Utilizing Topic-Based Similar Commit Information and CNN-LSTM Algorithm for Bug Localization

Geunseok Yang and Byungjeong Lee

Abstract: With the use of increasingly complex software, software bugs are inevitable. Software developers rely on bug reports to identify and fix these issues. In this process, developers inspect suspected buggy source code files, relying heavily on a bug report. This process is often time-consuming and increases the cost of software maintenance. To resolve this problem, we propose a novel bug localization method using topic-based similar commit information. First, the method determines similar topics for a given bug report. Then, it extracts similar bug reports and similar commit information for these topics. To extract similar bug reports on a topic, a similarity measure is calculated for a given bug report. In the process, for a given bug report and source code, features shared by similar source codes are classified and extracted; combining these features improves the method’s performance. The extracted features are presented to the convolutional neural network’s long short-term memory algorithm for model training. Finally, when a bug report is submitted to the model, a suspected buggy source code file is detected and recommended. To evaluate the performance of our method, a baseline performance comparison was conducted using code from open-source projects. Our method exhibits good performance.

Keywords: bug localization; bug report; deep learning

1. Introduction

With increasing software complexity, software bugs have become inevitable. To address such bugs, developers rely on bug reports to find buggy code files. This process can be time-consuming, depending on the quality of the bug report; it also requires developers to manually search for suspicious code files [1]. An automated recommender of candidate buggy code files can significantly reduce the cost of software maintenance.

In the past, bug reports and source code feature extraction methods have been used to detect appropriate buggy code files. Recently, deep learning algorithms for detection of suspected buggy code files have been proposed. Pradel et al. [2] proposed the neural trace-line model. This model consists of a line level and a trace level which identify buggy code files using a recurrent neural network (RNN). Wang et al. [3] improved the model’s bug localization performance using metadata and stack trace information in the analyzed bug report. Lam et al. [4] predicted buggy code files using a deep neural network (DNN) and the recurrent vector support machine (rVSM) method. They improved bug localization performance using stack traces and similar bug reports. Kim et al. [5] hypothesized that if the contents of the bug report were insufficient, an appropriate buggy file could not be predicted. Attachment files and extended queries were proposed through bug report analysis, and the performance of bug localization was improved by combining the proposed technique. Rao et al. [6] compared the performance of techniques such as text search model, unigram model, and vector space model, in bug localization, and the unigram model and vector space model showed the best performance. Saha et al. [7] classified the text of a bug report and a source code file into different groups and calculated the similarity between the
different groups individually. The performance was further improved by using structured information retrieval. In addition, deep learning algorithms have been used in many fields to solve various problems. For example, they are also used in research fields such as smart grid and sensing [8–16]. In this study, among deep learning algorithms, the convolutional neural network long short-term memory (CNN-LSTM) algorithm solves the problem of bug localization. A CNN algorithm is mainly used in image processing, but recently, it has also been widely used in text analysis. In this study, we introduce related research by dividing image processing and text analysis using a CNN algorithm as follows:

**Image Processing** Wei et al. [13] proposed the hypotheses-CNN-pooling method for multi-label image classification using a CNN algorithm. This study received object segments and performed multi-label prediction with maximum pooling. They showed better performance than the baseline in the image data. Wang et al. [14] proposed a multi-label image classification technique using a CNN and a RNN, which improved image classification performance over baseline by learning joint labels to characterize the relationship between semantic labels and image labels.

**Text Analysis** Rhanoui et al. [15] proposed sentiment analysis of French newspapers using CNN-BiLSTM (CNN and Bidirectional LSTM). The sensitivity analysis showed an accuracy of about 90%. Wang et al. [16] conducted text-dimensional-sensibility analysis using CNN-LSTM. Part of the text was divided to extract the features of each text, and the performance of sentiment analysis was improved.

In detail, the CNN and the image class are classified by extracting the features of the image based on a CNN, and the text class classification is performed by extracting the text features. However, as the length of the text increases, the vanishing gradient problem occurs, which adversely affects the model training, therefore, we try to solve it through LSTM [17]. In this study, we extract features for text class classification based on a CNN and solve the vanishing gradient problem through LSTM to prevent issues that may occur depending on the length of the program’s source code. In news data classification and Chinese data classification studies [18], the CNN-LSTM algorithm showed better results than other deep learning algorithms.

Here, we propose a bug localization method using similar topic-based commit information. We extract similar topics as well as similar bug reports and commit information. Then, features are extracted from similar bug reports and commit information and are applied to the convolutional neural network long short-term memory (CNN-LSTM) algorithm. Finally, a bug code file is recommended. In addition, to increase the method’s efficiency, features are extracted by considering the following: (1) the given bug report and the source codes of similar bug reports, and (2) the given source code and the source codes of similar bug reports. Then, the performances of each feature alone and in combination are compared. In this study, we use a machine learning algorithm to identify various patterns and rules in the data and improve the performance of bug localization. If a rule-based inference engine or heuristic algorithm are used, various patterns and rules might not be found; since it is very difficult to solve complex problems, machine learning algorithms are used in this study. In addition, in order to consider the sequential characteristics of LSTM in the learning-based model, the CNN-LSTM model is used by using the sequence order of the source code. Specifically, the features of the source code are extracted from the CNN and input into the LSTM to finally predict the buggy source code file. In general, the traceability between the bug report and the source code is not guaranteed. However, in this study, the algorithm is trained on a benchmark dataset [4] in the presence of traceability between the bug report and the source code. If a user submits a new bug report after training, our approach predicts a buggy source code file. If the method is used with a given bug report by open-source project developers, suspicious source code files can easily be found, helping to reduce the cost of software maintenance.

The contributions of this study are as follows:

- In the bug localization process, we extracted topic-based similar commit information and topic-based similar bug reports for a given bug report. Features were extracted
by calculating the similarity between the given bug report and the information in the source code files corresponding to similar bug reports. When a new bug report was submitted, candidate buggy source code files were recommended.

- Using the topic model, we extracted similar commit information for a given bug report, thus improving the performance of bug localization.
- In the feature extraction process, features were extracted from similar source codes for a given bug report and source code, and the performance of the algorithm using each feature and a combination of features was compared. We determined that use of the combination of features improved bug localization performance.
- To evaluate the effectiveness of our method, we compared its performance on baseline codes from open-source projects. Our method exhibited good performance.

This paper is organized as follows: In Section 2, we provide some background on the bug localization problem; the proposed bug localization method is presented in Section 3; the experiments are described in Section 4; the experimental results are presented in Section 5; in Section 6, related studies are described; and the study conclusions and future steps are given in Section 7.

2. Background Knowledge

2.1. Bug Report

In this study, bug reports and source code information are used for bug localization. A bug report is typically freely available from software developers and may include scenarios and dump files for bugs. For example, an Eclipse bug report (#413685) is shown in Figure 1 [19].

![Figure 1. An example of an Eclipse bug report (#413685).](image)

This report describes a bug associated with the platform’s user interface (UI). It was submitted on 24 July 2013, and the bug was corrected on 30 May 2014. The bug report contains a summary and an explanation and it is organized as a text file. To extract features
from this bug report, we perform preprocessing [20]. In this preprocessing stage, stop words are removed and a prototype is extracted. We can also check the historical status of the bug report. This bug report was corrected by hendrik.still on 31 July 2013.

2.2. Buggy Source Code

The program source code is composed of various text keywords, and preprocessing is performed to extract features from the source code. The preprocessing step includes tokenization of words, lemmatization of words, prevention of keyword removal, and application of CamelCase [20]. When a source code is preprocessed, various keywords are extracted; an example is shown for the sample source code (PreferenceDialog.java), for the Eclipse platform UI. This preprocessing step can yield various methods, listeners, classes (e.g., “setMessage”, “jface”, “setMinimumPageSize”, and “preference”). Using the terms of the bug report in Figure 1, we find a similar source code by matching the keywords “org”, “jface”, “preference”, “eclipse”, “warning”, and “cause”. Using the commit information in the bug report, the source code is corrected; the corrected source code files are shown in Figure 2 [21].

![Figure 1. An example of an Eclipse bug report (#413685).](image1)

![Figure 2. An example of a source code file (df1c938).](image2)

In the bug report (#413685), six source code files were changed, and 40 additions and 40 deletions were made. In this study, we predict the source code file by using the topic-based commit information for a given bug report.

2.3. Deep Learning Algorithm

In this study, we use the CNN-LSTM algorithm [18] to create a learning-based model. A flow chart of the algorithm is shown in Figure 3. The algorithm consists of two parts, i.e., the CNN and the LSTM network. The extracted features are presented to the CNN, while the output of the CNN is presented to the LSTM network. In detail, the bug reports and source code files are added to the input of the CNN and features of the input are extracted. Then, the extracted features are sequentially input into the LSTM model. The output of the LSTM network is a recommended buggy file. The CNN has a common configuration as
The proposed bug localization method is schematically illustrated in Figure 4. First, the bug report and source code files were extracted from the bug repository and preprocessed. Next, we built a topic model by extracting bug reports from the bug repository. During the process of constructing the topic model, a model was created using the word occurrence frequency in the bug report. Then, similar topics were found for a given bug report. Next, similar bug reports were extracted for these similar topics, and similar commit information was extracted as well. During the process of extracting similar bug reports, the source code files corresponding to the identified similar bug reports were located, and features that were shared between the source code of the given bug report and the source codes of similar bug reports were extracted. The extracted features were trained on the CNN-LSTM model, and the buggy source code file was scored. When a new bug report arrived at our model, the relevant buggy source code file was recommended, and the developer could debug the program in the file and correct the bug.

3.1. Preprocessing

Bug reports and program source codes are expressed using words; thus, preprocessing was performed for feature extraction using natural language processing tools. However, since the program source code contains structural information, the given bug report and program source code were subjected to different preprocessing approaches. First, on the one hand, stop words and special characters were removed from the given bug report, words were tokenized, and a prototype was extracted. For example, in the Eclipse platform UI bug report in Figure 1, the summary of the original bug report was “Fix compiler warnings in org.eclipse.jface.preference caused by moving jFace to Java 1.5”. By preprocessing this summary, the words (“fix”, “compiler”, “warning”, “org”, “eclipse”, “jface”, “preference”, “cause”, “move”, “jface”, “java”, and “1.5”) were extracted. The stop words (“in”, “by”, and “to”) were removed, and the word “caused” was extracted.
as “cause”. The preprocessing technique for bug reports was applied to all of the bug reports that were used, as well as to the summaries and descriptions of those bug reports. Secondly, on the other hand, the process for the program source code included tokenization of words, lemmatization of words, prevention of keyword removal, and use of CamelCase. Keyword removal prevention refers to retention of unique keywords used in Java language, such as “do”, “for”, and “this”. CamelCase was used to separate function names and included the lemmatization extraction process. For example, for the “PointcutHandlers” function, words are classified based on the starting point of capital letters, the keywords “Pointcut” and “Handlers” are extracted, and the root of the word is extracted when necessary (e.g., “Handler” for “Handlers”). The original function name is also extracted. Using the extracted word features, similar bug reports were identified, and similar source codes were determined for the identified bug reports.

![Diagram of the approach]

Figure 4. Overview of our approach.

3.2. Topic Modeling

Topic modeling is an unsupervised learning algorithm that finds hidden topics for words in a bug report and composes topics for each word in the bug report. By adjusting the topic modeling parameters, we can set the rate at which one bug report belongs to multiple topics and the rate at which one bug report word belongs to multiple topics. In this study, we performed topic modeling using bug reports. Topics were clustered using the frequency of words in the bug report, and each created topic consisted of topic words. An example of a topic model is shown in Table 1.

For example, in Topic-1, “font” was the most frequent word, so the topic was related to “font”. Words such as ‘color’, “size”, “icon”, and “look” were also regarded as topic words that were related to “font”. Topic-3 was related to “except”. Each topic was linked to a corresponding bug report. In this study, we searched for similar topics and conducted research using bug reports within the topic.
3.3. Similarity Measure

We find similar topic-based bug reports for a given bug report using the similarity measure [22], as shown in Equation (1) as follows:

$$\text{Bug}_\text{BM25}_\text{Score}(\text{Brt}, q_i) = \sum_{i=1}^{n} \log \left(1 + \frac{n - \text{BrtFeq}_N + 0.5}{\text{BrtFeq}_N + 0.5}\right) \times \frac{\text{BrtFeq}_N}{\text{BrtFeq}_N + k \times \left(1 - b + b \times \frac{\text{Len(Brt)}}{\text{BrtAvgLen}}\right)}$$ (1)

- $\text{Brt}$ is a bug report, and $q_i$ is the $i$-th query;
- $n$ is the total number of bug reports, and $\text{BrtFeq}_N$ is the number of bug reports containing the word;
- $k$ is the parameter that adjusts the weight of the word frequency, while $b$ adjusts the weight of the length of the bug report;
- $\text{Len(Brt)}$ denotes the average length of all bug reports, and $\text{BrtAvgLen}$ is the average number of words in all bug reports.

Using this approach, the degree of similarity is estimated for all bug reports under consideration. To determine similar source codes, we use Equation (2) as follows:

$$\text{BugSimScore} (\text{BugDoc}, \text{BugSrc}) = \frac{|\text{BugDoc}_\text{Term} \cap \text{BugSrc}_\text{Term}|}{|\text{BugDoc}_\text{Term} \cup \text{BugSrc}_\text{Term}|}$$ (2)

- $\text{BugDoc}$ represents the given bug report or source code, and $\text{BugSrc}$ represents the source code;
- $\text{BugDoc}_\text{Term}$ is a set of terms in the given bug report or source code, and $\text{BugSrc}_\text{Term}$ is a set of terms in the source code;
- The numerator is the frequency of words that are matched between the given document and similar source code;
- The denominator is the normalization factor.

The similarity of similar bug reports can be assessed and given artifacts including bug reports and source codes. In this study, the degree of similarity was calculated using the BM25 algorithm. This algorithm differs from other similarity algorithms by considering the length of the document. For example, searching for a specific keyword in a bug report with a long bug description may mean something different than searching in a short bug report. Therefore, in this study, the word search was appropriate considering the length of the bug report, and the BM25 algorithm was used to find similar bug reports.
3.4. Bug Localization

Before feature extraction, we ensured traceability between bug reports, source codes, and commit information. First, we extracted similar topic-based commits for a given bug report. Then, using the given bug report and source code, features were extracted from similar source codes. When using similar commit information, features were extracted using the most recent commit information. The extracted features included the terms in the bug report and the source code. In this paper, we proposed two methods of feature extraction. The feature extraction process is shown in Figure 5.

![Figure 5. Overview of the feature extraction process. (a) refers to the step of extracting similar bug reports from Topic-K, (b) refers to the step of extracting features from Topic-K’s bug report, and (c) refers to the step of extracting features from Topic-K’s source code.](image)

In detail, as shown in Figure 5a, the process is described which is akin to the process of determining similar bug reports. A bug report is traceable to its corresponding source code file; thus, the source code files of similar bug reports can be identified. In Figure 5b, the degree of similarity of the source code file that is similar to the given bug report is calculated. In Figure 5c, the degree of similarity of the source code file that is similar to the source code for a given bug report is calculated.

According to the calculated similarity, the Top-K similar source codes were selected, and features were extracted from the selected source codes. The extracted features were presented to the CNN-LSTM algorithm for training of the method. In the process of model training, we found bug reports in a given bug report on a similar topic and used the extracted bug reports as input for the CNN. The features extracted from the CNN were sequentially used as input for the LSTM and a similar source code file was produced as the output. Finally, the candidate buggy source code files were scored.
4. Experiment

4.1. Dataset

In this study, the following datasets were used to implement bug localization. These datasets are described in Table 2.

Table 2. Dataset summary.

| Projects | # of Reports | # of Sources |
|----------|--------------|--------------|
| AspectJ  | 593          | 4439         |
| Birt     | 4178         | 6841         |
| UI       | 6495         | 3454         |
| JDT      | 6274         | 8184         |
| SWT      | 4151         | 2056         |

We conducted bug localization research using the Eclipse (Apache, Birt, UI, JDT, SWT) dataset of open-source projects [21,23–26]. The source code was written in Java language. The bug reports used to construct the topic model were also extracted from the Eclipse repository [27]. Overall, 30,000 bug reports were extracted, covering the period from 10 October 2001 to 20 February 2016; these reports were extracted by parsing the JSON files in the bug repository [27]. In this study, we used a benchmark dataset that is commonly used in bug localization studies. The dataset consists of Eclipse open-source projects, and we used the Eclipse dataset for quantitative performance comparisons with a baseline. However, it is difficult to use this dataset for topic modeling because it does not contain many bug reports. Thus, other bug reports from the same period were additionally extracted, and it seemed appropriate to obtain additional bug reports from the same projects. The additionally obtained bug reports were used only for topic modeling construction.

4.2. Evaluation Metric

Our work describes a model learning process. Therefore, K-fold cross-validation [28] was used to reduce the bias in the data, with the number of folds (K) set to 10. In this 10-fold cross-validation, the entire dataset was divided in a ratio of 9:1; nine folds were used to train the model, while the remaining fold was used to test the trained model.

Using the Top-K concept, the model’s accuracy was calculated to determine if a correct source code file existed among the Top-K recommended source code files. In this study, the value of K ranged from 1 to 20. As an example, K = 7 means that 7 source code files were recommended, and if there was a source code file corresponding to the correct answer, the recommendation was considered to be successful. To enable an accurate performance comparison, this procedure was used both for the currently developed method and for the baseline methods with which the current method was compared.

4.3. Baselines

Our method was evaluated by comparing its performance to those of some baseline methods from related studies. These baseline methods were as follows:

- Lam et al. [4] (called DNNLoc) combined information retrieval and the DNN approach for prediction of buggy source code files. DNNLoc, described in the introduction, provides bug localization based on deep neural networks with bug report and source code and is compared with our approach as a baseline for performance comparison.
- LR (Learning-to-Rank) [29] extracts features from source code files, API (Application Programming Interface) [30] specifications, and bug correction history, and predicts buggy source code files using adaptive learning.
- NB (Naïve Bayes) [31] uses the version, platform, and priority information of the analyzed bug report, and learns the Naïve Bayes algorithm to predict buggy source code files.
• BL (Bug Localization) [32] calculates the degree of similarity between the analyzed bug report and source code files and predicts buggy source code files using the corrected source codes in similar bug reports.

For this study, public source code that can be reproduced and published was selected as a baseline. If a relevant study could not be reproduced, it was excluded from the baseline selection.

4.4. Research Question

In this study, experiments were designed and conducted to address the following research questions:

Research Question 1 (RQ1) How accurate is our model?

First, it is necessary to verify the performance of our model. During the model construction process, we classify and extract those features that are shared by the given bug report and its source code and similar source codes, respectively. This allows us to check whether combining features improves the model’s performance. In addition, optimal parameter values can be determined using parameter adaptation.

(RQ2) Does the proposed method provide better accuracy than baseline in terms of bug localization?

To address this question, we compared the performance of our method to that of baseline methods. If the performance of our method was better than that of the baseline methods, the developed method was deemed useful for bug localization. In addition, through statistical verification [33,34], null and alternative hypotheses were formulated, allowing us to determine whether there was a significant difference between our method and the baseline methods. For statistical verification, Shapiro normality [35] was tested first using the results of our model and the baseline approaches. If the normal distribution of the data was less than or equal to 0.05, a t-test [33] was performed; otherwise, the Wilcoxon test [34] was performed. The decision to reject the null hypothesis was based on the results of the t-test and Wilcoxon test.

The null hypotheses established, in this study, are as follows:

\[ H_{10}, H_{20}, H_{30}, H_{40} \] There is no significant difference in AspectJ between the DNNLoc, LR, NB, and BL methods and our method.
\[ H_{50}, H_{60}, H_{70}, H_{80} \] There is no significant difference in Birt between the DNNLoc, LR, NB, and BL methods and our method.
\[ H_{90}, H_{100}, H_{110}, H_{120} \] There is no significant difference in the Eclipse platform UI between the DNNLoc, LR, NB, and BL methods and our method.
\[ H_{130}, H_{140}, H_{150}, H_{160} \] There is no significant difference in JDT between the DNNLoc, LR, NB, and BL methods and our method.
\[ H_{170}, H_{180}, H_{190}, H_{200} \] There is no significant difference in SWT between the DNNLoc, LR, NB, and BL methods and our method.

4.5. Results

4.5.1. RQ1: How Accurate Is Our Model?

Our model’s performance is summarized in Figure 6. In Figure 6, the X-axis corresponds to the Top-K variable, while the Y-axis shows the corresponding average F-measure. For example, \( K = 10 \) means that there is a correct answer among the top 10 recommended buggy source code files. Clearly, performance increases as the number of recommended source codes increases. Overall, Top-15 shows good performance, with an average of 81.7% over all projects.

We also investigated which features of our similar bug report affect bug localization. Three cases were considered, as follows:

Case 1 Feature extraction from a given bug report for a similar source code (Figure 5b);
Case 2 Feature extraction from a given source code for a similar source code (Figure 5c);
Case 3 Feature extraction from a given bug report and a given source code for a similar source code (combination of Figure 5b,c).
The results of this feature extraction study are shown in Figure 7.

Figure 6. Results of our approach.

In Figure 6, the X-axis corresponds to the Top-K variable, while the Y-axis shows the corresponding average F-measure. For example, K = 10 means that there is a correct answer among the top 10 recommended buggy source code files. Clearly, performance increases as the number of recommended source codes increases. Overall, Top-15 shows good performance, with an average of 81.7% over all projects.

We also investigated which features of our similar bug report affect bug localization. Three cases were considered, as follows:

Case 1: Feature extraction from a given bug report for a similar source code (Figure 5b);

Case 2: Feature extraction from a given source code for a similar source code (Figure 5c);

Case 3: Feature extraction from a given bug report and a given source code for a similar source code (combination of Figure 5b,c).

The results of this feature extraction study are shown in Figure 7.

Figure 7. Feature extraction results.

In Figure 7, the X-axis refers to open-source projects, while the Y-axis refers to the average F-measure. In Case 1, features are extracted from a given bug report for the source code of a similar bug report. In Case 2, features are extracted from a given source code for the source code of a similar bug report. In Case 3, features are extracted using both a given bug report and source code, for the source code of a similar bug report. For the source code of a similar bug report, feature extraction from a given bug report and source code shows good performance.
Answer to RQ1: According to the results of our model, the average over all of the considered projects was 81.7% at Top-15, showing good performance. Use of a combination of features yielded better performance than use of individual feature sets.

4.5.2. RQ2: Does the Proposed Method Provide Better Accuracy than Baseline in Terms of Bug Localization?

To answer this question, our model was compared to some baseline methods. The results are shown in Figure 8.

![Figure 8. Comparisons with baseline.](image_url)

In Figure 8, the X-axis refers to the model type, including our method and the DNNLoc, LR, BL, and NB approaches. The Y-axis shows the F-measure for both our model and the considered baseline methods. In addition, we assessed the statistical differences between our model and the considered baseline methods. The statistical verification results are shown in Table 3.

For example, null hypothesis H9 holds that there is no significant difference between the performance of the DNNLoc approach and that of the method proposed in the Eclipse UI project. However, since the p-value is $2.097 \times 10^{-5}$ (0.00002097), this null hypothesis can be safely rejected. Therefore, our method and the DNNLoc method differ significantly on the Eclipse UI project. In addition, for null hypothesis H17, the p-value is 0.0001439 which is less than 0.05; therefore, the null hypothesis is rejected, and the alternative hypothesis is approved. Therefore, there is a significant difference in SWT between our model and the DNNLoc method. According to these results we reject all null hypotheses and approve all alternative hypotheses.

Answer to RQ2: Overall, our approach exhibits better performance than the considered baseline methods, and the differences are statistically significant.
Table 3. Bug localization statistical tests results.

| Null Hypothesis | p-Value       | Alternative Hypothesis |
|-----------------|---------------|------------------------|
| H1_0            | 0.0003198     | H1_a: Accept           |
| H2_0            | 6.599 × 10⁻⁷  | H2_a: Accept           |
| H3_0            | 1.172 × 10⁻⁸  | H3_a: Accept           |
| H4_0            | 5.784 × 10⁻⁷  | H4_a: Accept           |
| H5_0            | 1.782 × 10⁻⁵  | H5_a: Accept           |
| H6_0            | 5.169 × 10⁻⁹  | H6_a: Accept           |
| H7_0            | 9.03 × 10⁻⁸   | H7_a: Accept           |
| H8_0            | 4.395 × 10⁻⁶  | H8_a: Accept           |
| H9_0            | 2.097 × 10⁻⁵  | H9_a: Accept           |
| H10_0           | 4.907 × 10⁻⁷  | H10_a: Accept          |
| H11_0           | 1.872 × 10⁻⁷  | H11_a: Accept          |
| H12_0           | 1.351 × 10⁻⁶  | H12_a: Accept          |
| H13_0           | 7.454 × 10⁻⁷  | H13_a: Accept          |
| H14_0           | 8.26 × 10⁻⁸   | H14_a: Accept          |
| H15_0           | 1.153 × 10⁻⁸  | H15_a: Accept          |
| H16_0           | 3.563 × 10⁻⁶  | H16_a: Accept          |
| H17_0           | 0.0001439     | H17_a: Accept          |
| H18_0           | 1.819 × 10⁻⁶  | H18_a: Accept          |
| H19_0           | 4.529 × 10⁻⁶  | H19_a: Accept          |
| H20_0           | 8.838 × 10⁻⁶  | H20_a: Accept          |

5. Discussion

In our approach, features were extracted using topic-based commit information. Similarity between bug reports and source codes was also assessed. The extracted features were presented to the CNN-LSTM algorithm, which allowed for suggestion of candidate buggy source code files. Our method exhibited better performance than the considered baseline methods, in terms of the average F-measure over 10-fold cross-validation. We note that a typical program source code is composed of structural information and sequential characteristics of a programming language. For example, in the case of the C language main function, the structural information of “int void main” is included, and the association information that “void” can follow after the word “int” can be analyzed. Therefore, by analyzing the relationship between “int” and “void” and between “int” and “main” as sequential information, text analysis of the program source code can be performed. In order to consider these characteristics, we constructed a model based on LSTM and utilized sequential information of the source code. In detail, features of the source code are extracted based on a CNN, and features are sequentially put into LSTM inputs. Then, the source code is presented as input to the CNN and the output of the CNN is presented to the LSTM to finally predict the suspicious buggy source code file. We note that the CNN-LSTM algorithm using sequential information of source code showed better results than the DNN-based deep learning algorithm introduced in this study. In addition, the performance of our method was significantly different from the performances of the considered baseline methods. Performance comparisons were made by adjusting the number of similar source codes, as shown in Figure 9.
In Figure 9, the X-axis shows the number of similar codes (K) and the Y-axis shows the average F-measure (average TOP-1 to TOP-15). A steady increase was observed until TOP-3, but performance tended to decrease beyond that point. For example, in AspectJ, when three similar source codes are used, the F-measure (average TOP-1 to TOP-15) is about 71%. In all projects, TOP-3 performed well, and TOP-5 performed the worst. The word sequence extracted from the source code has been applied to CNN-LSTM. As the number of source codes increases, the overall source code size (LOC) and the size of the word sequence increase. The application of similar source codes up to TOP-3 produces good results. However, some noise was created during model training and the F-measure decreased when the number of source codes became larger than three. In the future, we plan to investigate the relationship between code size and model learning, including in other open-source projects. The LSTM algorithm is known to solve the problem of hidden layer weights not updating normally and eventually disappearing. In future studies, we plan to investigate this issue and improve the method’s performance by adjusting the size of the word dictionary.

As a construct threat of our study, the method’s performance accuracy was calculated using K-fold cross-validation and the Top-K concept. However, the evaluation scale that was employed may not ensure the generalizability of our approach. In future studies, we plan to test various evaluation scales to further validate the approach. The parameters used in our approach are generally well known. However, these parameters do not always yield good performance. We plan to optimize the values of the parameters by training the model on more data from various projects. In this study, our approach was validated on Eclipse open-source projects based on Java programming language. This confinement to open-source projects may negatively affect the method’s generalizability. We plan to apply other open-source projects based on various programming languages in the future. In addition, in this work, our method was validated on code that was written in Java language; in future studies, we will seek to extend the applicability of the method to other programming languages. Even if there is an absence of similar bug reports, the proposed bug localization process can proceed using only the content of the new bug report, but additional verification of model performance will be required.
6. Related Works

In this paper, we explain related research by dividing the information retrieval-based methods into bug localization, machine learning, and semantic methods.

6.1. Information Retrieval-Based Methods

Information retrieval-based methods preprocess a bug report and its corresponding source code, and extract features from classified tokens. Zhou et al. [32] identified similar source codes and similar bug reports for a given bug report using the VSM approach and extracted related features. They performed using an open-source project, and the accuracy was about 62% (TOP-10) in Eclipse. Our study differs in that it compares the degree of similarity between a given bug report and source code using the source code linked to the identified similar bug report. Wang et al. [3] improved the bug localization performance by using the version history of the source code file, code structure information such as function and variable names, and a similar bug report extraction technique. They combined the observations proposed in their study to improve bug localization performance by up to about 67%. Kim et al. [5] hypothesized that a lack of sufficient information in the analyzed bug report may preclude the model from identifying the location of the corresponding buggy source code. In details, they extended the bug report with attachments and extended queries. The proposed method in the experiment improved the performance by about 17% (TOP-1) as compared with the baseline. Poshvanyk et al. [36] applied the latent semantic indexing (LSI) technique to bug localization using cosine similarity in source code files and bug reports. Lukins et al. [37] extracted a source code file that was similar to a given bug report by extracting topics from the bug report and source code. In the topic extraction process, the latent Dirichlet allocation technique was applied, and the modularity and scalability were shown to be good. Rao et al. [6] used a complex text search model, the unigram model (UM), vector space model (VSM), latent semantic analysis (LSA), and latent Dirichlet allocation for bug localization. According to the results of that study, the UM and VSM showed good performance in terms of bug localization. Saha et al. [7] proposed BLuIR using structured information retrieval to further improve the performance of existing solutions. BLuIR classifies the source code file and the text of the bug report into different groups and calculates the similarities between different groups separately before combining the similarity scores. In detail, the text of the source code file is divided into class name, method name, identifier name, and comment, and the text of the summary and description of the bug report is used. Moreno et al. [38] analyzed control flow and data flow dependence through bug reports and stack traces. They experimented with 14 open-source projects, with an efficiency of around 82% in Lucene. Rahman et al. [39] extracted words from source code files and performed bug localization based on information retrieval.

6.2. Machine Learning-Based Methods

Kim et al. [31] trained a Naïve Bayes model by extracting feature-related information from the bug report. The extracted information included version, platform, and priority-related information. Lam et al. [4] proposed a bug localization approach that combines information retrieval and machine learning techniques. By extracting features from bug reports and source codes, they identified buggy source code files using the rVSM and DNN algorithms. Huo et al. [40] proposed the NP-CNN algorithm for extracting structural information from source code. This algorithm extracts integrated features from the analyzed bug report and its corresponding source code. Performance is improved by utilizing sequential features in the source code.

6.3. Semantic Bug Localization

The words that are extracted from the analyzed bug report and its corresponding source code may be ambiguous, and it may not be possible to extract them satisfactorily using search-based techniques. This approach extracts the bug report as one vector and the source code as another vector; thus, semantic information can be ignored in the process.
of extracting the same vector. Ye et al. [29] addressed the difference between the words extracted from a bug report and those extracted from its corresponding source code using the API specification and bug correction history. Mou et al. [41] identified structural information by learning the vector representation of the analyzed source code using a tree-based CNN algorithm.

6.4. Qualitative Comparison

In most studies, information retrieval or model learning was conducted using bug reports and stack traces. A qualitative comparison between the different baseline approaches is given in Table 4.

Most studies used VSM, Metadata, Attachment, and LDA (Latent Dirichlet Allocation) for information retrieval. In addition, IR (Information Retrieval) and ML (Machine Learning) were combined to improve the performance of buggy code file prediction (e.g., the studies by Lam [4] and Huo [40]). However, the performance of the models in those studies needed to be improved upon. In this study, we predicted buggy files by combining LDA and Metadata using CNN-LSTM. Our work showed better performance than Lam’s method [4]. Our approach differs from previous approaches in that information is extracted using topic-based commits and metadata and learning is performed using a CNN-LSTM network.

Table 4. Qualitative comparison of different approaches.

| Author   | IR       | ML         |
|----------|----------|------------|
| Zhou [32]| VSM      | X          |
| Wang [3] | Metadata | X          |
| Kim [5]  | Query, Attachment | X          |
| Lukins [37]| LDA    | X          |
| Kim [31] | Metadata | Naive Bayes|
| Lam [4]  | IR       | DNN        |
| Huo [40] | IR       | NP-CNN     |
| Ye [29]  | Metadata | X          |
| Mou [41] | X        | Tree-based CNN|
| Our Approach | LDA, Metadata | CNN-LSTM |

7. Conclusions

In this study, we proposed a bug localization technique using topic-based similar commit information for a given bug report. First, we found similar topics using a given bug report. Then, we extracted similar commits corresponding to similar topics. Next, we extracted similar bug reports for these topics. Then, by extracting the source codes related to these similar bug reports, the degree of similarity between the analyzed bug report and the source code was calculated. The model was trained by extracting features from similar source codes and presenting them to a CNN-LSTM network. In addition, a given bug report and common features were extracted from similar source codes and use of a combination of features improved the bug localization performance. To evaluate the performance of our method, an open-source project was used to compare it to the performances of different baseline approaches. Our method exhibited superior performance than the considered baseline methods.

Our approach was applied to the Eclipse open-source project based on Java language to verify the model, but we plan to extend the model further so that it can be applied to various open-source projects and various programming languages in the future.
Author Contributions: Conceptualization, G.Y. and B.L.; methodology, G.Y. and B.L.; writing—original draft preparation, G.Y. and B.L.; writing—review and editing, G.Y. and B.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT). (No. 2020R1A2B5B01002467).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The benchmark data used is added into the reference [21, 23–26].

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

References
1. Seacord, R.C.; Plakosh, D.; Lewis, G.A. Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices; Addison-Wesley Professional. 2003. Available online: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=30668 (accessed on 27 February 2021).
2. Pradel, M.; Murali, V.; Qian, R.; Machalica, M.; Meijer, E.; Chandra, S. Scaffle: Bug localization on millions of files. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Los Angeles, CA, USA, 18–22 July 2020; pp. 225–236.
3. Wang, Y.; Yao, Y.; Tong, H.; Huo, X.; Li, M.; Xu, F.; Lu, J. Enhancing supervised bug localization with metadata and stack-trace.Knowl. Inf. Syst. 2020, 62, 2461–2484. [CrossRef]
4. 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 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), Buenos Aires, Argentina, 22–23 May 2017; pp. 218–229.
5. Kim, M.; Lee, A. A Novel Approach to Automatic Query Reformulation for IR-based Bug Localization. In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, Limassol, Cyprus, 8–12 April 2019; pp. 1752–1759.
6. Rao, S.; Kak, A. Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models. In Proceedings of the 8th Working Conference on Mining Software Repositories, Honolulu, HI, USA, 21–22 May 2011; pp. 43–52.
7. Saha, R.K.; Lease, M.; Khurshid, S.; Perry, D.E. Improving Bug Localization using Structured Information Retrieval. In Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, Silicon Valley, CA, USA, 11–15 November 2013; pp. 345–355.
8. Li, X.; Xu, S.; Hua, X. Pattern Recognition of Grating Perimeter Intrusion Behavior in Deep Learning Method. Symmetry 2021, 13, 87. [CrossRef]
9. Kalajdjeski, J.; Zdravevski, E.; Corizzo, R.; Lameski, P.; Kalajdziski, S.; Pires, I.M.; Garcia, N.M.; Trajkovik, V. Air Pollution Prediction with Multi-Modal Data and Deep Neural Networks. Remote Sens. 2020, 12, 4142. [CrossRef]
10. Xue, W.; Dai, X.; Liu, L. Remote Sensing Scene Classification based on Multi-structure Deep Features Fusion. IEEE Access 2020, 8, 28746–28755. [CrossRef]
11. Ceci, M.; Corizzo, R.; Japkowicz, N.; Mignone, P.; Pio, G. ECHAD: Embedding-based Change Detection from Multivariate Time Series in Smart Grids. IEEE Access 2020, 8, 156053–156066. [CrossRef]
12. Mulyanto, M.; Faisal, M.; Prakosa, S.W.; Leu, J.S. Effectiveness of Focal Loss for Minority Classification in Network Intrusion Detection Systems. Symmetry 2021, 13, 4. [CrossRef]
13. Wei, Y.; Xia, W.; Lin, M.; Huang, J.; Ni, B.; Dong, J.; Zhao, Y.; Yan, S. HCP: A Flexible CNN Framework for Multi-Label Image Classification. IEEE Trans. Pattern Anal. Mach. Intell. 2015, 38, 1901–1907. [CrossRef]
14. Wang, J.; Yang, Y.; Mao, J.; Huang, Z.; Huang, C.; Xu, W. CNN-RNN: A Unified Framework for Multi-label Image Classification. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 2285–2294.
15. Rhanoui, M.; Mikram, M.; Youssi, S.; Barzali, S. A CNN-BiLSTM Model for Document-level Sentiment Analysis. Mach. Learn. Knowl. Extr. 2019, 1, 832–847. [CrossRef]
16. Wang, J.; Yu, L.C.; Lai, K.R.; Zhang, X. Tree-structured Regional CNN-LSTM Model for Dimensional Sentiment Analysis. IEEE/ACM Trans. Audio Speech Lang. Process. 2019, 28, 581–591. [CrossRef]
17. Sundermeyer, M.; Schlüter, R.; Ney, H. LSTM Neural Networks for Language Modeling. In Proceedings of the Thirteenth Annual Conference of the Computational Speech Communication Association, Portland, Oregon, 9–13 September 2012.
18. She, X.; Zhang, D. Text Classification based on Hybrid CNN-LSTM Hybrid Model. In Proceedings of the 2018 11th International Symposium on Computational Intelligence and Design, Hangzhou, China, 8–9 December 2018; Volume 2, pp. 185–189.
19. Eclipse Bugzilla. Available online: https://bugs.eclipse.org/bugs/show_bug.cgi?id=413685 (accessed on 27 February 2021).
20. Yang, G.; Min, K.; Lee, B. Applying deep learning algorithm to automatic bug localization and repair. In Proceedings of the 35th Annual ACM Symposium on Applied Computing, Brno, Czech Republic, 30 March–3 April 2020; pp. 1634–1641.
21. Eclipse Platform UI Github. Available online: https://github.com/eclipse/eclipse.platform.ui/commit/df1c93847b49179796573d35b960d7fa7d4f6ef (accessed on 27 February 2021).
22. Wikipedia. Available online: https://en.wikipedia.org/wiki/Okapi_BM25 (accessed on 27 February 2021).
23. Eclipse AspectJ Github. Available online: https://github.com/eclipse/org.aspectj (accessed on 27 February 2021).
24. Eclipse Birt Github. Available online: https://github.com/eclipse/birt (accessed on 27 February 2021).
25. Eclipse SWT Github. Available online: https://github.com/eclipse/eclipse.platform.swt (accessed on 27 February 2021).
26. Eclipse Bugzilla. Available online: https://bugs.eclipse.org/bugs/ (accessed on 27 February 2021).
27. Kohavi, R. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In Proceedings of the 14th International Joint Conference on Artificial Intelligence, Montreal, QC, Canada, 20–25 August 1995; Volume 14, pp. 1137–1145.
28. Ye, X.; Bunescu, R.; Liu, C. Learning to Rank Relevant Files for Bug Reports using Domain Knowledge. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–22 November 2014; pp. 689–699.
29. API. Available online: https://en.wikipedia.org/wiki/API (accessed on 27 February 2021).
30. Kim, D.; Tao, Y.; Kim, S.; Zeller, A. Where Should We Fix this Bug? A Two-phase Recommendation Model. IEEE Trans. Softw. Eng. 2013, 39, 1597–1610.
31. 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 34th International Conference on Software Engineering, Zurich, Switzerland, 2–9 June 2012; pp. 14–24.
32. The T-Test. Available online: https://conjointly.com/kb/statistical-student-t-test (accessed on 27 February 2021).
33. Wilcox, F. Individual Comparisons by Ranking Methods. In Introduction to statistical analysis; Springer: New York, NY, USA, 1992; pp. 196–202.
34. Shapiro, S.S.; Wilk, M.B. An Analysis of Variance Test for Normality (Complete Samples). Biometrika 1965, 52, 591–611. [CrossRef]
35. Poshyanvanyk, D.; Gueheneuc, Y.G.; Marcus, A.; Antoniol, G.; Rajlich, V. Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval. IEEE Trans. Softw. Eng. 2007, 33, 420–432. [CrossRef]
36. Lukins, S.K.; Kraft, N.A.; Etzkorn, L.H. Source Code Retrieval for Bug Localization using Latent Dirichlet Allocation. In Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, 15–18 October 2008; pp. 155–164.
37. Rahman, M.M.; Roy, C.K. Improving IR-based Bug Localization with Context-aware Query Reformulation. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA, 4–9 November 2018; pp. 621–632.
38. Huo, X.; Li, M.; Zhou, Z.H. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code. In Proceedings of the International Joint Conference on Artificial Intelligence, New York, NY, USA, 9–15 July 2016; pp. 1606–1612.
39. Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional Neural Networks over Tree Structures for Programming Language Processing. arXiv 2014, arXiv:1409.5718.