*big data and cognitive computing*

*Article*

# Locating Source Code Bugs in Software Information Systems Using Information Retrieval Techniques

**Ali Alawneh [1,*], Iyad M. Alazzam [2] and Khadijah Shatnawi [3]**

1 Management Information Systems Department, Faculty of Information Technology, Philadelphia University, Amman 19392, Jordan
2 Information Systems Department, Faculty of Information Technology and Computer Science, Yarmouk University, Irbid 21163, Jordan
3 Computer Information Systems Department, Faculty of Computer and Information Technology, Jordan University of Science and Technology, Irbid 22110, Jordan
* Correspondence: aalawneh@philadelphia.edu.jo

**Abstract:** Bug localization is the process through which the buggy source code files are located regarding a certain bug report. Bug localization is an overwhelming and time-consuming process. Automating bug localization is the key to help developers and increase their productivities. Expanding bug reports with more semantic and increasing software understanding using information retrieval and natural language techniques will be the way to locate the buggy source code file, in which the bug report works as a query and source code as search space. This research investigates the effect of segmenting open source files into executable code and comments, as they have a conflicting nature, seeks the effect of synonyms on the accuracy of bug localization, and examines the effect of "part-of-speech" techniques on reducing the manual inspection for appropriate synonyms. This research aims to approve that such methods improve the accuracy of bug localization tasks. The used approach was evaluated on three Java open source software, namely Eclipse 3.1, AspectJ 1.0, and SWT 3.1; we implement our dedicated Java tool to adopt our methodology and conduct several experiments on each software. The experimental results reveal a considerable improvement in recall and precision levels, and the developed methods display an accuracy improvement of 4–10% compared with the state-of-the-art approaches.

**Keywords:** software system; source code; bug localization; natural language understanding; information retrieval; VSM; IR; TF-IDF; bug report

**Citation:** Alawneh, A.; Alazzam, I.M.; Shatnawi, K. Locating Source Code Bugs in Software Information Systems Using Information Retrieval Techniques. *Big Data Cogn. Comput.* **2022**, *6*, 156. https://doi.org/10.3390/bdcc6040156

Academic Editors: Paweł Weichbroth, Jolanta Kowal and Mieczysław Lech Owoc

Received: 30 September 2022
Accepted: 21 November 2022
Published: 13 December 2022

## 1. Introduction

The rapid evolution in software systems, increasing uses, and demand for software in our daily life create an imperative need to ensure software quality. Many activities are performed during the software development cycle to deliver software of high quality [1], where 50% of the developer's efforts are spent on programming and debugging the source code. In debugging, developers spend a lot of time to locate and determine the buggy source code and perform heavy work to fix them [2]. In the end, software systems are often shipped with defects (bugs) [3]. Software quality is related closely to bugs that may lead to unexpected behavior of the software components. Once an abnormal behavior is detected in a software project, bugs are reported directly in a document called a bug report (BR) [4]. In case of the waterfall software development model, the external risks are frequently reviewed and the internal risks are only reviewed once or twice on a yearly basis. On the other hand, the external risks in the case of agile software development model could be reviewed in every iteration and the internal risk once or twice per year. Then, the scope requirements are cleared to the appropriate requirements repository; S.A. Aljawarneh et al. (2017) [5].

Bug report (BR) documents may convey information that could be the starting point for fixing bugs, by determining the relevant buggy source code elements of the software applications. Bug localization refers to finding a buggy source code file relevant to a specific bug report [6]. Manual bug localization is a time-consuming and work-intensive activity, especially for large projects, where the number of files and reports is huge.

When errors occur, the users describe these errors using their terms, which may be written in poor quality; this will lead to a poor query that will cost developers time and efforts. The bug reports represent the query written by users using a different combination of terms, so the same query is represented in a different format. Additional semantics and rules are needed to understand the different query formats reported for the same bug, so the computer system understands these different queries and retrieves the buggy source files. Open source software contains many source files and each file consists of code and comment; comments involve a lot of code terms. Eventually, the maintenance of software will be improved, thus empowering the bug localization process. Comments are typically written by developers using natural language terms [7]. One of the main issues is automatically analyzing the technical terms used in comments.

Software comprehension is dedicated to understanding how different software artifacts interact with each other. To leverage software comprehension techniques, developers carefully elect identifying names for method, classes, variables, and other software elements (software artifacts) by following a systematic pattern of the "naming convention" used for the specific programming language. These systematic patterns convey natural languages terms that are also repeated within comment blocks [8]. To increase the accuracy of software exploration and analysis tools and improve the ability of such software to recommend related software fragments, we need to find the right way to deal with the content of source code. Many approaches are proposed in the literature to handle buggy source code; most of these focus on information retrieval (IR) methods (static approach), which are utilized to determine buggy source code files based on the bug report (BR) [2–4,9,10]. IR methods depend on the textual content of bug reports and source code files in order to automatically determine and rank the relevant buggy source files. IR-based bug localization methods are preferred for their low computational cost. Furthermore, IR methods are feasible at any phase of system development. Moreover, no information is needed about program execution, so IR methods are broadly exploited in bug localization studies [4], in which the bug reports are treated as the query and the buggy source file is treated as a document to be retrieved [6].

The main contribution of this research is to solve the irrelevant retrieval problem of buggy source code files. Thus, our methodology considers the source code segmentation process and seeks the effect of bug reports' synonyms and comments' synonyms on the bug localization. Our methodology considers part-of-speech (POS) tagging to enhance understanding of a bug report written using natural language, and especially to deteriorate the manual checkup for suitable synonyms. Some natural language processing techniques are utilized to formulate term vectors with valuable and relevant terms, thus improving the accuracy of bug localization. In addition, existing research methodologies in bug localization pay no attention to the textual nature of both bug reports and comments, as these files are written using natural language terms, one written by the developers and the other written by end users. According to this precarious nature, we adopt some of part of the speech tagging technique and synonyms to leverage the understanding of software and bug reports and reduce the manual efforts, in order to achieve better performance.

In this research, we concentrate on enhancing the quality of the bug report query by augmenting the bug report with more valuable terms; we implement an approach that tags each term in the query and comment files using the POS tagger, and then expand and augment them with synonyms generated using the WordNet database.

Because comments represent a rich source to understand software functionality and involve many code terms, each source file is segmented into comments and the executable source code itself without comments. The segmentation stage was applied on three Java open sources AspectJ 1.0, Eclipse 3.1, and SWT 3.1. The Java source code files are processed using our coded Java tool "SynPos 1.0". Using our tool, we extract and analyze different source code structures, such as class, method, and field names, from source files. The extracted structures are preprocessed into terms using a different combination of preprocessing techniques implemented in our tool. To enhance the comment understanding, we feed comment files with extra semantics using POS and synonyms.

Different combinations of preprocessing techniques are applied, including stop word removal to filter out those words without any significant meaning in English, text tokenization, and stemming. Text similarities for processed comment files, source code files, and bug reports are computed using the vector space model (VSM).

This research mainly addresses the following questions:

1. How well does the segmentation of the source code body and comments affect the bug localization process?
2. Does our developed POS method improve the bug localization process?
3. To what extent could the synonyms of bug reports and comments improve the bug localization process?
4. Does our developed approach improve the accuracy of bug localization?

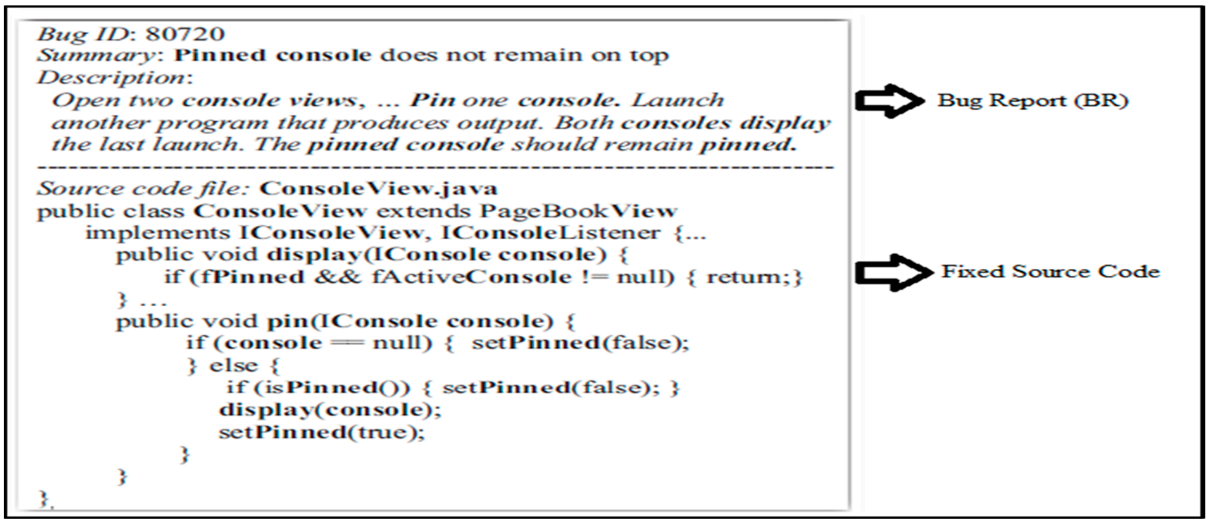
## 2. Theoretical Background

In software engineering, software comprehension is a crucial process for understanding how software artifacts (methods, classes, and fields) relate to each other. Software systems require continuous improvement and maintenance owing to emerging bugs reported by different users, locating the source fragments that cause such bugs, defined as bug localization. Understanding software structure and bug reports directly affects the performance of bug localization.

### *2.1. Bug Report*

A bug report is a textual document that contains many fields where information about the reported bug may be defined; this information includes a summary of the reported problem, an exhaustive description of the reported issue, the date of the observed issue, and the names of the buggy files modified to resolve the reported issue. Several studies stated that many BR terms also exist in the source code files [6].

Bug localization is a technique used to help developers to identify the faulty files related to the specific bug report. Figure 1 shows a bug report and fixed source file.



*Bug ID*: 80720
*Summary*: **Pinned console does not remain on top**
*Description*:
*Open two console views, ... Pin one console. Launch another program that produces output. Both consoles display the last launch. The pinned console should remain pinned.*

*Source code file:* **ConsoleView.java**

```
public class ConsoleView extends PageBookView
    implements IConsoleView, IConsoleListener {...
        public void display(IConsole console) {
            if (fPinned && fActiveConsole != null) { return;}
        } ...
        public void pin(IConsole console) {
            if (console == null) {  setPinned(false);
            } else {
                if (isPinned()) { setPinned(false); }
                display(console);
                setPinned(true);
            }
        }
},
```

**Figure 1.** Bug report example.

Figure 1 illustrates IR-based bug localization, where the bug report with ID: 80720 is a real bug reported for Eclipse 3.1. As the bug report is received, the developers struggle to locate the relevant buggy files among thousands of Eclipse source files to fix the reported bugs. It is obvious from Figure 1 that the bug report and source code share many words like pin (pinned), console, view, and display. Thus, we induce that the reported bug is about a console view. Indeed, Eclipse 3.1 contains a source code file named ConsoleView.Java, which also share similar words. Figure 1 demonstrates a worthy match between the bug report and the source code. For large software like Eclipse, thousands of buggy files are searched and ranked based on the textual similarity with the bug report, then the developers examine the ranked relevant buggy files until they fix the bug. The goal of bug localization is to retrieve the relevant buggy files to the suspected bug [3].

### 2.2. Bug Localization Techniques

Bug localization uses different techniques that are mainly divided into static and dynamic techniques:

- Static techniques: These depend on the structure of source code. These techniques assist the developer to locate an individual program element like classes and methods. The static approach can be performed at any development phase, and mainly depends on IR techniques that represent source code as a textual corpus to be searched against the query, which is a bug report.
- Dynamic techniques: This technique depends on program execution, so it is not applicable at any stage of software development as it needs runtime information such as execution traces [11], and thus more time and effort consumed to collect such data [9].

IR-based bug localization considers a static technique that was broadly exploited in bug localization, where the software's source code files denote the documents to be searched against a certain bug report, which denotes a search query. The similarities between various program units and the obtained bug report are calculated by IR-based bug localization. Then, owing to textual parallels to the bug report, program units and files are grouped in descending order. This recommended list of program files and units will help developers to locate the buggy source files and then developers will conduct the manual assessment [10].

### 2.3. Preprocessing Techniques

Extracted textual contents from bug reports and source files require preprocessing. Text preprocessing mainly aims to filter collected textual data from noise and extract the most valuable term that significantly affects the accuracy of the bug localization process, so bug localization is preceded by the text preprocessing phase, which maps the raw plain text of bug reports and source files into vectors of significant and valuable terms [12]. Some common text preprocessing techniques used by the literature include the following:

- Stop Words Removal: very frequent stop words like (the, a, and, or, and so on) are useless and add noise as they do not discriminate documents against each other, so stop words should be removed.
- Text Normalization: splitting text into tokens, for example, an identifier named "First-Variable" is tokenized to First and Variable according to camel case notation.
- Stemming: removing affixes and suffixes to extract the root of terms [13].

### 2.4. Information Retrieval Models

There are many IR models. We illustrate the most popular IR models; vector space model (VSM), latent semantic indexing (LSI), smoothed unigram model (SUM), and latent dirichlet allocation (LDA). A brief description of the previously mentioned models is presented below:

- Vector Space Model (VSM): represents queries and documents as vectors of terms' weights. The weight is usually expressed in terms of (TF-IDF) of the corresponding term. Term frequency (TF) indicates the number of term occurrences in the document, while inverse document frequency (IDF) indicates the number of documents that contain the term in a corpus. The higher the TF and IDF of a word, the more significant the term would be. Eventually, a higher weight will be assigned [12].
- Smoothed Unigram Model (SUM): is a probabilistic model that ranks the documents based on the probabilities to generate all query terms. This model is smoothed to alleviate zero probability for every document, so it is called the smoothed unigram model (SUM) [12].
- Latent Semantic Indexing Model (LSI): this model exploits singular value decomposition (SVD) to reduce dimensional space generated by documents from the term–document matrix, sometimes called LSA (latent semantic analysis) [12].
- Latent Dirichlet Allocation Model (LDA): is a probabilistic topic model that presents the documents and queries according to their topic. Each topic is denoted as a vector of terms, where each document is denoted as a vector of the topic [12].

## 3. Previous Works

This section presents previous research related to bug localization. The literature introduced bug localization from different perspectives; IR-based techniques, dynamic approaches, machine learning approaches, hybrid approaches, and approaches using synonyms.

### *Spectrum-Based Bug Localization (Dynamic Approaches)*

Dynamic, or spectrum-based, bug localization approaches depend on the statistics of program analysis information, semantics, and the execution trace (passing/failing test cases). This approach performs many experiments on software program to find differences between the control flows from passing and failing runs of the system under different input situations [4].

The authors of [14] proposed an approach namely "EnSpec" that performs entropy metrics to discriminate buggy code from non-buggy code, an extensive evaluation on "Defects4J" and "ManyBugs" performed, and higher efficiency was achieved.

However, spectrum-based bug localization (dynamic approaches) is rarely explored in the literature because of its limitations and complexity related concerns; dynamic approaches rely on software execution, which is not applicable at any stage of software development, specifically in the case of a critical error that may extend several modules [4]; and spectrum techniques require intensive information about program execution, and it is not easy to collect such information [15].

While spectrum-based techniques consume a lot of time and effort, many researchers pay great attention to IR-based techniques, as they are simple and applicable at different phases of software development; moreover, they are favored for their low computational cost [3,10]. IR-based bug localization effectiveness is affected by many factors; adapted IR techniques like the vector space model (VSM), LSI, and LDA, as well as the source of information such as version history, source code structure, bug report structure, stack trace, and previous bug report, play a crucial role in the accuracy of bug localization.

Source code compromises a restricted structure and rich textual contents, and source code terms reveal a high degree of duplication. Moreover, the comments written in code with free natural language style require intensive effort to avoid term mismatch. From this point, IR methods tend to consider source code structure in their experiment to maintain a higher accuracy.

The authors of [4] proposed a paradigm based on information-theoretic IR methods that utilize semantic characteristics of textual content of source code; they apply "pointwise mutual information (PMI) "and "normalized Google distance (NGD)" to assess the semantic similarity between source code and bug report. Their experiment was performed on five bug localization datasets and they rely on VSM, JSM, and LSI to capture semantic and lexical similarity. Their method achieves a significant improvement in the bug localization process.

Rather than semantic code characteristic, the work of [10] presented the "BLUiR" tool. The BLUiR architecture is based on IR methods; it extracts and defines different code structures' (class, identifiers, comment, methods) names using an abstract syntax tree (AST), then text preprocessing techniques are applied to the extracted text, such as text normalization, stop word removal, and stemming using Indri toolkit, as well as TF-IDF embedded in Indri toolkit, exploited to calculate the similarity between BR and different source code artifacts. They perform their experiment on similar datasets using BugLocater [3], which consists of four popular open source projects: Eclipse 3.1, AspectJ 1.0, SWT 3.1, and ZXing 3.1. The comparison with BugLocater shows that BLUiR outperforms BugLocater (from 26% to 35% in terms of precision and from 32% to 42% in terms of recall).

On the other hand, the work of [16] navigated another code analysis technique by locating repeated code segments, called code clones. The researchers exploit existing code clones analysis tools such as CCFinderX. The generated result from CCFinderX is extended using LSI. Rhino, Eclipse, and Mozilla are used in conducting their experiment.

On the other side, researchers tend to explore bug reports that can represent a fat query. IR techniques automatically seek the relevant buggy code files against this fat query, then a rank is given according to the relevancy between source code documents and bug report query. Once a new bug is reported, developers deviate to fix the reported bug according to the ranked buggy code list. From the predictive power of the previously fixed bug, the authors of [3] proposed BugLocator using the revised vector space model (rVSM) to rank source files, and the calculated rank was adjusted according to a similar bug fixed previously. BugLocator was evaluated on four open source projects (Eclipse 3.1, AspectJ 1.0, SWT 3.1, and ZXing 3.1), and the results show the effectiveness of BugLocator.

In addition to the previous BugLocator tool, the authors of [11] developed two approaches—one named (SOURCE) that measures the similarity between source code and the description of the bug report, and the other named (BUG) that measures the similarity between bug report text and the text of the previously fixed bug report. Stop words removal and splitting compound term according to uppercase letter are performed in text preprocessing, but stemming is applied within the SOURCE approach only, as no enhancement is achieved with the BUG approach, thus (TF-IDF) is used to weight terms. The proposed approaches were evaluated on small to medium Java projects (ArgoUML, JabRef, jEdit, and muCommander). The proposed approaches increased the number of relevant files from 50% to 57%.

Likewise, the authors of [17] proposed a bug localization approach that compromises three components combining several textual aspects to localize relevant source files for each bug report: the token matching component, to find exact matches for tokens in bug reports and source files; the similarity-based component, to exploit rVSM to enhance score for extracted tokens; and finally a classification component that relies on previously fixed bug reports to score suspicious buggy source files according to existing fixed bug reports. The score from the three components is aggregated to rank source files for a certain bug report. The proposed approach is evaluated on two open source software projects (SWT and ZXing), along with comparisons against BugLocator and BLUiR. The result reveals that 80% of bugs ranked in the top 10.

It is considered that previous bug reports may provide a good indication to locate buggy source code, but bug reports earlier than 14 days add no value and will decrease the bug localization performance [6].

Many existing approaches represent source code files as one unit and bug report as one plain text, but this is not the case. BRs could be rich in information like stack traces, which stack the nested files executed as a result of the reported bug.

The usefulness of stack traces (STs) explored by [9] developed text retrieval techniques that utilize the utility of ST named "Lobster"; it considers the bug report and source files to measure textual similarity using text retrieval techniques, and ST is used to determine the dependency according to structural code components matching; the higher the total similarity, the more relevant it is to the source code. They evaluate their approach using 17 versions of 14 Java open-source software programs; according to the achieved improvement, they strongly advocate considering stack traces in bug localization.

The work of [18] demonstrated how the use of STs improves the performance of bug localization; the researchers proposed a method that depends on source code structures in addition to stack traces' analysis; the method includes treatment for each source file into its components and each component matched to a certain BR, then a component score and a length score are calculated. Furthermore, the ST is assessed to calculate an ST boost score. Consequently, the component with the highest score represents the relevant one for the BR. The derived score is then multiplied by the length score to extract the similarity score applied for each source file. Finally, the ST score is aggregated to the similarity score to compute the overall score for a given source file. Empirically, the proposed method improves the bug localization for all evaluated software programs (Eclipse 3.1, AspectJ 1.0, and SWT 3.1). The results outperform the BugLocater results (from 44.5% to 51.6% in terms of precision and from 50.2% to 58.2% in terms of recall).

Accurate bug localization is described to be problematic, as the large and diverse sources of information can have a significant impact on bug localization accuracy; one of these sources is version history, which could be used to predict the probability of bug proneness in the successive software versions. The authors of [15] incorporated version histories of a software project with IR techniques to determine the posterior probability of a buggy source file. They introduce one model based on bug histories and the other based on modification histories. The result shows an improvement of 30% in bug localization accuracy.

Moreover, the authors of [13] developed an IR-based bug localization technique, which considers source code structure and version history; frequently changed files are also considered in computing ranks for suggested files, aiming to generate a better score for buggy source code files, and both class and method names are considered in computing similarity against BR. The proposed approach combined with rVSM gains higher accuracy. They apply the proposed techniques on three OS software programs: SWT, ZXing, and Guava; the results show an improvement in the recall of 7% and 8% in precision against similar existing techniques.

However, version history older than 20 days deteriorates the performance. Moreover, a large software project has a long history that might be time-consuming [6].

To leverage the performance and accuracy of bug localization, researchers take advantage of co-operating diverse sources of information.

The work of [19] proposed a novel IR-based bug localization approach, named "Bug Localization using Integrated Analysis (BLIA)". BLIA integrates texts, stack traces, and comments in bug reports; moreover, structured information of source files and the source code change history are considered. The granularity of bug localization improved from the file level to the method level by utilizing the previous bug repository. BLIA was evaluated using three open source projects (AspectJ 1.0, SWT 3.1, and ZXing 3.1). They noticed that BLIA achieve considerable improvement over BugLocater and BLUiR (54% and 42%, respectively).

The work of [20] presented additional sources of information such as methods that were fixed in previous 'similar' bugs, methods placed in similar bug reports, incidences of previous bugs, and stack traces in the bug report. Their proposed approach showed an enhancement in recall from 2% to 25%.

Object-oriented source code comprises different granularities, where several classes include many methods, variables, and nested classes. A software tester is interested in fragments that cause the bugs; once the buggy unit of source code is detected, all embedded components must be fixed accordingly. Related to this perspective, researchers developed different approaches on different granularities; applying similar approaches at different granularity levels results in different accuracies.

Bug localization at the method level can lead software testers to the faulty method. The authors of [21] evaluated the performance of VSM, LSI, and LDA at the method level; similarly, the authors of [2] found that VSM satisfies the best top-k performance and the least required effort at the method level, whereas the authors of [6] conducted their experiments on the class level.

Bug localization is broadly studied by researchers using some popular IR models such as the vector space model (VSM), latent semantic indexing (LSI), and latent dirichlet allocation (LDA); the results show a considerable variation between these studies.

The researchers in [2] studied the effect of different IR model configurations on the top-k performance and effort required at the method level; they performed their experiments on two large scale software programs, Eclipse and Mozilla, using a large number of IR classifiers (3172) on 5266 bug report systems. They concluded that the performance of top-k is highly affected by the configuration of the classifier from 0.44% to 36% and the required efforts from 4395 to 50,000 LOC; they also found that similar top-k models required different efforts, and they found that VSM was the best top-k with the least effort.

The authors of [22] developed an LDA-based bug localization technique. The results revealed that the LDA-based technique often performs significantly compared with LSI-based techniques for most bugs.

Furthermore, the authors of [23] proposed an approach called DrewBL, which constructs semantic-VSM, then the relevancy between source files and BR is fed into a natural language analysis tool named word2vec. The experimental results reveal a 7.4% improvement over BugLocater.

Machine learning (ML) techniques have been adopted widely in bug localization. ML has left a clear fingerprint in many different fields that mainly focuses on developing techniques that enable computers to "learn" by extracting information from data automatically. The most commonly used ML algorithms for bug localization are deep neural network (DNN) [24] and recurrent neural network (RNN) [25].

Many researchers have utilized ML in the bug localization discipline; the authors of [25] developed BugTranslator based on recurrent neural network (RNN). Bug reports were encoded into various context vectors using one RNN, with another RNN used for decoding into code tokens of buggy source files. The experimental results state that BugTranslator outperforms many existing IR-based techniques and enhances the performance of bug localization.

The authors of [26] developed DeepLoc based on convolutional neural network (CNN), where DeepLoc has evaluated over 18,500 bug reports extracted from AspectJ 1.0, Eclipse 3.1, JDT, SWT 3.1, and Tomcat projects. According to the authors, DeepLoc outperformed several state-of-the-art approaches.

To improve the bug localization accuracy, some researchers proposed a hybrid approach from state-of-the-art bug localization approaches.

Some researchers chose to bridge the gap either by combining different approaches or by incorporating a diverse source of information, such as the approach of [27], who exploited deep neural network (DNN) related to bug tokens to code tokens combined with rVSM, in order to acquire textual similarity between bug reports and source files. The results show that IR-based and deep learning satisfy a higher degree of accuracy compared with the state-of-the-art approaches. An exact study was found in [24], in which a combination of DNN and rVSM is evaluated.

In response to the limitation stated either by static or dynamic bug localization techniques, the authors of [28] explored the effect of different pieces of execution information on three existing IR-based techniques: the baseline technique, BugLocator, and BLUiR. Whereas the work of [29] proposed an automated model that identifies and sorts the classes liable to a bug, the proposed model employed text mining approaches, a list of classes ranked using lexical similarity between bug reports and the API descriptions, to improve the accuracy; the non-dominant sorting genetic algorithm (NSGA-II) was performed and the model was evaluated on three Java open source applications, noticing that NSGA-II beats the traditional methods of bug localization.

Fixing the reported bug appropriately plays a major role in software development and maintenance. Once a defect has happened, the software users report the software defect as a bug report. The same defect may be expressed differently by various users; to eliminate the diversity between the same bug reports, understanding the bug report will be the key. Expanding bug reports with more valuable semantic is explored in the literature for a different purpose.

The authors of [30] proposed a method to deduct duplicate reports by leveraging bug report semantic; they used the semantic LDA topic model, then they applied preprocessing on the bug report, they added synonyms for bug report terms to leverage bug report semantic, and they evaluated their method on the bug reports of Eclipse. Their method shows an improvement in the recall of 10.53%.

A data sampling technique was presented by [31] to enhance code smell prediction. Their study focuses on data that are unbalanced. Their effort attempts to identify eight distinct categories of code smells using feature engineering and sampling methodologies. The key innovation in their work is the use of three separate naive Bayes classifiers to identify code smells. They observed that the Gaussian naive Bayes classifier outperformed the Bernoulli naive Bayes and multinomial naive Bayes classifiers in their assessment of 629 projects.

A mathematical model based on bug dependency was created by [32] to create software reliability growth models. They used entropy, a measure of the system's unpredictability and irregularity, to analyze the bug summary description and comments.

Bug triage and software defect prediction are both challenging issues. An island binary mothflame optimization (IsBMFO) base model, put forward by [33], separates the solution in the population into island-style subpopulations. Then, each island is handled separately. They chose IsBMFo as their feature selector and SVM, KNN, and naive Bayes as their classification classifiers. KNN and naive Bayes both fared worse than the SVM with IsBMFO.

On the same rhythm, supplementing source code with further semantic was the interest of [34]; they have presented an approach for feature localization that enriches the source code and query with synonyms generated by the WordNet database, but a manual exploration of the possible synonyms was conducted. They used LSI as a retrieval model; their approach evaluated using two open source software programs: Qt and Hippodraw. To evaluate their approach, they conduct four different experiments for each software based on 21 features; the presented approach proves that adding synonyms to source code significantly enhances the feature location process and achieves a higher level of precision and recall.

To wrap up, we have presented a comprehensive study of related works in bug localization techniques. It can be noticed that existing bug localization techniques conflict with each other and suffer from some deficiencies. In addition, the majority of these techniques do not consider the issue related to the nature of source code and the textual nature of bug reports and comments embedded within source code.

## 4. Methodology

In methodology section, Section 4.1, introduces the overall research design; Section 4.2 explains the selected dataset; Section 4.3 illustrates different research phases; Section 4.4 shows data analysis procedures; and, finally, Section 4.5 lists the software tools that will be used during experiments and analysis of the result. Figure 2 below presents the used methodology.

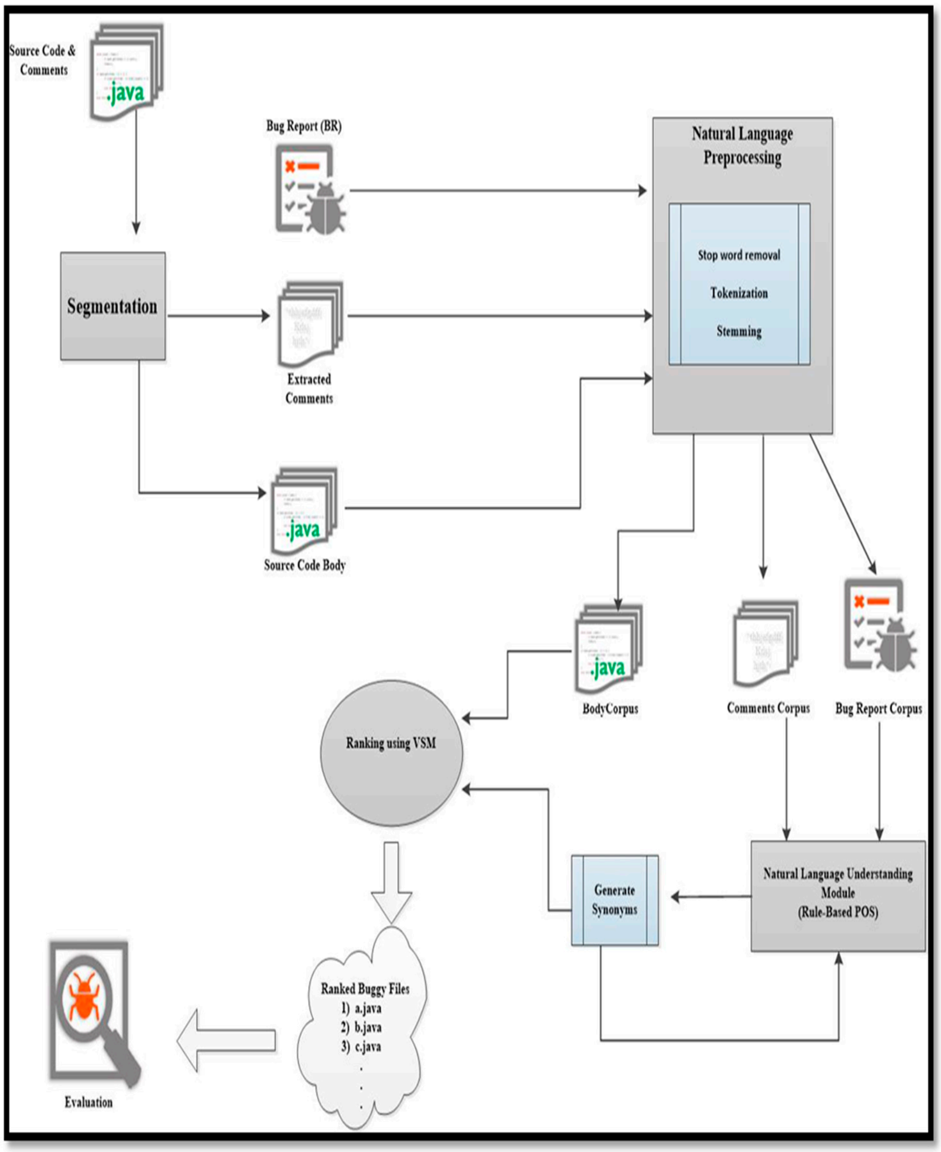


**Figure 2.** The study methodology.

## *4.1. Data Sampling*

This research depends on three public datasets that the authors of [3] used to evaluate BugLocator. This dataset involves three popular Java open source projects: Eclipse 3.1, AspectJ 1.0, and SWT 3.1.

- AspectJ 1.0, aspect-oriented programming (AOP): An aspect programming paradigm that aims to leverage modularity by aspect separation and adding extra behavior to specific code without code modification, using various extensions of Java programming language.
- Eclipse 3.1, integrated development environment (IDE): This is an integrated development environment (IDE) used widely in programming software applications and systems. Eclipse is written in Java and mainly used to develop powerful Java applications and systems. It offers the ability to develop other programming applications using customized plugins, including Ada, ABAP, C, C++, C#, Clojure, COBOL, D, Erlang, Fortran, Groovy, Haskell, JavaScript, and Julia.

SWT 3.1, graphical user interface (GUI): The Standard Widget Toolkit (SWT) is a graphical widget toolkit that is used with the Java platform. It was formerly developed by Stephen Northover at IBM and is now retained by the Eclipse Foundation in parallel with the Eclipse IDE. It is a substitute for the Abstract Window Toolkit (AWT). Table 1 is a summary table showing information about those datasets.

**Table 1.** Bug localization datasets.

| Dataset | Description | # of Source File | # of Bug Report |
|---|---|---|---|
| SWT | Java widget toolkit | 484 | 98 |
| AspectJ | Java aspect-oriented extension | 6485 | 286 |
| Eclipse | Open-source software for Java development | 12,836 | 3075 |

## *4.2. Converting Software and Bug Reports into XML Format*

We implement a Java module that converts source code and bug reports into XML format. Refs. [35,36] pointed out that XML is considered as a first step towards the Semantic Web vision; however, XML provides no semantic (meaning) to the data; a more structured XML format enables us to explore software and extract different source code artifacts (method, class, identifiers). Figure 3 represents the process of converting source files to XML.

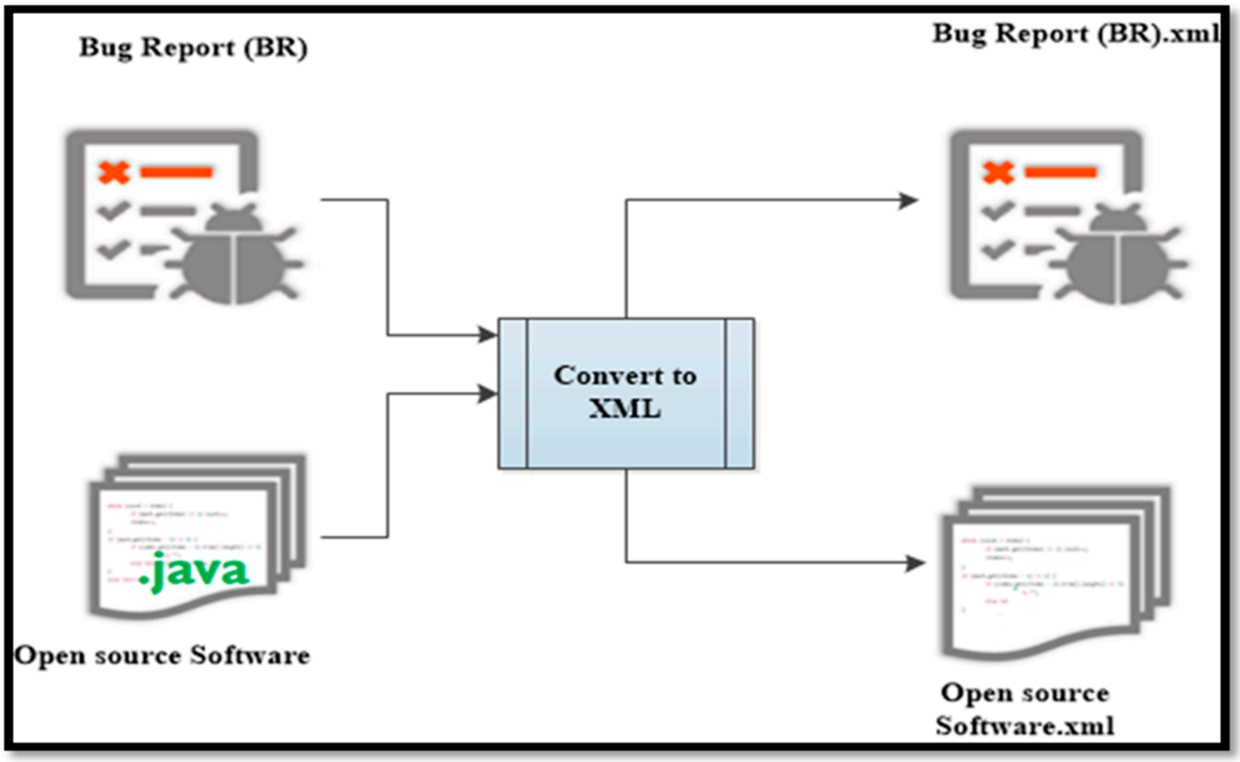
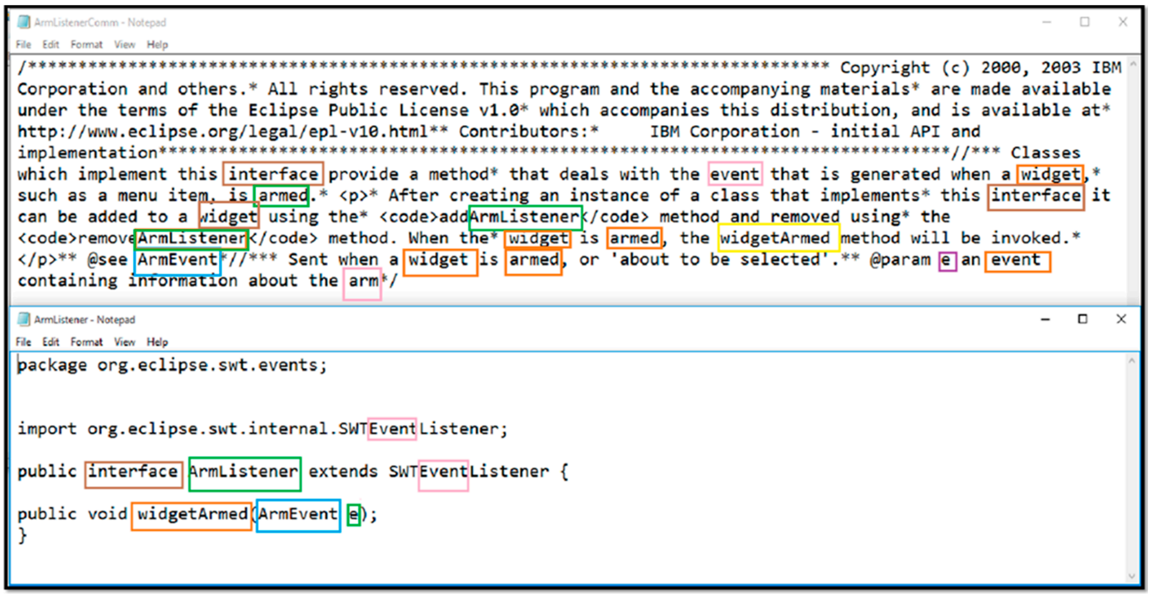


**Figure 3.** The process of conversion.

## *4.3. Source Code Segmentation*

By manually exploring one comment file and corresponding code file, which exist within SWT open source software named "ArmListener", we notice that identifier name, method name, classes, controls, and other software artifacts are listed within the comment block. Figure 4 shows terms that are common between code body and comment in source code file" ArmListener" from the SWT Project.



```
/******************************************************************************* Copyright (c) 2000, 2003 IBM
Corporation and others.* All rights reserved. This program and the accompanying materials* are made available
under the terms of the Eclipse Public License v1.0* which accompanies this distribution, and is available at*
http://www.eclipse.org/legal/epl-v10.html** Contributors:*     IBM Corporation - initial API and
implementation*******************************************************************************//*** Classes
which implement this interface provide a method* that deals with the event that is generated when a widget,*
such as a menu item, is armed.* <p>* After creating an instance of a class that implements* this interface it
can be added to a widget using the* <code>addArmListener</code> method and removed using* the
<code>removeArmListener</code> method. When the* widget is armed, the widgetArmed method will be invoked.*
</p>** @see ArmEvent*//*** Sent when a widget is armed, or 'about to be selected'.** @param e an event
containing information about the arm*/
```

```
package org.eclipse.swt.events;


import org.eclipse.swt.internal.SWTEventListener;

public interface ArmListener extends SWTEventListener {

public void widgetArmed(ArmEvent e);
}
```

**Figure 4.** The shared terms between code body and comment in the source code file "ArmListener".

Therefore, each source code file is divided into two segments; one for the source code body and the second for comments embedded within the source code. Two source code segments are ready to be processed using different natural language preprocessing techniques. Figure 5 shows the segmentation phase.

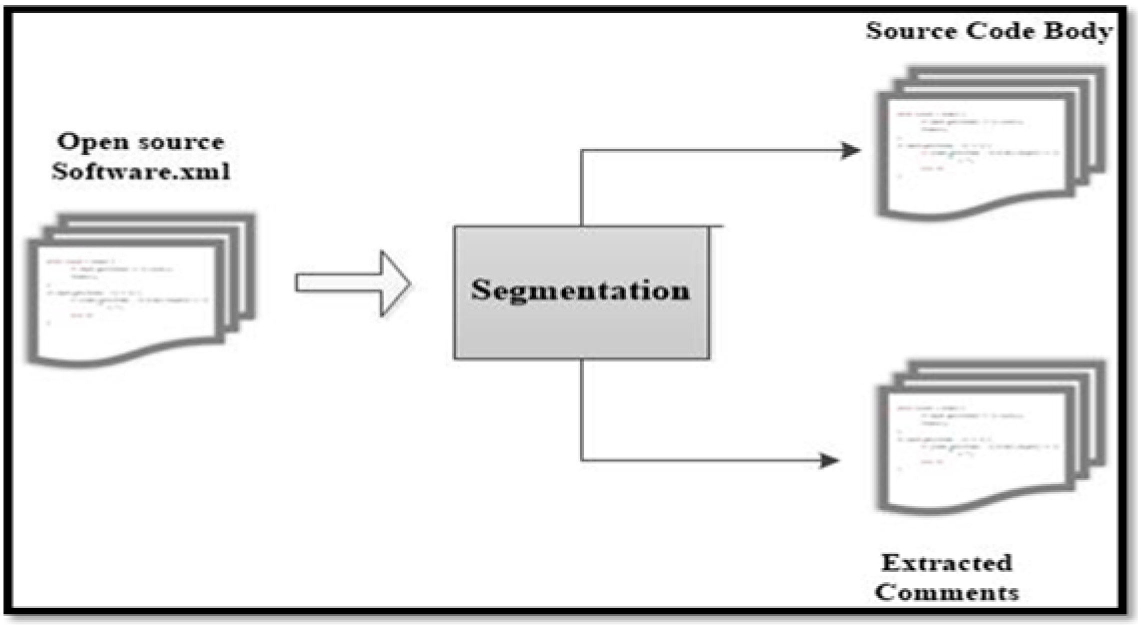


**Figure 5.** The segmentation phase.

### 4.3.1. Extracting Software Artifacts

To achieve better software understanding and increase the accuracy of software exploration and analysis tools that improve the ability of such software to recommend related software fragments, we need to extract different software artifacts (method, classes, variables), then the repository is populated with the extracted software artifacts. Each method in the source file is then mapped to a document. Figure 6 represents this phase.

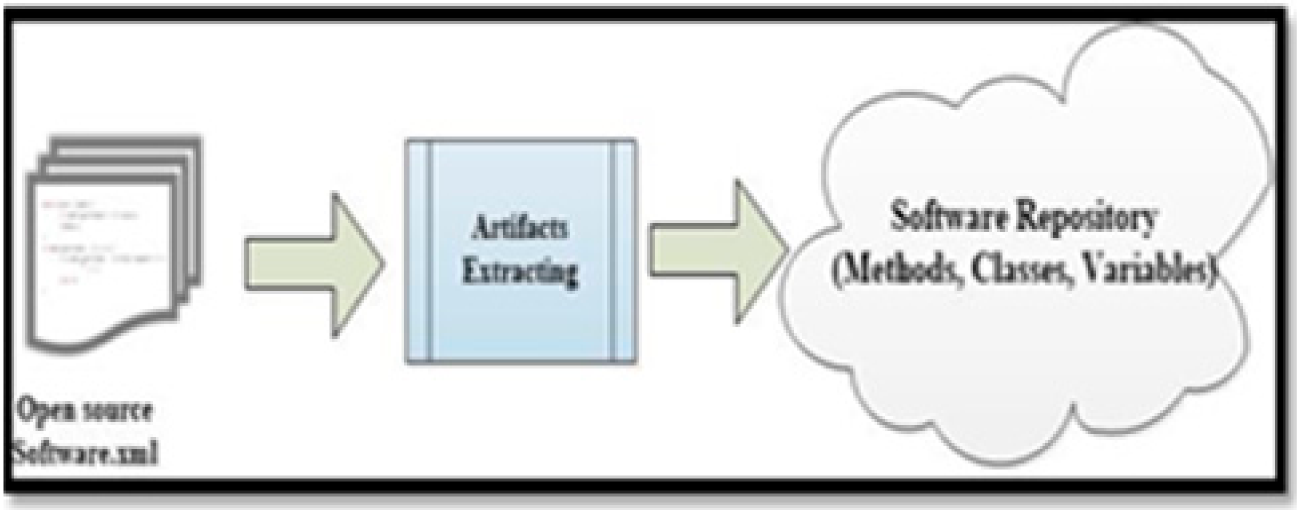


**Figure 6.** The extraction process of software artifacts.

### 4.3.2. Natural Language Preprocessing

The extracted documents incorporate a large volume of text, but not all text inside these documents is equally valuable. To increase the efficiency of the IR-based model, which is mainly affected by text preprocessing, we apply some text preprocessing techniques. In the preprocessing module, we mainly implement the three basic techniques: stop words removal, tokenization, and stemming.

- Stop words removal: these words deteriorate the understanding of documents' meaning and they are not valuable as index terms, so any stop words like articles and pronouns are removed. Moreover, as the programming language keywords, i.e., break for, char, and default, are only essential to run the program and do not provide any relevant information to the bug report, we remove Java keywords.
- Tokenization: every compound word in the documents is divided into its component; "ConsoleView" after tokenization will generate the console and view. Tokenization is necessary to increase the relevant valuable terms. Applying the tokenization process leverages the indexing process, and thus enables a higher match between the bug report and the document corresponding to that tokenized term; rather, there is no synonym generation tool can generate synonyms for non-tokenized terms.

Stemming: the most common preprocessing step. It is used to extract the root or base form of the word based on the commonly used algorithm; the Porter algorithm. Below are some of the rules used in the stemming process:

1. if the word ends in "ing", remove the "ing", for example, "processing" after stemming will be "process"
2. if the word ends in "ly", remove the "ly", for example, "frinedly" after stemming will be "friend"
3. if the word ends in "ed", remove the "ed", for example, "happened" after stemming will be "happen".

These preprocessing steps are applied to source codes, comments, and bug reports. After processing the source code files, comments files, and bug reports, a corpus that consists of a bag of valuable terms of these files is generated. Each method in the source code is mapped to a vector of the key terms. Figure 7 represents the preprocessing phase

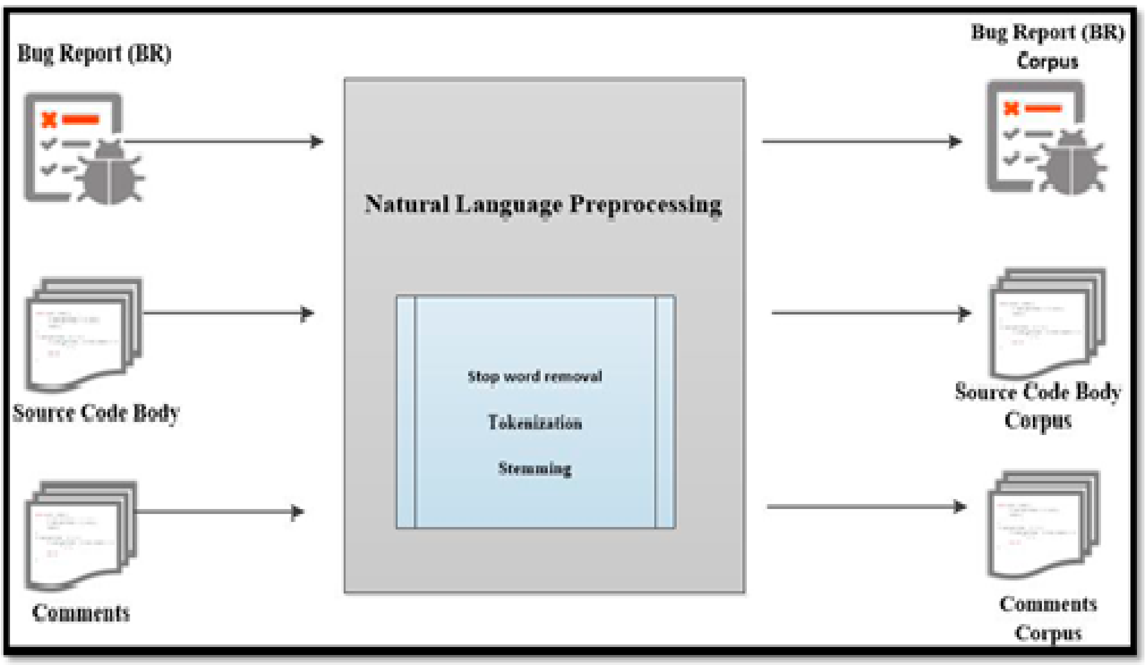


**Figure 7.** The preprocessing phase.

4.3.3. Natural Language Understanding (Part-of-Speech and Synonyms' Generation)

The bug report represents the query and these queries are written by the end user using their languages; on the other hand, comments are written by the developers using their terms—so the lexicons used by the users are close to the lexicons used by the developers, so some terms are common between comments and bug reports (query). Augmenting software comments and bug reports with more semantics will leverage bug localization effectiveness.

The input for this process is the processed bug reports and comments corpora, as the users' query (bug reports) describes the reported bugs using a different combination of terms, so the same query is represented in a different format. Some rules are needed to understand the different query formats reported for the same bug, so the computer system understands these different queries and retrieves the buggy source files, thus the bug localization accuracy is enhanced.

Bug reports' understanding is the key issue in bug localization; a bug report must be properly analyzed to explain what exactly is meant by such a query, thus the indicated buggy source code file could be retrieved, so a set of rules are developed for understanding reported bugs [37].

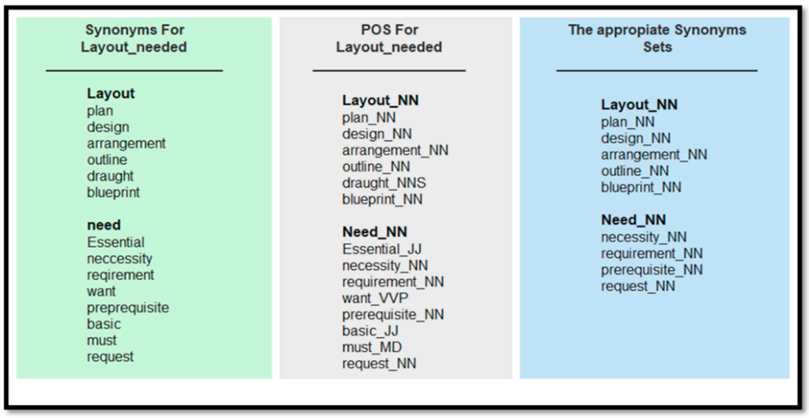
It is crucial to identify the type of bug report terms dealt with, especially nouns, so the bug report terms are processed to classify them into grammatical terms: noun, verb, and particle, using a part-of-speech tagger (POS tagger). In this research, we consider a rule-based tagger as an approach to build some rules and these are implemented in the POS module. Some of the manually developed contextual rules to understand the bug report are investigated, for example, the following:

Rule#1: if the bug description or summary contains noun + suffix ".Java", then the preceding noun will be highly recommended to be the buggy file name.
Rule#2: if the bug description or summary contains noun + (), then the preceding noun will be highly recommended to be the name of the method that causes such bugs.
Rule#3: if the bug description or summary contains the words "fetal error" + pronouns, then the following noun will be highly recommended to be the buggy file name.
Rule#4: if the bug description or summary contains the symbol "[" + noun + "]", then the noun involved within brackets will be highly recommended to be the buggy file/method name.

Rule#5: if the bug description or summary contains noun + symbol "#" + noun, then this noun will be highly recommended to be the buggy service name.
Rule#6: if the bug description or summary contains noun + symbol "/", then this will be highly recommended to be the path of the buggy source file.

In addition to performing a set of handwritten contextual rules; we exploit the generated POS tags to minimize the manual inspection for inappropriate synonyms generated by the WorldNet database linked in our dedicated Java tool. The authors of [34] adopted a manual examination for irrelevant synonyms in their work, which is not practical especially for a huge dataset like ours.

To deal with this problem, we only return synonyms of a word from WordNet that have the same POS. Figure 8 illustrates how we reduce the synonyms sets using POS.



| Synonyms For Layout_needed | POS For Layout_needed | The appropiate Synonyms Sets |
|---|---|---|
| **Layout** | **Layout_NN** | **Layout_NN** |
| plan | plan_NN | plan_NN |
| design | design_NN | design_NN |
| arrangement | arrangement_NN | arrangement_NN |
| outline | outline_NN | outline_NN |
| draught | draught_NNS | blueprint_NN |
| blueprint | blueprint_NN | |
| **need** | **Need_NN** | **Need_NN** |
| Essential | Essential_JJ | necessity_NN |
| neccessity | necessity_NN | requirement_NN |
| reqirement | requirement_NN | prerequisite_NN |
| want | want_VVP | request_NN |
| preprequisite | prerequisite_NN | |
| basic | basic_JJ | |
| must | must_MD | |
| request | request_NN | |

**Figure 8.** The reduction process of synonyms sets using POS.

As we notice in Figure 8," Layout_Needed" is an identifier name that exists in the widget file in the SWT dataset; after preprocessing, we get "layout, need". The set of probable synonyms sets of "layout" are "plan, design, arrangement, outline, draught, blueprint", and the possible synonyms set of "need" are "essential, necessity, requirement, want, prerequisite, basic, must, request". According to our method, we shrink the set to 11 possible synonyms instead of 16 (including the origin words). Figure 9 represents our method to shrink the synonyms sets.

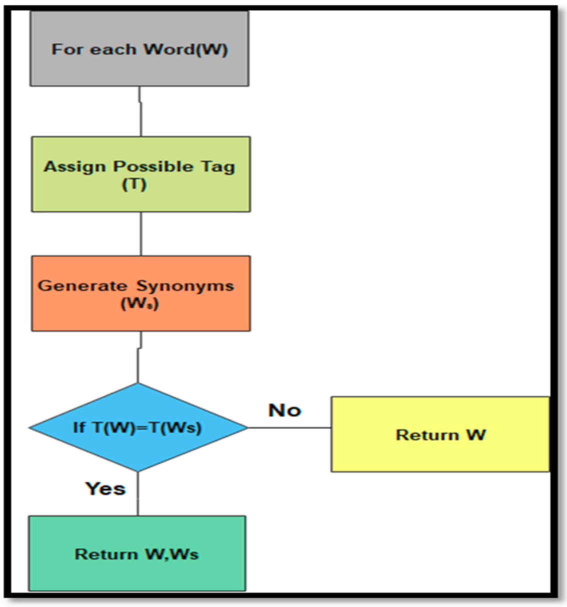


**Figure 9.** The shrinking method of the synonyms sets.

The valuable corpora (BR with synonyms, comments with synonyms, source code body) are ready for creating a vector of key terms. Figure 10 represents generating synonyms.

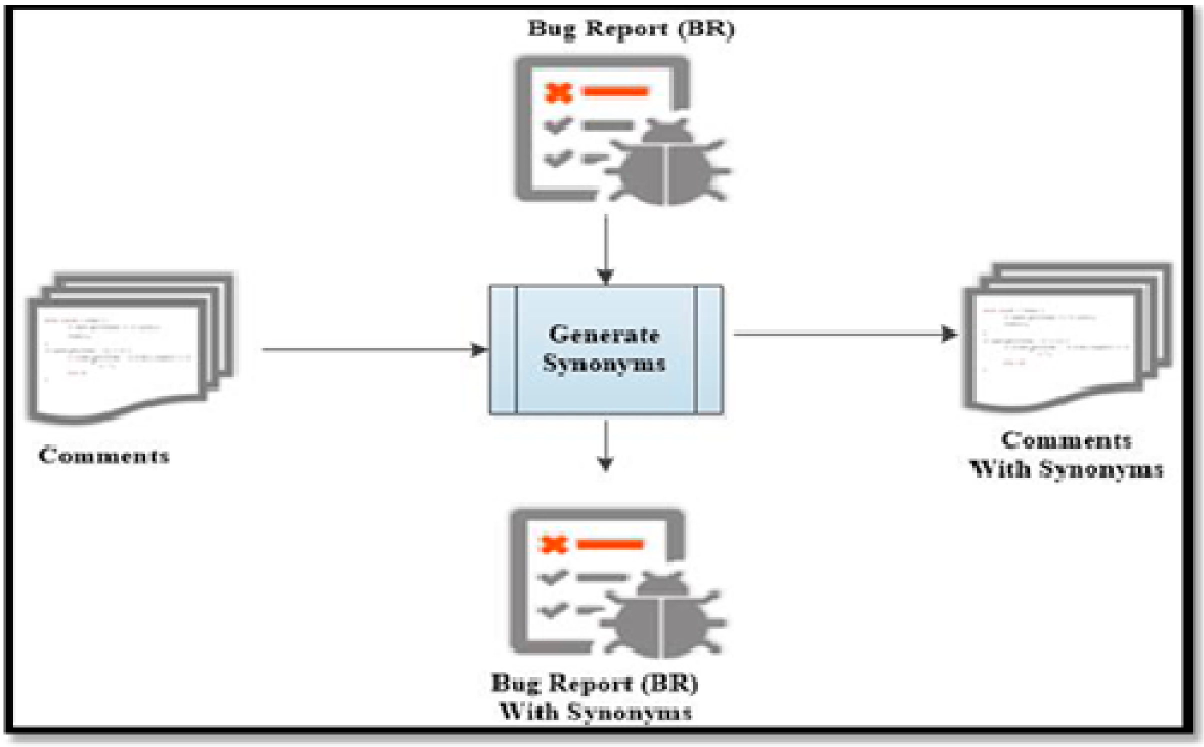


**Figure 10.** The process of generating synonyms.

### 4.3.4. Ranking and Retrieval Model

For fast and effective retrieval, the corpus is indexed using a text retrieval model, thus extracted terms from processed documents are ranked. The vector space model (VSM) is widely used in the literature and is a powerful model in locating buggy source code. VSM mapped each document to a vector of the key terms, and the weight is assigned using TF-IDF. In this research, we adopt the VSM to generate vectors of key terms. The three different vectors—the source code vector $\vec{S}$, comment vector $\vec{C}$, and bug report vector $\vec{B}$, are scored using *TF-IDF*.

*TF* is computed for each vector independently as follows [25]:

$$TF(termT, S) = log\ f_{t,s} + 1 \quad (1)$$

$$TF(termT, C) = log f_{t,c} + 1 \quad (2)$$

$$TF(termT, B) = log f_{t,b} + 1 \quad (3)$$

Similarly, *IDF* is computed as follows [24]:

$$IDF = log \frac{N}{n_i} \quad (4)$$

The weight is computed in the form *TF-IDF* as follows [25]:

$$TF-IDF = \begin{cases} (1 + \log f_{i,j}) \times \log \frac{N}{n_i} & f_{i,j} > 0 \\ 0 & otherwiswe \end{cases} \quad (5)$$

Cosine similarity exploited to derive the associated similarity between vectors according to the cosine similarity equation [3]:

$$(B,S)^n = \sum_{i=0}^{n} \frac{B}{\sqrt{\sum_{i=1}^{n} B^2}} \times \frac{S}{\sqrt{\sum_{i=1}^{n} S^2}} \tag{6}$$

$$(B,C)^n = \sum_{i=0}^{n} \frac{B}{\sqrt{\sum_{i=1}^{n} B^2}} \times \frac{C}{\sqrt{\sum_{i=1}^{n} C^2}} \tag{7}$$

According to similarity scores, source code documents are ranked against the bug report.

In VSM, methods correspond by rows and the key terms correspond by columns; in other words, each method is represented as a vector of key terms. The intersection between column and row represents the weight based on term frequency; that is, the number of times that the term appears in the corresponding documents (method or query) [38].

#### 4.3.5. Data Analysis and Interpretation

To evaluate the accuracy of our methodology, this research relies on precision, recall, and accuracy in term of f1-score. Precision represents the proportion of relevant documents among retrieved documents, whereas recall represents the proportion of relevant documents retrieved among the overall relevant documents.

Precision expressed as stated in [4]:

$$Precision = \frac{|relevent\ doc| \cap |retreived\ doc|}{|retreived\ doc|} \tag{8}$$

The recall expressed as stated in [4]:

$$Recall = \frac{|relevent\ doc| \cap |retreived\ doc|}{|relevent\ doc|} \tag{9}$$

The methodology accuracy is determined in term of F1-measure, which combines the precision and recall of the model and is commonly used for evaluating IR systems and natural language models. The f-measure expressed in term of precision and recall:

$$F1 - measure = 2 \times \frac{precision \times recall}{precision \times recall} \tag{10}$$

### *4.4. Experiments and Results*

In software engineering, software comprehension is a crucial process for understanding how software artifacts (methods, classes, fields) relate to each other. Software systems require continuous improvement and maintenance owing to emerging bugs reported by different users; locating the source fragments that cause such bugs can help the developers in locating buggy source files more effectively.

This research concentrates on augmenting source code and bug reports with more semantics to achieve a better understanding and raise the performance of bug localization. To achieve these goals, we design our dedicated Java tool named "SynPos Buglocater" that implements our illustrated approach. Different experiments were conducted to evaluate our methodology and answer our research questions. The effectiveness of the conducted experiments is measured based on precision and recall. Precision is the percentage of relevant results that are retrieved. The recall represents the percentage of all relevant results that were correctly retrieved.

RQ1: How well does the segmentation of the source code body and comments affect the bug localization process?

To answer our first research question, we apply segmentation on source files; to evaluate the effect of segmentation on the bug localization process, we conduct two experiments; the first experiment was conducted with segmentation and the second experiment without segmentation.

Without Segmentation: in this experiment, we run our tool to retrieve the buggy source file considering the whole software as is; this experiment was applied on three software programs: SWT, AspectJ, and Eclipse. The recall values for this experiments are 55.80%, 53,77%, and 43.29%, respectively. The precision values for these datasets are 55.36%, 55.33%, and 55.77%, respectively.

With Segmentation: this represents one of the main issues that we examine in our paper. In this experiment, we run our tool to retrieve the buggy source file by segmenting source code into segments: code and comments. This experiment was applied on three software programs: SWT, AspectJ, and Eclipse. The recall values for these experiments are 74.65%, 79.05%, and 74.25%, respectively. As we notice, segmenting the source code enhances the recall levels for SWT, AspectJ, and Eclipse by 18.85%, 25.28%, and 30.96%, respectively.

Furthermore, the precision values in Table 2 for these datasets are 75.62%, 72.25%, and 76.05%, respectively. As we notice, segmenting the source code enhances the precision levels for SWT, AspectJ, and Eclipse by 20.26%, 16.92%, and 20.28%, respectively.

**Table 2.** Recall and precision for segmentation.

| With Segmentation | | |
|---|---|---|
| **Software** | **Recall** | **Precision** |
| ***SWT*** | 74.65% | 75.62% |
| ***AspectJ*** | 79.05% | 72.25% |
| ***Eclipse*** | 74.25% | 76.05% |
| **Without Segmentation** | | |
| **Software** | **Recall** | **Precision** |
| ***SWT*** | 55.80% | 55.36% |
| ***AspectJ*** | 53.77% | 55.33% |
| ***Eclipse*** | 43.29% | 55.77% |

RQ2: Does our developed POS tagger improve the bug localization process?

To answer the second research question, we exploit the idea of tagging different words with a suitable tag and applying a set of handwritten contextual rules.

To evaluate the effect of POS on the bug localization process, we conduct several combination cases between bug reports and comments. Mainly, we illustrate the following experiments that have a vital effect on the results; the first experiment was conducted without applying POS and the second experiment with applying POS.

Without POS: in this experiment, we run our tool to retrieve the buggy source file without applying POS; this experiment was applied on three software programs: SWT, AspectJ, and Eclipse. The recall values for this experiment are 48.80%, 43.68%, and 55.96%, respectively. The precision values for these datasets are 63.20%, 49.96%, and 36.60%, respectively.

With POS: this represents one of the main issues that we examine in our paper. In this experiment, we run our tool to retrieve the buggy source file affected by applying POS; here, we conduct several combinations and we find that POS gains the best result when applied on both the comment and bug report, which support our presented methodology. This experiment was applied on three software programs: SWT, AspectJ, and Eclipse. The recall values for this experiment are 65.92%, 66.14%, and 63.11% respectively. As we notice, applying POS enhances the recall levels for SWT, AspectJ, and Eclipse by 17.12%, 22.46%, and 7.15%, respectively.

Furthermore, the precision values in Table 3 for these datasets are 68.89%, 63.11%, and 69.13%, respectively. As we notice, applying POS enhances the precision levels for SWT, AspectJ, and Eclipse by 5.69%, 13.15%, and 5.53%, respectively.

**Table 3.** Recall and precision for POS.

| With POS | | |
|---|---|---|
| **Software** | **Recall** | **Precision** |
| ***SWT*** | 65.92% | 68.89% |
| ***AspectJ*** | 66.14% | 63.11% |
| ***Eclipse*** | 63.11% | 69.13% |
| **Without POS** | | |
| **Software** | **Recall** | **Precision** |
| ***SWT*** | 48.80% | 63.20% |
| ***AspectJ*** | 43.68% | 49.96% |
| ***Eclipse*** | 55.96% | 63.60% |

RQ3: To what extent could the synonyms of bug reports and comments improve the bug localization process?

We conduct different type of experiments using a different combination of cases; for example, bug report with synonyms and without synonyms and, on the other hand, comments with synonyms and without synonyms. We notice that using synonyms with bug and comments enhances the performance of locating the buggy source code files.

The main issue in using synonyms is the way to select only the possible synonyms and ignore the irrelevant synonyms. This is the motivation behind the use of POS and synonyms concurrently. To evaluate the effect of synonyms, we apply a method that enhances the synonyms' selection task based on their POS. To evaluate that method, we conduct several experiments that involved several combination cases between bug reports and comments. Mainly, we illustrate the following experiments that have a vital effect on the results; the first case concentrates on generating synonyms without POS and the second case with applying POS.

With Synonyms: we conduct several combinations, include the following: bug report only, comments only, and both bug reports and comments. The best result was for applying synonyms on the combination of bug reports and comments. To evaluate the use of synonyms effectively and examine the benefit of POS, we conduct different experiments.

Synonyms Without POS: in this experiment, we run our tool to retrieve the buggy source file by generating synonyms without applying POS; this experiment was applied on three software programs: SWT, AspectJ, and Eclipse. The recall values for this experiment are 59.11%, 54.35%, and 43.87%, respectively. The precision values for these datasets are 66.33%, 55.20%, and 52.01%, respectively.

Synonyms with POS: this represents one of the main issues that we examine in our paper. In this experiment, we run our tool to retrieve the buggy source file affected by generating synonyms concurrently with applying POS. This experiment was applied on three software programs: SWT, AspectJ, and Eclipse. The recall values for this experiment are 74.65%, 79.05%, and 74.25%, respectively. As we notice, generating synonyms and using POS to reduce the synonyms sets enhance the recall levels for SWT, AspectJ, and Eclipse by 15.54%, 24.70%, and 30.38%, respectively.

Furthermore, the precision values in Table 4 for these datasets are 75.62%, 72.25%, and 76.05%, respectively. As we notice, applying POS enhances the precision levels for SWT, AspectJ, and Eclipse by 9.29%, 17.05%, and 24.04%, respectively.

RQ4: Does our developed approach improve the accuracy of bug localization?

To answer the last question, we not only investigated the effectiveness of our methodology, but we also compared our methodology against some IR-bug localization approaches. Table 5 represents the accuracy of our methodology against other approaches in terms of f1-measure. As we can see, a considerable improvement over the listed state-of-the-art was satisfied. Our methodology gains 78% on average, which exceeds those of [3,10,39] by 10%, 9%, and 4%, respectively.

**Table 4.** Recall and precision for synonyms using POS.

| **With Synonyms and POS** | | |
|---|---|---|
| **Software** | **Recall** | **Precision** |
| ***SWT*** | 74.65% | 75.62% |
| ***AspectJ*** | 79.05% | 72.25% |
| ***Eclipse*** | 74.25% | 76.05% |
| **With Synonyms Only** | | |
| **Software** | **Recall** | **Precision** |
| ***SWT*** | 59.11% | 66.33% |
| ***AspectJ*** | 54.35% | 55.20% |
| ***Eclipse*** | 43.87% | 52.01% |

**Table 5.** Accuracy of different methodologies.

| **Dataset** | **BugLocater (Zhou et al. 2012 [3])** | **BLUiR (Saha et al. 2013 [10])** | **AmaLgam (Wang and Lo 2014 [39])** | **Our Approach** |
|---|---|---|---|---|
| SWT | 69.44% | 76.03% | 84.03% | 85.22% |
| AspectJ | 71.67% | 67.07% | 70.85% | 71.65% |
| Eclipse | 62.97% | 65.13% | 67.83% | 75.67% |
| **Average** | **68.00%** | **69.00%** | **74.00%** | **78.00%** |

### *4.5. Evaluation and Discussion*

In this research, we conduct an empirical study to examine the effectiveness of our methodology. We conduct several experiments using three large-size datasets: SWT 3.1, AspectJ 1.0, and Eclipse 3.1. After segmenting source code, we apply the main preprocessing techniques, including stop words removal, tokenization, stemming, a generated corpus tagged with POS tagging, and augmented with synonyms. Segmenting source code increases the recall level by more than 25% on average. From different applied experiments, we notice that synonyms, in some cases, deteriorate the performance if we do not integrate with POS; in this case, a considerable improvement in recall is found compared with applying only synonyms. Applying synonyms with POS leads to a 23.54% improvement. Our methodology exhibits the best results when segmenting source code as we deal with large-size software, rather than considering only the source code body for bug localization, involving comments beside bug reports leads to a desirable enhancement in bug localization, as comments' textual nature is close to the nature of bug reports.

According to [34], there is a conflict result, for several reasons; they have examined poor quality commented software and they have also conducted their experiments on a limited number of features (22 feature). To avoid this conflict, we apply their method (using synonyms with manual inspection of appropriate synonym from synonym sets) on our datasets. According to their results, they satisfy an average recall level of 75.5; when applying their method on our datasets, we gain an average recall of 52.44%, while our method (applying synonyms with POS) satisfies 75.98%, which means a 23.54% improvement in recall over their method. According to these comparisons, we approve that our methodology overcomes the state-of-the-art approaches.

## 5. Conclusions and Future Works

This research presented a new approach for facilitating and improving the process of bug location using an IR technique combined with natural language techniques. The approach is undertaken using the VSM, which is a powerful IR model, along with augmenting the source code and bug reports with synonyms of terms. Our methodology was initiated by collecting source code artifacts; then, different artifacts were processed using a set of preprocessing techniques to generate a repository of processed terms (corpus), and the terms 'synonyms are then added to the corpus; on the other hand, bug reports are also processed and some rules are applied, then different queries are run to retrieve the list of suggested buggy files.

The accuracy of bug localization mainly depends on the quality of the extracted textual data. In this research, we explore three Java open source software programs. The main limitation of this research is the applicability of our developed methodology to the different programming language software. Moreover, our methodology depends on source code comments, so the poor quality or absence of comments will be problematic and will confuse our methods.

Three open source systems, AspectJ 1.0, Eclipse 3.1, and SWT 3.1, were selected to evaluate the presented methodology; all of these software systems are written using Java language and all of them are well documented. To examine the effectiveness of the presented methodology, we conduct different experiments. According to the obtained results, our presented approach satisfies a significant improvement over the other approaches. Our methodology gains 78% accuracy on average, which exceeds the state-of-the-art approaches.

As future work, we plan to run our approach on a system written in other programming languages like C++ and Python. In addition, we also plan to validate the effectiveness of our presented approach in combination with the existing approaches. Moreover, we plan to investigate other preprocessing steps to further enhance the process of locating buggy files.

We plan to improve our rules to apply them to more bug reports. Moreover, we plan to apply our presented approach based on other different retrieval models such as LSI and LDA.

**Author Contributions:** Conceptualization, A.A. and I.M.A.; Methodology, A.A.; Validation, K.S.; Investigation, I.M.A.; Resources, K.S.; Data curation, K.S.; Writing—original draft, K.S.; Writing—review & editing, A.A.; Visualization, K.S.; Supervision, I.M.A.; Project administration, I.M.A. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** All data were presented in main text.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Hanandeh, F.; Saifan, A.A.; Akour, M.; Al-Hussein, N.K.; Shatnawi, K.Z. Evaluating Maintainability of Open Source Software: A Case Study. *Int. J. Open Source Softw. Process. (IJOSSP)* **2017**, *8*, 1–20. [CrossRef]
2. Tantithamthavorn, C.; Abebe, S.L.; Hassan, A.E.; Ihara, A.; Matsumoto, K. The Impact of IR-based Classifier Configuration on the Performance and the Effort of Method-Level Bug Localization. *Inf. Softw. Technol.* **2018**, *102*, 160–174. [CrossRef]
3. Zhou, J.; Zhang, H.; Lo, D. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 14–24.
4. Khatiwada, S.; Tushev, M.; Mahmoud, A. Just enough semantics: An information theoretic approach for ir-based software bug localization. *Inf. Softw. Technol.* **2018**, *93*, 45–57. [CrossRef]
5. Aljawarneh, S.A.; Alawneh, A.; Jaradat, R. Cloud security engineering: Early stages of SDLC. *Future Gener. Comput. Syst.* **2017**, *74*, 385–392. [CrossRef]
6. Dilshener, T.; Wermelinger, M.; Yu, Y. Locating bugs without looking back. *Autom. Softw. Eng.* **2018**, *25*, 383–434. [CrossRef]
7. Huang, Y.; Huang, S.; Chen, H.; Chen, X.; Zheng, Z.; Luo, X.; Jia, N.; Hu, X.; Zhou, X. Towards automatically generating block comments for code snippets. *Inf. Softw. Technol.* **2020**, *127*, 106373. [CrossRef]
8. Newman, C.D.; AlSuhaibani, R.S.; Decker, M.J.; Peruma, A.; Kaushik, D.; Mkaouer, M.W.; Hill, E. On the generation, structure, and semantics of grammar patterns in source code identifiers. *J. Syst. Softw.* **2020**, *170*, 110740. [CrossRef]
9. Moreno, L.; Treadway, J.J.; Marcus, A.; Shen, W. On the use of stack traces to improve text retrieval-based bug localization. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 29 September–3 October 2014; pp. 151–160.

10. Saha, R.K.; Lease, M.; Khurshid, S.; Perry, D.E. Improving bug localization using structured information retrieval. In Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Silicon Valley, CA, USA, 11–15 November 2013; pp. 345–355.
11. Davies, S.; Roper, M.; Wood, M. Using bug report similarity to enhance bug localisation. In Proceedings of the 2012 19th Working Conference on Reverse Engineering, Kingston, ON, Canada, 15–18 October 2012; pp. 125–134.
12. Wang, S.; Liu, T.; Tan, L. Automatically learning semantic features for defect prediction. In Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), Austin, TX, USA, 14–22 May 2016; pp. 297–308.
13. Rahman, S.; Ganguly, K.K.; Sakib, K. An improved bug localization using structured information retrieval and version history. In Proceedings of the 2015 18th International Conference on Computer and Information Technology (ICCIT), Dhaka, Bangladesh, 21–23 December 2015; pp. 190–195.
14. Chakraborty, S.; Li, Y.; Irvine, M.; Saha, R.; Ray, B. Entropy Guided Spectrum Based Bug Localization Using Statistical Language Model. *arXiv* **2018**, arXiv:1802.06947.
15. Sisman, B.; Kak, A.C. Incorporating version histories in information retrieval based bug localization. In Proceedings of the 2012 9th IEEE Working Conference on Mining Software Repositories (MSR), Zurich, Switzerland, 2–3 June 2012; pp. 50–59.
16. Beard, M. Extending bug localization using information retrieval and code clone location techniques. In Proceedings of the 2011 18th Working Conference on Reverse Engineering, Limerick, Ireland, 17–20 October 2011; pp. 425–428.
17. Gharibi, R.; Rasekh, A.H.; Sadreddini, M.H. Locating relevant source files for bug reports using textual analysis. In Proceedings of the 2017 International Symposium on Computer Science and Software Engineering Conference (CSSE), Shiraz, Iran, 25–27 October 2017; pp. 67–72.
18. Wong, C.P.; Xiong, Y.; Zhang, H.; Hao, D.; Zhang, L.; Mei, H. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 29 September–3 October 2014; pp. 181–190.
19. Youm, K.C.; Ahn, J.; Lee, E. Improved bug localization based on code change histories and bug reports. *Inf. Softw. Technol.* **2017**, *82*, 177–192. [CrossRef]
20. Davies, S.; Roper, M. Bug localisation through diverse sources of information. In Proceedings of the 2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Pasadena, CA, USA, 4–7 November 2013; pp. 126–131.
21. Alduailij, M.; Al-Duailej, M. Performance evaluation of information retrieval models in bug localization on the method level. In Proceedings of the 2015 International Conference on Collaboration Technologies and Systems (CTS), Atlanta, GA, USA, 1–5 June 2015; pp. 305–313.
22. Lukins, S.K.; Kraft, N.A.; Etzkorn, L.H. Source code retrieval for bug localization using latent dirichlet allocation. In Proceedings of the 2008 15th Working Conference on Reverse Engineering, Antwerp, Belgium, 15–18 October 2008; pp. 155–164.
23. Uneno, Y.; Mizuno, O.; Choi, E. Using a Distributed Representation of Words in Localizing Relevant Files for Bug Reports. In Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS), Vienna, Austria, 1–3 August 2016; pp. 183–190. [CrossRef]
24. Lam, A.N.; Nguyen, A.T.; Nguyen, H.A.; Nguyen, T.N. Bug localization with combination of deep learning and information retrieval. In Proceedings of the 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), Buenos Aires, Argentina, 22–23 May 2017; pp. 218–229.
25. Xiao, Y.; Keung, J.; Bennin, K.E.; Mi, Q. Machine translation-based bug localization technique for bridging lexical gap. *Inf. Softw. Technol.* **2018**, *99*, 58–61. [CrossRef]
26. Xiao, Y.; Keung, J.; Bennin, K.E.; Mi, Q. Improving bug localization with word embedding and enhanced convolutional neural networks. *Inf. Softw. Technol.* **2019**, *105*, 17–29. [CrossRef]
27. Lam, A.N.; Nguyen, A.T.; Nguyen, H.A.; Nguyen, T.N. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; pp. 476–481.
28. Dao, T.; Zhang, L.; Meng, N. How does execution information help with information-retrieval based bug localization? In Proceedings of the 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), Buenos Aires, Argentina, 22–23 May 2017; pp. 241–250.
29. Malhotra, R.; Aggarwal, S.; Girdhar, R.; Chugh, R. Bug localization in software using NSGA-II. In Proceedings of the 2018 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE), Penang, Malaysia, 28–29 April 2018; pp. 428–433.
30. Zou, J.; Xu, L.; Yang, M.; Zhang, X.; Zeng, J.; Hirokawa, S. Automated duplicate bug report detection using multi-factor analysis. *Ieice Trans. Inf. Syst.* **2016**, *99*, 1762–1775. [CrossRef]
31. Gupta, A.; Suri, B.; Kumar, V.; Misra, S.; Blažauskas, T.; Damaševičius, R. Software Code Smell Prediction Model Using Shannon, Rényi and Tsallis Entropies. *Entropy* **2018**, *20*, 372. [CrossRef] [PubMed]
32. Kumari, M.; Misra, A.; Misra, S.; Fernandez Sanz, L.; Damasevicius, R.; Singh, V.B. Quantitative Quality Evaluation of Software Products by Considering Summary and Comments Entropy of a Reported Bug. *Entropy* **2019**, *21*, 91. [CrossRef] [PubMed]
33. Khurma, R.A.; Alsawalqah, H.; Aljarah, I.; Elaziz, M.A.; Damaševičius, R. An Enhanced Evolutionary Software Defect Prediction Method Using Island Moth Flame Optimization. *Mathematics* **2021**, *9*, 1722. [CrossRef]
34. Saifan, A.A.; Obeidat, L. Feature Location Enhancement Based on Source Code Augmentation with Synonyms of Terms. *Softw. Pract. Exp.* **2021**, *51*, 235–259. [CrossRef]

35. Hanna, S.; Alawneh, A. An Approach of Web Service Quality Attributes Specification. *Commun. IBIMA* **2010**, *2010*, 13. Available online: http://www.ibimapublishing.com/journals/CIBIMA/cibima.html (accessed on 15 September 2022). [CrossRef]
36. Hanna, S.; Alawneh, A.A. An ontology for the quality attributes of web services. Knowledge Management and Innovation in Advancing Economies: Analyses and Solutions. In Proceedings of the 13th International Business Information Management Association Conference, Marrakech, Morocco, 9–10 November 2009; Volume 3, pp. 1348–1358.
37. Al-Shawakfa, E. A Rule-based Approach to Understand Questions in Arabic Question Answering. *Jordanian J. Comput. Inf. Technol.* **2016**, *2*, 210–231.
38. Alazzam, I. Using Information Retrieval to Improve Integration Testing. Ph.D. Thesis, North Dakota State University, Fargo, ND, USA, 2012.
39. Wang, S.; Lo, D. Version history, similar report, and structure: Putting them together for improved bug localization. In Proceedings of the 22nd International Conference on Program Comprehension, Hyderabad, India, 2–3 June 2014; pp. 53–63.