategy based on Hybrid Artificial Bee Colony for Variable Strength Test Sets Generation. *International Journal of Computers Communications & Control*, 2021, 16(5). https://doi.org/10.48550/arXiv.2110.03728
[17] AMEEN A. O., MOJEED H., BOLARIWA A., BALOGUN A., MABAYOJE M., USMAN-HAMZAH F., ABDULRAHEEM M., and MOJEED H. A. Application of Shuffled Frog-Leaping Algorithm for Optimal Software Project Scheduling and Staffing. Proceedings of International Conference of Reliable Information and Communication Technology, Langkawi, 2020, pp. 293-303. https://doi.org/10.1007/978-3-030-70713-2_28
[18] VALDIVIA-GARCIA H., SHIHAB E., and NAGAPPAN M. Characterizing and predicting blocking bugs in open source projects. *Journal of Systems and Software*, 2018, 143: 44-58. https://doi.org/10.1016/j.jss.2018.03.053
[19] LUCIA L., LO D., JIANG L. THUNG F., and BUDI A. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process*, 2014, 26(2): 172-219. https://doi.org/10.1002/smr.1616
[20] ZHOU J., ZHANG H., and LO D. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. Proceedings of the 34th International Conference on Software Engineering, Zurich, 2012, pp. 14-24. https://doi.org/10.1109/ICSE.2012.6227210
[21] ANVIK J., HIEW L., and MURPHY G. C. Who should fix this bug. Proceedings of the 28th international conference on Software engineering, Pittsburgh, 2006, pp. 361-370. https://doi.org/10.1145/1134285.1134336
[22] MILLS C., PARRA E., PANTIUCHINA J., BAVOTA G., and HAIDUC S. On the relationship between bug reports and queries for text retrieval-based bug localization. *Empirical Software Engineering*, 2020, 25(5): 3086-3127. https://doi.org/10.1007/s10664-020-09823-w
[23] BALOGUN A. O., BASRI S., MAHAMAD S., ABDULKADIR S. J., ALMOMANI M. A., ADEYEMO V. E., AL-TASHI Q., MOJEED H. A., IMAM A. A., and BAJEH A. O. Impact of feature selection methods on the predictive performance of software defect prediction models: An extensive empirical study. *Symmetry*, 2020, 12(7): 1147. https://doi.org/10.3390/sym12071147
[24] WU R., WEN M., CHEUNG S.-C., and ZHANG H. Changelocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering*, 2018, 23(5): 2866-2900. https://doi.org/10.1007/s10664-017-9567-4
[25] ZHANG X., YAO Y., WANG Y., XU F., and LU J. Exploring metadata in bug reports for bug localization. Proceedings of 24th Asia-Pacific Software Engineering Conference, Nanjing, 2017, pp. 328-337. https://doi.org/10.1109/APSEC.2017.39
[26] LE T.-D. B., THUNG F., and LO D. Will this localization tool be effective for this bug? Mitigating the impact of unreliability of information retrieval based bug localization tools. *Empirical Software Engineering*, 2017, 22(4): 2237-2279. https://doi.org/10.1007/s10664-016-9484-y
[27] WANG Y., YAO Y., TONG H., HUO X., LI M., XU F., and LU J. Bug localization via supervised topic modeling. Proceedings of IEEE international conference on data mining, 2018, pp. 607-616. http://tonghanghang.org/pdfs/icdm2018_bug.pdf
[28] AKBAR S. A., & KAK A. C. A large-scale comparative evaluation of IR-based tools for bug localization. Proceedings of the 17th International Conference on Mining Software Repositories, Seoul, 2020, pp. 21-31. https://doi.org/10.1145/3379597.3387474
[29] FANG F., WU J., LI Y., YE X., ALJEDAANI W., and MKAOUER M. W. On the classification of bug reports to improve bug localization. *Soft Computing*, 2021, 25(11): 7307-7323. https://doi.org/10.1007/s00500-021-05689-2
[30] MURALI V., GROSS L., QIAN R., and CHANDRA S. Industry-scale IR-based Bug Localization: A Perspective from Facebook. Proceedings of IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice, Madrid, 2021, pp. 188-197. https://doi.org/10.48550/arXiv.2010.09977
[31] MORENO L., TREADWAY J. J., MARCUS A., and SHEN W. On the use of stack traces to improve text retrieval-based bug localization. Proceedings of IEEE International Conference on Software Maintenance and Evolution, Victoria, 2014, pp. 151-160. https://doi.org/10.1109/ICSME.2014.37
[32] RAO S., & KAK A. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. Proceedings of the 8th Working Conference on Mining Software Repositories, Pittsburgh, 2011, pp. 43-52. https://doi.org/10.1145/1985441.1985451
[33] RAHMAN S., GANGULY K. K., and SAKIB K. An improved bug localization using structured information retrieval and version history, Proceedings of 18th International Conference on Computer and Information Technology, Dhaka, 2015, pp. 190-195. https://doi.org/10.1109/ICCITechn.2015.7488066
[34] ZHAI C., COHEN W. W., and LAFFERTY J. Beyond independent relevance: methods and evaluation metrics for subtopic retrieval. *ACM SIGIR Forum*, 2015, 49(1): 2-9. https://doi.org/10.1145/2795403.2795405
[35] WONG C.-P., XIONG Y., ZHANG H., HAO D., ZHANG L., and MEI H. Boosting bug-report-oriented fault

localization with segmentation and stack-trace analysis. Proceedings of IEEE International Conference on Software Maintenance and Evolution, Victoria, 2014, pp. 181-190. https://doi.org/10.1109/ICSME.2014.40

[36] YOUM K. C., AHN J., and LEE E. Improved bug localization based on code change histories and bug reports. *Information and Software Technology*, 2017, 82: 177-192. https://doi.org/10.1016/j.infsof.2016.11.002

[37] DAO T., ZHANG L., and MENG N. How does execution information help with information-retrieval based bug localization. Proceedings of IEEE/ACM 25th International Conference on Program Comprehension, Buenos Aires, 2017, pp. 241-250. https://doi.org/10.1109/ICPC.2017.29

[38] THUNG F. Automatic prediction of bug fixing effort measured by code churn size. Proceedings of the 5th International Workshop on Software Mining, New York, 2016, pp. 18-23. https://doi.org/10.1145/2975961.2975964

[39] SAHA R. K., LEASE M., KHURSHID S., and PERRY D. E. Improving bug localization using structured information retrieval. Proceedings of 28th IEEE/ACM International Conference on Automated Software Engineering, Silicon Valley CA, 2013, pp. 345-355. https://doi.org/10.1109/ASE.2013.6693093

[40] POSHYVANYK D., GUÉHÉNEUC Y.-G., MARCUS A., ANTONIOL G., and RAJLICH V. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 2007, 33(6): pp. 420-432. https://doi.org/10.1109/TSE.2007.1016

[41] LUKINS S. K., KRAFT N. A., and ETZKORN L. H. Bug localization using latent dirichlet allocation. *Information and Software Technology,* 2010, 52(9): 972-990. https://doi.org/10.1016/j.infsof.2010.04.002

[42] JELODAR H., WANG Y., YUAN C., FENG X., JIANG X., LI Y., and ZHAO L. Latent Dirichlet allocation (LDA) and topic modeling: models, applications, a survey. *Multimedia Tools and Applications*, 2019, 78(11): 15169-15211. https://doi.org/10.48550/arXiv.1711.04305

[43] ARUN R., SURESH V., MADHAVAN C. V., and MURTHY M. N. On finding the natural number of topics with latent dirichlet allocation: Some observations. Proceedings of Pacific-Asia conference on knowledge discovery and data mining, Berlin, 2010, pp. 391-402. https://doi.org/10.1007/978-3-642-13657-3_43

[44] YE X., SHEN H., MA X., BUNESCU R., and LIU C. From word embeddings to document similarities for improved information retrieval in software engineering. Proceedings of the 38th international conference on software engineering, Austin, 2016, pp. 404-415. https://doi.org/10.1145/2884781.2884862

[45] TAKAHASHI A., SAE-LIM N., HAYASHI S., and SAEKI M. A preliminary study on using code smells to improve bug localization. Proceedings of the 26th Conference on Program Comprehension, Pittsburgh, 2018, pp. 324-327. https://doi.org/10.1145/3196321.3196361

[46] SANGLE S., MUVVA S., CHIMALAKONDA S., PONNALAGU K., and VENKOPARAO V. G. DRAST--A Deep Learning and AST Based Approach for Bug Localization. *arXiv preprint*, 2020: 2011.03449. https://doi.org/10.48550/arXiv.2011.03449

[47] MAHAJAN G., & CHAUDHARY N. Improving Bug Localization using IR-based Textual Similarity and Vectorization Scoring Framework. *International Journal of Advances in Soft Computing & Its Applications*, 2020, 12(2): 23-32. https://www.semanticscholar.org/paper/Improving-Bug-Localization-using-IR-based-Textual-Mahajan-Chaudhary/c7e5e583304d5b9694af2a15a38cddf4db68fedf

[48] QIU F., YAN M., XIA X., WANG X., FAN Y., HASSAN A. E., and LO D. JITO: a tool for just-in-time defect identification and localization. Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 2020, pp. 1586-1590. https://ink.library.smu.edu.sg/sis_research/5537

[49] CHENG S., YAN X., and KHAN A. A. A Similarity Integration Method based Information Retrieval and Word Embedding in Bug Localization. Proceedings of 20th International Conference on Software Quality, Reliability and Security, Macau, 2020, pp. 180-187. https://doi.org/10.1109/QRS51102.2020.00034

[50] XIAO Y., KEUNG J., MI Q., and BENNIN K. E. Bug localization with semantic and structural features using convolutional neural network and cascade forest. Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering, New York, 2018, pp. 101-111. https://doi.org/10.1145/3210459.3210469

[51] TANTITHAMTHAVORN C., ABEBE S. L., HASSAN A. E., IHARA A., and MATSUMOTO K. The impact of IR-based classifier configuration on the performance and the effort of method-level bug localization. *Information and Software Technology*, 2018, 102: 160-174. https://doi.org/10.1016/j.infsof.2018.06.001

[52] LAM A. N., NGUYEN A. T., NGUYEN H. A., and NGUYEN T. N. Bug localization with combination of deep learning and information retrieval. Proceedings of IEEE/ACM 25th International Conference on Program Comprehension, Buenos Aires, 2017, pp. 218-229. https://doi.org/10.1109/ICPC.2017.24

[53] LOYOLA P., GAJANANAN K., and SATOH F. Bug localization by learning to rank and represent bug inducing changes. Proceedings of the 27th ACM International Conference on Information and Knowledge Management, Torino, 2018, pp. 657-665. https://doi.org/10.1145/3269206.3271811

参考文:

[1] BALOGUN A. O., BASRI S., ABDULKADIR S. J. 和 HASHIM A. S. 软件缺陷预测中特征选择方法的性能分析：一种搜索方法。应用科学，2019，9(13)：2764. https://doi.org/10.3390/app9132764

[2] XIAO Y., KEUNG J., BENNIN K. E., 和 MI Q. 通过词嵌入和增强的卷积神经网络改进错误定位。信息和软件技术，2019，105：17-29 https://doi.org/10.1016/j.infsof.2018.08.002

[3] BALOGUN A. O., BASRI S., MAHAMAD S., CAPRETZ L. F., IMAM A. A., ALMOMANI M. A., ADEYEMO V. E., 和 KUMAR G. 软件缺陷预测中一种新的基于秩聚合的混合多过滤器包装特征选择方法。计算

智能和神经科学，2021，2021：5069016. https://doi.org/10.1155/2021/5069016
[4] BETTENBURG N., PREMRAJ R., ZIMMERMANN T., 和 KIM S. 从错误报告中提取结构信息。2008 年采矿软件存储库国际工作会议论文集，纽约，2008 年，第 27-30 页. https://doi.org/10.1145/1370750.1370757
[5] NGUYEN A. T., NGUYEN T. T., AL-KOFAHI J., NGUYEN H. V., 和 NGUYEN T. N. 一种基于主题的方法，用于从错误报告中缩小错误文件的搜索空间，第 26 届电气和电子工程师学会/计算机协会自动化软件工程国际会议论文集，劳伦斯，2011 年，第 263-272 页. https://doi.org/10.1109/ASE.2011.6100062
[6] SHI Z., KEUNG J., BENNIN K. E., 和 ZHANG X. 比较混合错误定位中的学习排名技术。应用软计算，2018，62：636-648. https://doi.org/10.1016/j.asoc.2017.10.048
[7] MILLS C., PANTIUCHINA J., PARRA E., BAVOTA G., 和 HAIDUC S. 错误报告是否足以用于基于文本检索的错误本地化？电气和电子工程师协会国际软件维护和演进会议论文集，马德里，2018，pp. 381-392. https://doi.org/10.1109/ICSME.2018.00046
[8] HERBOLD S., TRAUTSCH A., 和 LEDEL B. 错误修复更改的大规模手动验证。第 17 届国际挖掘软件存储库会议论文集，匹兹堡，2020 年，第 611-614 页. https://doi.org/10.1145/3379597.3387504
[9] PINGCLASAI N., HATA H., 和 MATSUMOTO K.-I. 使用主题建模将错误报告分类为错误和其他请求。第 20 届亚太软件工程会议论文集，曼谷，2013 年，第 13-18 页. https://doi.org/10.1109/APSEC.2013.105
[10] GU Y., XUAN J., ZHANG H., ZHANG L., FAN Q., XIE X., 和 QIAN T. 故障是否存在于堆栈跟踪中？通过预测碰撞故障的位置来辅助碰撞定位。系统与软件杂志，2019，148：88-104. https://doi.org/10.1016/j.jss.2018.11.004
[11] WANG Y., HUANG Z., FANG B., 和 LI Y. 基于频谱的故障定位通过扩大非故障区域来提高故障绝对排序。电气和电子工程师协会访问，2018，6：8925-8933. https://doi.org/10.1109/ACCESS.2018.2796849
[12] KIM J., PARK J., 和 LEE E. 一种新的软件故障定位混合算法。第九届泛在信息管理与通信国际会议论文集，纽约，2015，第 1-8 页. https://doi.org/10.1145/2701126.2701207
[13] HASSELBRING W., CARR L., HETTRICK S., PACKER H., 和 TIROPANIS T. 开源研究软件。电脑，2020，53(8)：84-88. https://doi.ieeecomputersociety.org/10.1109/MC.2020.2998235
[14] SINGH S., 和 SINGH S. K. 一种通过突变进行异常处理和多线程错误定位的新方法。印度电气和电子工程师学会年度会议论文集，新德里，2016，pp. 1-6 https://doi.org/10.1109/INDICON.2015.7443160
[15] KILINÇ D., YÜCALAR F., BORANDAĞ E., 和 ASLAN E. 用于错误定位的多级重新排序方法。专家系统，2016，33(3)：286-294. https://doi.org/10.1111/exsy.12150
[16] ALAZZAWI A. K., RAIS H. M., BASRI S., ALSARIERA Y. A., CAPRETZ L. F., BALOGUN A. O., 和 IMAM A. A. 用于可变强度测试集生成的基于混合人工蜂群的基于汉明的吨路策略。国际计算机通信与控制杂志，2021，16(5). https://doi.org/10.48550/arXiv.2110.03728
[17] AMEEN A. O., MOJEED H., BOLARIWA A., BALOGUN A., MABAYOJE M., USMAN-HAMZAH F., ABDULRAHEEM M., 和 MOJEED H. A. 洗牌蛙跳算法在优化软件项目调度和人员配置中的应用。国际可靠信息和通信技术会议论文集，兰卡威，2020 年，第 293-303 页. https://doi.org/10.1007/978-3-030-70713-2_28
[18] VALDIVIA-GARCIA H., SHIHAB E., 和 NAGAPPAN M. 表征和预测开源项目中的阻塞错误。系统与软件杂志，2018，143：44-58. https://doi.org/10.1016/j.jss.2018.03.053
[19] LUCIA L., LO D., JIANG L. THUNG F., 和 BUDI A. 故障定位关联度量的扩展综合研究。软件杂志：进化与过程，2014，26(2)：172-219. https://doi.org/10.1002/smr.1616
[20] ZHOU J., ZHANG H., 和 LO D. 应该在哪里修复错误？基于错误报告的更准确的基于信息检索的错误定位。第 34 届国际软件工程会议论文集，苏黎世，2012 年，第 14-24 页. https://doi.org/10.1109/ICSE.2012.6227210
[21] ANVIK J., HIEW L., 和 MURPHY G. C. 谁应该修复这个错误。第 28 届国际软件工程会议论文集，匹兹堡，2006 年，第 361-370 页. https://doi.org/10.1145/1134285.1134336
[22] MILLS C., PARRA E., PANTIUCHINA J., BAVOTA G., 和 HAIDUC S. 基于文本检索的错误本地化的错误报告和查询之间的关系。实证软件工程，2020，25(5)：3086-3127. https://doi.org/10.1007/s10664-020-09823-w
[23] BALOGUN A. O., BASRI S., MAHAMAD S., ABDULKADIR S. J., ALMOMANI M. A., ADEYEMO V. E., AL-TASHI Q., MOJEED H. A., IMAM A. A., 和 BAJEH A. O. 特征选择方法对软件缺陷预测模型预测性能的影响：一项广泛的实证研究。对称，2020，12(7)：1147. https://doi.org/10.3390/sym12071147
[24] WU R., WEN M., CHEUNG S.-C., 和 ZHANG H. 变更定位器：根据崩溃报告定位导致崩溃的更改。实证软件工程，2018，23(5)：2866-2900. https://doi.org/10.1007/s10664-017-9567-4
[25] ZHANG X., YAO Y., WANG Y., XU F., 和 LU J. 探索

错误报告中的元数据以进行错误本地化。第 24 届亚太软件工程会议论文集，南京，2017，第 328-337 页. https://doi.org/10.1109/APSEC.2017.39

[26] LE T.-D. B., THUNG F., 和 LO D. 这个本地化工具对这个错误有效吗？减轻基于信息检索的错误定位工具不可靠性的影响。实证软件工程, 2017, 22(4): 2237-2279. https://doi.org/10.1007/s10664-016-9484-y

[27] WANG Y., YAO Y., TONG H., HUO X., LI M., XU F., 和 LU J. 通过有监督的主题建模进行错误定位。电气和电子工程师协会数据挖掘国际会议论文集, 2018, 第 607-616 页. http://tonghanghang.org/pdfs/icdm2018_bug.pdf

[28] AKBAR S. A., 和 KAK A. C. 基于信息检索的错误定位工具的大规模比较评估。第 17 届国际采矿软件存储库会议论文集，首尔，2020, 第 21-31 页. https://doi.org/10.1145/3379597.3387474

[29] FANG F., WU J., LI Y., YE X., ALJEDAANI W., 和 MKAOUER M. W. 关于漏洞报告的分类以改进漏洞本地化。软计算，2021, 25(11): 7307-7323. https://doi.org/10.1007/s00500-021-05689-2

[30] MURALI V., GROSS L., QIAN R., 和 CHANDRA S. 基于行业规模信息可检索的错误本地化：来自脸书的观点。电气和电子工程师协会/计算机机械协会第 43 届软件工程国际会议论文集：实践中的软件工程，马德里, 2021, 第 188-197 页. https://doi.org/10.48550/arXiv.2010.09977

[31] MORENO L., TREADWAY J. J., MARCUS A., 和 SHEN W. 关于使用堆栈跟踪来改进基于文本检索的错误定位。电气和电子工程师学会国际软件维护和演进会议论文集，维多利亚，2014 年，第 151-160 页. https://doi.org/10.1109/ICSME.2014.37

[32] RAO S., 和 KAK A. 从软件库中检索错误定位：通用和复合文本模型的比较研究。第八届挖掘软件存储库工作会议论文集，匹兹堡，2011 年，第 43-52 页. https://doi.org/10.1145/1985441.1985451

[33] RAHMAN S., GANGULY K. K., 和 SAKIB K. 使用结构化信息检索和版本历史改进的错误定位，第 18 届计算机和信息技术国际会议论文集，达卡，2015 年，第 190-195 页. https://doi.org/10.1109/ICCITechn.2015.7488066

[34] ZHAI C., COHEN W. W., 和 LAFFERTY J. 超越独立相关性：子主题检索的方法和评估指标。信息检索特别兴趣小组（计算机协会）论坛, 2015, 49(1): 2-9. https://doi.org/10.1145/2795403.2795405

[35] WONG C.-P., XIONG Y., ZHANG H., HAO D., ZHANG L., 和 MEI H. 通过分段和堆栈跟踪分析促进面向错误报告的故障定位。电气和电子工程师学会国际软件维护和演进会议论文集，维多利亚，2014 年，第 181-190 页. https://doi.org/10.1109/ICSME.2014.40

[36] YOUM K. C., AHN J., 和 LEE E. 基于代码更改历史和错误报告改进了错误本地化。信息和软件技术, 2017, 82: 177-192. https://doi.org/10.1016/j.infsof.2016.11.002

[37] DAO T., ZHANG L., 和 MENG N. 执行信息如何帮助基于信息检索的错误本地化。电气和电子工程师协会/计算机协会第 25 届程序理解国际会议论文集，布宜诺斯艾利斯，2017 年，第 241-250 页. https://doi.org/10.1109/ICPC.2017.29

[38] THUNG F. 自动预测由代码流失大小衡量的错误修复工作量。第五届国际软件挖掘研讨会论文集，纽约，2016 年，第 18-23 页. https://doi.org/10.1145/2975961.2975964

[39] SAHA R. K., LEASE M., KHURSHID S., 和 PERRY D. E. 使用结构化信息检索改进错误定位。第 28 届电气和电子工程师协会/计算机协会自动化软件工程国际会议论文集，加利福尼亚州硅谷, 2013, 第 345-355 页. https://doi.org/10.1109/ASE.2013.6693093

[40] POSHYVANYK D., GUÉHÉNEUC Y.-G., MARCUS A., ANTONIOL G., 和 RAJLICH V. 使用基于执行场景和信息检索的方法的概率排序进行特征定位。电气和电子工程师学会软件工程汇刊，2007，33(6)：第 420-432 页. https://doi.org/10.1109/TSE.2007.1016

[41] LUKINS S. K., KRAFT N. A., 和 ETZKORN L. H. 使用潜在狄利克雷分配的错误定位。信息和软件技术, 2010, 52(9): 972-990. https://doi.org/10.1016/j.infsof.2010.04.002

[42] JELODAR H., WANG Y., YUAN C., FENG X., JIANG X., LI Y., 和 ZHAO L. 潜在狄利克雷分配和主题建模：模型、应用、调查。多媒体工具和应用程序, 2019, 78(11): 15169-15211. https://doi.org/10.48550/arXiv.1711.04305

[43] ARUN R., SURESH V., MADHAVAN C. V., 和 MURTHY M. N. 关于通过潜在狄利克雷分配找到主题的自然数：一些观察。亚太知识发现和数据挖掘会议论文集，柏林，2010 年，第 391-402 页. https://doi.org/10.1007/978-3-642-13657-3_43

[44] YE X., SHEN H., MA X., BUNESCU R., 和 LIU C. 从词嵌入到文档相似性，以改进软件工程中的信息检索。第 38 届国际软件工程会议论文集，奥斯汀，2016 年，第 404-415 页. https://doi.org/10.1145/2884781.2884862

[45] TAKAHASHI A., SAE-LIM N., HAYASHI S., 和 SAEKI M. 使用代码异味改善错误定位的初步研究。第 26 届程序理解会议论文集，匹兹堡，2018 年，第 324-327 页. https://doi.org/10.1145/3196321.3196361

[46] SANGLE S., MUVVA S., CHIMALAKONDA S., PONNALAGU K., 和 VENKOPARAO V. G. 荒漠化风险评估支持工具 - 基于深度学习和抽象语法树的错误定位方法。档案预印本，2020: 2011.03449. https://doi.org/10.48550/arXiv.2011.03449

[47] MAHAJAN G., 和 CHAUDHARY N. 使用基于信息可检索的文本相似性和矢量化评分框架改进错误定位。国际软计算及其应用进展杂志，2020, 12(2): 23-32. https://www.semanticscholar.org/paper/Improving-Bug-Localization-using-IR-based-Textual-Mahajan-Chaudhary/c7e5e583304d5b9694af2a15a38cddf4db68fedf

[48] QIU F., YAN M., XIA X., WANG X., FAN Y., HASSAN A. E., 和 LO D. 即时缺陷识别和定位的工具。第 28 届欧洲软件工程会议和软件工程基础研讨会的计算机协会联合会议论文集，新加坡，2020 年，第 1586-1590 页. https://ink.library.smu.edu.sg/sis_research/5537

[49] CHENG S., YAN X., 和 KHAN A. A. 一种基于信息检索和词嵌入的相似性集成方法在错误定位中。第 20 届软件质量、可靠性和安全国际会议论文集，澳门, 2020, 第 180-187 页. https://doi.org/10.1109/QRS51102.2020.00034

[50] XIAO Y., KEUNG J., MI Q., 和 BENNIN K. E. 使用卷积神经网络和级联森林进行具有语义和结构特征的错误定位。第 22 届软件工程评估与评估国际会议论文集，纽约，2018 年，第 101-111 页. https://doi.org/10.1145/3210459.3210469

[51] TANTITHAMTHAVORN C., ABEBE S. L., HASSAN A. E., IHARA A., 和 MATSUMOTO K. 基于信息可检索的分类器配置对方法级错误定位的性能和工作量的影响。信息和软件技术，2018, 102: 160-174. https://doi.org/10.1016/j.infsof.2018.06.001

[52] LAM A. N., NGUYEN A. T., NGUYEN H. A., 和 NGUYEN T. N. 结合深度学习和信息检索的错误定位。电气和电子工程师协会/计算机协会第 25 届程序理解国际会议论文集，布宜诺斯艾利斯，2017 年，第 218-229 页. https://doi.org/10.1109/ICPC.2017.24

[53] LOYOLA P., GAJANANAN K., 和 SATOH F. 通过学习排名和表示引起错误的变化来定位错误。第 27 届计算机协会国际信息与知识管理会议论文集，都灵，2018 年，第 657-665 页. https://doi.org/10.1145/3269206.3271811