SBugLocater: Bug Localization Based on Deep Matching and Information Retrieval

Xuxiang Huang,1 Chen Xiang,1 Hua Li,1 and Peng He1,2,3

1School of Computer Science and Information Engineering, Hubei University, Wuhan 430062, China
2Hubei Key Laboratory of Applied Mathematics, Hubei University, Wuhan 430062, China
3Hubei Engineering Research Center for Educational Informationization, Wuhan 430062, China

Correspondence should be addressed to Peng He; penghe@hubu.edu.cn

Received 23 March 2022; Revised 27 June 2022; Accepted 30 June 2022; Published 24 August 2022

Copyright © 2022 Xuxiang Huang et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Bug localization is a technology that locates buggy source files using bug reports reported by users. Automatic localization of buggy files can speed up the process of bug fixing to improve the efficiency and productivity of software quality assurance teams. Nowadays, some research studies have investigated the natural language information retrieval technology, but few of them have applied the matching technology in deep learning to bug localization. Therefore, we propose a bug localization model SBugLocater based on deep matching and IR. The model composes of three layers: semantic matching layer, relevance matching layer, and IR layer. In particular, the relevance matching layer captures fine-grained local matching signals, while coarse-grained semantic similarity signals come from the semantic matching layer. Further, based on collaborative filtering in different directions, the IR layer works to find whether bug reports and source files are related, which indirectly transforms the matching task of different grammatical structures between bug reports and source files into the same structure and solves the mismatching problem of the first two matching models when the query is short. In our work, four benchmark data sets are used as experimental data sets and Accuracy@k, MAP, and MRR as evaluation metrics, which are used to compare and analyze the performance of bug localization with the four state-of-the-art methods. Experimental results show that SBugLocater outperforms the four models. For example, compared with the best of the four models, the evaluation metrics of Accuracy@10, MAP, and MRR are improved by 6.9%, 13.9%, and 17%, respectively.

1. Introduction

In software engineering, the traditional method of bug localization is that programmers manually locate the responsible source code file according to the program bug report fed back by users. Bug localization is an important task for locating and fixing bugs, which is widely regarded as one of the most tedious, time-consuming, and expensive activities in program debugging [1, 2] due to the increasing scale and complexity of software. Locating the responsible source code file for a newly reported bug usually involves careful analysis of the bug report and numerous source files [3]. Automated bug localization, which has been widely concerned in recent years, mitigates this problem.

In existing bug localization research studies, spectrum-based [4] and IR-based [5–7] methods are two mainstream efforts. The spectrum-based method extracts static features from source codes or execution information, which is a time-consuming process. The IR-based method regards bug reports as queries and source files as documents and then sorts out candidate files according to text similarity between bug reports and source files [8]. The IR-based method focuses its most attention on term weights of natural language texts without considering source codes’ semantics. According to a previous study, semantic information is important for code suggestion [9] and code completion [10] and is also significant to bug localization. For this reason, research on semantic-based bug
localization has become the main trend at present. To bridge the semantic gap, deep learning is applied to semantic analysis. The focus of semantic matching model is to build accurate semantic representation for the whole bug reports and source files. LS-CNN employs CNN and LSTM to capture semantic information in bug reports and structure information among sentences in source files. Xiao et al. [11] combined multiple NLP-based semantic information extraction technologies to help represent bug reports and source files and designed an enhanced CNN to make use of bug fixing history. Some research studies have made progress in bug localization based on semantic matching, but none of them ever evaluated the relevance matching model’s performance in bug localization, nor has it analyzed the advantages of combining the two matching models compared with a single matching model. In addition, our model also considers the problem of low matching score with source files caused by brief and rough description of some bug reports, so the information retrieval method based on collaborative filtering is introduced into the model. We analyze the advantages of the combination of the two models over the single model through theoretical analysis and ablation experiments. In addition, we also discussed the performance improvement of the matching model by combining the IR model based on collaborative filtering. We performed empirical experiments on four project data sets to show the impact of each of the three layers on bug localization performance and compared SBugLocater performance against four state-of-the-art bug localization methods (DeepLocator, HyLoc, LR + We, and BugLocator). The main contributions of this work are summarized as follows:

1. We introduce a pretrained model into bug localization, which can learn the semantic representation of bug reports and source file and effectively bridge the semantic gap between them.
2. We designed a relevance matching model based on attention mechanism to capture keyword matching signals.
3. We propose a new bug localization framework SBugLocater, which integrates relevance matching and semantic matching and cooperates with information retrieval based on collaborative filtering to locate bugs.
4. We conduct experimental comparisons with state-of-the-art models on four benchmark data sets, and the results show that SBugLocater outperforms the existing models.

The rest of this study is distributed as follows. Section 2 examines related work. Section 3 introduces some primary knowledge and technology of SBugLocater. Section 4 elaborates SBugLocater and its overall framework. Section 5 presents experiment settings and evaluation metrics. Sections 6 and 7 analyze and discuss the experimental results. Section 8 discusses threats posed to SBugLocater’s effectiveness. Section 9 summarizes the whole paper.

2. Related Work

Conventional methods of bug localization include information retrieval and spectrum-based analysis. For example, Saha et al. [1] proposed structured information retrieval BLUiR, which is based on code structure. The model can work more accurately in bug localization. It is shown that only with source files and bug reports and bug similarity data for retrieval, BLUiR is superior to current state-of-the-art tools in the applications it considers. Zhou et al. [12] proposed BugLocator, which was based on a revised vector space model (rVSM) [13], to measure the textual similarity between bug reports and source files using information from similar bugs that had previously been fixed. The weighted sum of the two rankings was used to locate the relevant files for a bug. BugLocator was a famous technique that attempted to rank relevant source files for bug reports [14]. Another one of the most promising bug localization methods is spectrum-based fault localization (SBFL) [15–17], which typically uses failed and passed program spectra to assess the risks of all program entities. However, it does not clearly distinguish the different certainty degrees between the passed spectrum and the information associated with the failure spectrum, which may give rise to unreliable fault localization. Xie et al. [17] put forward an improved method to increase the prediction accuracy of SBFL by eliminating uncertain information. All statements are divided into two groups in accordance with different levels of suspiciousness, and then, different evaluation schemes are used for the two groups. Experimental studies indicate, for some SBFL technologies, that this work can considerably improve their performances within some cases, while in other ones it still works without performance deterioration.

A deep neural network has made significant breakthroughs in many fields such as artificial intelligence and text processing [18] and has also achieved good results in bug localization. For example, Lee et al. [19] hold the belief that most of the previous studies only focused on open-source projects without considering in-depth learning technologies and proposed the use of convolutional neural network and word embedding technology to build an automatic fault detection system. Huo et al. [20] proposed one new convolutional neural network, NP-CNN. It utilizes vocabulary and program structure information to learn unified features from natural language and program language source codes and automatically locates potential erroneous source codes according to bug reports. Experimental results in widely used software projects show that NP-CNN is rather superior to current state-of-the-art tools in locating buggy source files. Li et al. [21] proposed a bug prediction framework based on convolutional neural network (DP-CNN), which utilizes deep learning to achieve effective feature generation. Based on the evaluation of methods of seven open-source projects in the bug prediction, experimental results showed that the efficiency of DP-CNN improved by an average of 12%. Meanwhile, the deep learning model combined with IR models also achieved great success. Lam et al. [22] proposed a new approach that combines the deep neural network (DNN) with information retrieval technology rVSM, which
collects features about the textual similarity between bug reports and source files [9, 23]. DNN is used to learn to associate terms in bug reports with potentially different code tokens and terms within source files and documents. The results demonstrate that DNN and IR can complement each other well and obtain higher bug localization accuracy than any single model.

3. Preliminary Study

In this section, a pretrained model is introduced firstly. Then, we introduce information retrieval-based models, which are rapidly developed in recent years. Finally, the semantic matching model and relevance matching model are introduced.

3.1. Pretrained Model. The ALBERT (A Lite BERT) model proposed by Lan et al. [24] is a lightweight pretrained language model based on the BERT model, which adopts the same bidirectional transformer encoder (TRIM) structure as the BERT model. ALBERT greatly reduces model parameters through parameter sharing, matrix decomposition, and other technologies. ALBERT substitutes NSP (next sentence prediction) loss by SOP (sentence order prediction) loss to improve the performance of downstream and shorten the training time using sharing parameters.

ALBERT model only uses the left side encoder of transformer [12]. The encoder structure consists of multi-head attention, add and norm, and feedforward neural network. The multi-head self-attention module input has three vectors: Q (Query), K (Key), and V (Value). As Q vector, each word in its context is represented by K vector, and the similarity between Q vector and each K vector is regarded as the attention weight; then, the weight of each word in context is integrated into the original V of the target word.

\[
\begin{align*}
\text{Attention}(Q, K, V) & = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V, \\
\text{where the dimension of Q and K vectors is } d_k.
\end{align*}
\]

3.2. IR-Based Model. Using the IR-based method, bug localization can be completed at a low cost of text analysis. This method is later applied in collaborative filtering to cooperate with the deep learning model so as to acquire more precise matching score. In all kinds of IR models, vector space model (VSM) is widely used for bug localization. In the traditional bug localization methods based on VSM, the similarity score between source file \( s \) and the bug report \( r \) is obtained by cosine similarity, which is shown in the following equation:

\[
\text{Score}_{\text{sim}} = \cos(r, s),
\]

\[
= \frac{\vec{v}_r \cdot \vec{v}_s}{|\vec{v}_r| |\vec{v}_s|},
\]

where \( v_r \) and \( v_s \) represent the weight vectors of bug report \( r \) and source file \( s \), respectively.

The weight of a word in a document can be obtained according to term frequency (TF). The calculation formula of TF-IDF is as follows:

\[
\begin{align*}
tf(t, d) & = \frac{f_{td}}{N}, \\
idf(t) & = \log \left( \frac{N}{df_t} \right), \\
u_{r,d} & = tf(t, d) \times idf(t).
\end{align*}
\]

At present, there are many variants based on different TFs (term frequencies) to improve the traditional vector space model, such as the revised vector space model (rVSM) proposed by Zhou et al. The formula is as follows:

\[
tf(t, d) = \log(f_{td} + 1).
\]

where \( f_{td} \) represents the number of occurrences of the word \( t \) in the source file \( d \), \( N \) represents the number of all source files in corpus, and \( df_t \) represents the number of files containing the word \( t \) in the source file.

The conventional VSM is suitable for sorting out short texts, while the rVSM considers document length, thus allowing longer source files to have a higher prioritization rank within the ordering. The function \( g(x) \) is as follows:

\[
g(M) = \frac{1}{1 + e^{-N(M)}}.
\]

The formula is a logistic function. The longer the document, the bigger the function value. In addition, we have standardized \( M \) of each document, and the formula is as follows:

\[
N(x) = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}},
\]

where \( x_{\text{max}} \) and \( x_{\text{min}} \) represent the maximum and minimum values in data set \( x \), respectively. The calculation formula of rVSM is as follows:

\[
rVSM\text{Score}(r, s) = g(M) \times \cos(r, s).
\]

3.3. Semantic Matching and Relevance Matching. In this section, we introduce two categories of matching in NLP tasks: semantic matching and relevance matching [2] firstly. Then, we discuss the differences between them and their respective applicable scenarios. The necessity of using two matching models simultaneously in the data set is explained.

There are two types of text matching: semantic matching and relevance matching. Semantic matching is to judge the matching degree of two pieces of text according to semantic information. The semantics are usually homogeneous. For example, both of the sentences are composed of natural language. In contrast, the relevance matching requires the identification of documents related to the given query, which is typically keyword-based.

At present, there are different matching models for semantic matching and relevance matching, respectively.

(1) Representation-based matching model [2] shown in Figure 1, which is used for semantic matching, focuses on encoding text. Its basic idea is to vectorize
text first and then calculate similarity. Representational algorithm directly calculates semantic similarity through vector distance or output classification layer using encoded vector after passing through embedding layer and encoding layer. Typical network structures include DSSM and CDSMM.

(2) Interaction-based matching model [2] shown in Figure 2, which is used in relevance matching, does not need to construct a single- or multi-granularity representation for texts but enables two pieces of texts to interact with each other in order to establish basic matching signals, such as term-level matching signal. Then, the deep neural networks are used to learn the global matching based on these local matching signals.

Semantic matching and relevance matching have their own emphasized factors[2]. Semantic matching emphasizes three factors: similarity matching signals, compositional meanings, and global matching requirement, while relevance matching emphasizes three another factors: exact matching signals, query term importance, and diverse matching requirement.

Next, two common types of bug reports are discussed below and we will analyze their respective characteristics and applicative matching models detailedly.

3.3.1. Multiple Component Bug Report. The description of common bug reports is usually composed of multiple components. Figure 3 shows 322293 bug report in SWT project consisting of multiple components, including natural language, code snippet, and stack trace, which have different syntax structures. Apparently, it is not enough to treat multiple components in bug reports as the same type and then represent them through a unified feature extraction model. It is difficult for deep learning model to obtain accurate semantic representation of bug reports consisting of multiple components.

For this type of bug report, the relevance matching model can work well. As there is at least one class, method, variable, or annotation term in such multicomponent bug reports, source files with these terms have a high probability of having the problem described in the bug report [1]. In such cases, the accurate matching between the keywords in the source file and the keywords in the bug report becomes the most important signal in bug localization, which is the factor emphasized by relevance matching. This also explains why some traditional IR-based models (mainly based on exact matching between words), such as BugLocator, can work quite well.

In addition, the mixing of multiple components can greatly increase the length of bug reports, making it more difficult to learn semantic representations. For this type of bug report, some IR-based approaches show the usefulness of considering different components. Moreno et al.[25] improved the text retrieval model by calculating stack traces in reports and structural similarities between code elements in program dependency diagrams. Thus, it can be seen that using the local matching feature of the association matching model, only considering specific components such as the stack trace of the above example will play a great role in improving the accuracy of the model to deal with such bug reports.

3.3.2. Single Component Bug Report. We find that bug reports consisting of a single component also exist in the data set. Figure 4 shows 55776 bug report in SWT project, which only consists of natural language. Comparing the bug reports in this type with the source files where the bug exists,
we can find that few words can be exactly matched. In this case, there is only a semantic relationship between bug reports and source files. It poses a great challenge to the relevance matching model based on locally exact matching signals.

Semantic matching usually identifies the semantics of texts as a whole and infers about their semantic relationships. For example, “cheap” and “low price” do not overlap in terms so that similarity score calculated by BM25 [26] or TF-IDF is 0, while the semantic matching model does not have this problem. Semantic matching mainly solves the problem that there is no text overlap when calculating similarity, which leads to matching failure. In addition, we found that the semantic matching has drawbacks. Semantic matching emphasizes global matching requirements, while error points in source files may be in code fragments, such as classes or methods, in bug localization tasks. Therefore, the relevance matching model will focus on the noise information in other code details, which will influence the final weight assignment [27]. In addition, semantic matching models, which generally get sentence representation, lose semantic focus and are prone to semantic bias, making it difficult to measure the contextual importance of words.

Therefore, we use a pooling layer [28] in models to avoid semantic bias issues. We split the source files into same-size snippets (150 tokens per snippet in our experiments), and the model only extracts the most relevant k fragments to get the final matching score, which alleviates the disadvantages brought by global matching to some extent.

4. Approach

4.1. Data Preprocessing. In computer science, AST (abstract syntax tree) [29, 29–31] is an abstract tree-like representation of grammatical structure of source codes, in which each node corresponds to a syntactic structure within codes; that is, the leaf nodes represent tokens in the source codes and non-leaf nodes represent syntactic structures, such as the function definitions and function expressions.

SBugLocater, whose overall framework is provided in Figure 5, parses the source codes into AST using Javalang, a pure Python library adept in processing the Java language. It provides a Java-specific lexical parser and parser that map each syntactic structure into a class object. Firstly, the entire source files are mapped to the compilation unit object, the root node of AST through which all nodes can be traversed. Given the path to the source codes of the software, the token sequence for all class files in the software can be output. Given a path P, the PARSE-AST function will traverse all source files in this path and parse each source code file into an AST in turn.

AST parses source files and converts syntactic structures into nodes of the AST. Normally, some nodes are specific to methods or classes. For instance, assignments and internal type declarations cannot be generalized to the entire project as extracting these nodes may dilute the importance of other types of nodes. To better extract semantic information, three kinds of typical token nodes are chosen from AST nodes.

4.2. Relevance Matching Model. The relevance matching layer uses attention-based DRMM [32], which is an enhanced interaction model to extract matching signals of bug reports and source files as shown in Figure 6.

Firstly, a scaled dot-product attention score is calculated for each bug term relative to all source code tokens.

$$a_{i,j} = \text{softmax} \left( \frac{w(r_{j})^T w(s_{i})}{\|r_{j}\|_2} \right),$$

Here, $w(.)$ represents the word2vec embedding of words. We use their attention score relative to the bug term as weights to calculate the representation of source code tokens.

$$v_{si} = \sum_{m} a_{i,j} w(s_{j}).$$

Then, $v_{si}$ and bug term encoding of L2 normalization are used to perform element-wise product operation to obtain the slice-aware bug term encoding representing $v_{si}$. 

![Figure 4: SWT-55776 bug report.](image-url)
Through the element-wise product between $w(r_i)$ and $v_{si}$ after L2 normalization, the slice-aware bug report representation $\phi(s_i)$ is obtained.

$$\phi(s_i) = \frac{v_{si}}{\|v_{si}\|} \odot \frac{w(r_i)}{\|w(r_i)\|}$$

(10)

Then, the similarity score between bug reports and source file slices is calculated via the dense layer.

$$\text{score}_{j,n} = \sigma(w^T\phi(s_i) + b),$$

(11)

where $w^T$ represents the weight of dense layer, $b$ represents the bias term, and $\sigma$ is the activation function.

Afterwards, the top-$k$ score can be obtained through $k$-max pooling layer, in which way the feature information of the first $k$ segments with the highest relevance can be retained for subsequent stages.

$$\text{Score}_k = w^T(k\max(\text{Score}_{j,1}, \ldots, \text{Score}_{j,N})) + b.$$ 

(12)

Ultimately, the final matching score is weighted by IDF of each token in bug report.

$$\text{Score} = \sum_j \text{idf}_j \times \text{Score}_j.$$ 

(13)

4.3. Semantic Matching Model. In the semantic matching layer (Figure 7), two ALBERT models are trained as encoders, respectively. The source file is split into slices, which contain 150 tokens in length. These slices and bug report are input into the corresponding ALBERT model to obtain the representation vectors. For slices in source files, we can obtain the representation vector ahead of time and put it in the index [33]. After the bug report is submitted, calculating is performed in real time to accomplish the recall task for related bug reports.

Eventually, the $k$-max pooling layer is still used to extract the feature information of the $k$ slices with the best semantic matching performance in all code slices for use in the subsequent stages and the final semantic matching score is obtained through the dense layer.
4.4. IR Model Based on Collaborative Filtering. A collaborative filtering algorithm has been widely used in recommendation systems. Collaborative filtering recommendation can be divided into three types: user-based collaborative filtering, item-based collaborative filtering, and model-based collaborative filtering. User-based collaborative filtering mainly considers the similarity between users. User-based collaborative filtering mainly considers the similarity between users. Firstly, it identifies similar users of target users and recommends items with the highest scores to the target users based on the analysis of the items that similar users are fond of while item-based collaborative filtering is similar to user-based collaborative filtering, which finds out similar items of the target users’ favorite items and recommends them to the users.

Here, the recommendation of collaborative filtering is applied to bug localization task and the revised vector space model is used to calculate text similarity between bug reports and source files, which cooperates with the previous deep matching model to obtain more accurate correlation between the two.

If bug report A highly resembles bug report B when A’s bugs are being located, the source file corresponding to bug report B is also highly likely to have A’s bugs.

For the given bug report r and source file s, firstly all bug reports related to source file s are found to form the bug report set br (r, s) and then all the texts in the bug report are merged to calculate the text similarity between the synthesized text and the report r based on rVSM mentioned in the section, and the result is regarded as collaborative filtering score of

\[ \text{Score}_{cf} = \text{rVSM}(r, br(r, s)). \] (14)

4.5. Feature Fusion Layer. After obtaining the matching score of the deep matching model and the collaborative filtering score of the IR model, the fusion layer fuses various features through the dense layer [35].

\[ \text{Score}_{\text{final}} = \sigma(w[\text{Score}_\text{Relevance}; \text{Score}_\text{Semantic}; \text{Score}_{cf}] + b). \] (15)

Since there are usually only a few source files related to a bug in bug localization, there may be hundreds of unrelated source files. This is a highly imbalanced data set, and the accuracy will be difficult to raise if model treats it in the usual way. Therefore, we introduced the focal loss [36] function to optimize the calculation of loss.

To have a better understanding of focal loss, the cross-entropy loss function is first introduced as shown in the following formula:

\[
L = -y \log \tilde{y} + (1 - y) \log (1 - \tilde{y}),
\]

where \( y \) equals 1 means the sample is positive class and \( y \) equals 0 means the sample is negative class. \( \tilde{y} \) is the output of the activation function, and its value is between \([0, 1]\). Therefore, for the positive sample, the larger the output probability, the smaller the loss. At this point, in the case of imbalance between positive and negative samples:

\[
L_{fi} = \begin{cases} 
-(1 - \tilde{y}) \log \tilde{y}, & y = 1, \\
-\tilde{y} \log (1 - \tilde{y}), & y = 0.
\end{cases}
\] (17)

Focal loss optimizes the cross-entropy loss function by adding a module factor \( \alpha \) and a balance factor \( \alpha \) to balance the uneven proportion of positive and negative samples on the original basis as shown in the following formula:

\[
L_{fi} = \begin{cases} 
-\alpha(1 - \tilde{y}) \log \tilde{y}, & y = 1, \\
-\tilde{y} \log (1 - \tilde{y}), & y = 0.
\end{cases}
\] (18)

where the weight value of category 1 is \( \alpha \) and that of category 0 was \( 1 - \alpha \).

For bug localization tasks, the following formula can be used based on the above formula:

\[
L_{bc} = -\alpha y_i^k (1 - \tilde{y}_i^k) \log \tilde{y}_i^k - (1 - \alpha) (1 - \tilde{y}_i^k)(\tilde{y}_i^k) \log (1 - \tilde{y}_i^k),
\]

where \( y_i^k \) represents the related label between the ith bug report and the kth source file, \( \tilde{y}_i^k \) represents the predicted value of whether the ith bug report and the kth source file are related, \( \alpha \) is the balancing factor, and \( \gamma \) is the rate of weight reduction in simple samples. After that, through the forward propagation of data in the network and the numerical value of the error between the predicted quantification value and
the actual observed value utilizing the above loss function, the information of the loss function is propagated backward and the parameters in the network are updated according to the mentioned criteria. Model training is to find a group of parameters with lower loss in the neural network, and the loss can be used to evaluate the performance of the current parameters of the network.

5. Experimental Setup

5.1. Data Sets. To build and verify the SBugLocater, which is designed for bug localization exclusively, we use four open-source Java projects [7], which are widely used in previous bug localization research works, including the Eclipse UI, which is an integrated development platform user interface, JDT, which is a set of Java development tools for Eclipse, SWT Java widget toolkit, and Tomcat, which is Web application server and servlet container. These data sets are publicly available.

5.2. Experimental Design. In this experiment, these data sets are divided into training set (60%), validation set (20%), and testing set (20%). The oldest and latest bug reports are used as validation set and testing set, respectively, and other bug reports are used as training sets. The basic information of bug report data is shown in Table 1. As mentioned above, there is a serious category imbalance in bug localization task; that is, a bug report is only relevant to one or several source files but irrelevant to all other source files and the number of negative cases is much larger than the number of positive ones. Therefore, negative samplings are used for negative cases.

We use 100-dimension pretrained word2vec based on the skip-gram model [37]. During the training process, cosine similarity of word vectors between bug reports and all source files is sorted out. The first 300 unrelated files are then selected as a negative sample, meaning that the actual bug files and the other 300 least similar files are set as a training set along with each bug report. In the validation and testing sets, each bug report is paired with all source files.

We adopt the widely used AdamW [38] optimizer, an improved algorithm based on Adam + L2 normalization. ALBERT model adopts basic configurations, which consist of 16 transformer encoders. The maximum input length of the model is 512. If the input length is less than 512, 0 will be added. To keep input data complete, one source file will be truncated every 200 tokens as input of ALBERT model in the form of one slice, while bug reports will not be truncated only if the length exceeds 512, in which case we take the first 512 tokens.

All experiments are conducted on a Ubuntu 16.04 server with 14-core CPUs, 32 gigabytes of RAM, and 24 gigabytes of NVIDIA RTX A5000 video memory.

5.3. Evaluation Metrics. To evaluate the performance of SBugLocater, three metrics are used for analysis: Accuracy@k, mean reciprocal rank (MRR), and mean average precision (MAP).

MAP, which can measure performances when a query has more than one related document, is more suitable for bug localization because a bug report may have more than two bug files on average. MAP is the average of the AvgP(i) of a set of bug reports Q, each of which is the average of the accuracy values of the bug reports. The formula for bug localization is as follows:

$$\text{MAP} = \frac{1}{Q} \sum_{i=1}^{Q} \text{AvgP}(i),$$

$$\text{AvgP}(i) = \frac{\sum_{j=1}^{M} (T(j)/j) \times \text{ind}(j)}{B(i)},$$

where M is the number of source files that ith bug report retrieved, B(i) is the number of bug files of the ith bug report, and (T(j)/j) is the precision of jth report. ind(j) indicates whether the file located at the rank of j is the buggy file (ind(j) = 1) or not (ind(j) = 0).

MRR is the mean of the reciprocal rank that accumulates the inverse of the position of the first correctly located buggy file for each bug. For a set of Q bug reports, the MRR in this study is computed as follows:

$$\text{MRR} = \frac{1}{Q} \sum_{i=1}^{Q} \frac{1}{\text{rank}_i},$$

where rank_i represents the location of the bug file that is first correctly located for every bug report.

Accuracy@k: the percentage of bug reports in top-k source files have at least one source file correctly recommended.

The higher the accuracy of MAP, MRR, and Accuracy@k, the better the bug localization performance.

6. Experimental Results

In this section, we aimed to answering three research questions.

6.1. RQ1: Model Performance under Different Model Settings. In this section, we select the two most important parameters in SBugLocater, code fragment length and k in k-max pooling, and use Accuracy@k (k = 1, 5, 10) to compare the performance differences under several values. Table 2 shows the change in Accuracy@k (k = 1, 5, 10) with the length of the code fragment. When the length of the code fragment reaches 150, performance reaches the highest; when the fragment length exceeds 150, the performance tends to decrease, possibly because the fragment length is small and it is difficult to extract features from information fragments, while the length is too long, each code fragment carries more noise data. Therefore, we chose 150 as the optimal snippet length.

Next, we compare the Accuracy@k (k = 1, 5, 10) using different k-max pooling. Table 3 shows that when k = 3, 4, and 5, the accuracy of SBugLocater is close to the maximum value. Similarly, the accuracy of SBugLocater decreases as
the $k$ value is too large due to the selection of too many noisy fragments. Therefore, $k = 4$ is selected as the best $k$-max pooling parameter.

6.2. RQ2: How Does the SBugLocater Perform in Bug Localization?

(1) LR+WE [39]: it uses word embedding to enhance the previously proposed learning-to-rank (LR) model [7], which locates buggy files by combining six widely used features.

(2) BugLocator [12]: it obtains the text similarity between bug reports and source files using the optimized VSM method named rVSM.

(3) DeepLocator [40]: it uses rTF-IDuF and AST (abstract syntax tree) to preprocess bug reports and source files and then uses CNN to extract features from them.

(4) DeepLoc [11]: during word embedding, the report summaries, descriptions, and source files are converted into vectors using the word2vec [41], Sent2Vec [42], and weighted average word embeddings. Two CNNs are separately used to extract features from them. Finally, an enhanced CNN is used to combine the features.

The experiment uses Accuracy@$k = 1, 5, and 10$ and the values of MAP and MRR to evaluate bug localization performance.

As can be seen from Table 1, the data set JDT has the largest number of bug reports and source files. SBugLocater achieves Accuracy@$k = 1, 5, and 10$ values of 48%, 73%, and 85%, respectively. That is to say, SBugLocater can locate buggy files for 48% of bug reports correctly when only one source file is recommended, and it can locate buggy files for 73% of bug reports when five source files are recommended. The SBugLocater performs best on the JDT data set, and it is supposed that ALBERT used in SBugLocater is more suitable for large data sets due to the large number of model parameters, while its effect on small data sets seems ordinary. In contrast, although DeepLocator and DeepLoc adopt the deep semantic matching model, when $k = 1, 5,$ and 10, their performance on Accuracy@$k = 39\%, 63\%,$ and 73%, respectively.

Table 4 shows the results of LR+WE, BugLocator, DeepLocator, and DeepLoc on the four public data sets. SBugLocater shows an obvious advantage. The results of the four models (LR+WE, BugLocator, DeepLocator, and DeepLoc) demonstrate that BugLocator has the worst performance, while DeepLoc shows evident improvements in...
the three models. The performance gap between LR + WE and DeepLocator is not obvious. By comparison, SBugLocator that this study proposed is superior in all aspects of performance metrics. For instance, compared with DeepLoc, which is a novel bug localization model, on SWT, the evaluation measures of Accuracy@10, MAP, and MRR are improved by 6.9%, 13.9%, and 17%, respectively.

6.3. RQ3: How Well Do the Three Feature Extraction Models Play in Bug Localization? We conduct the ablation experiment to investigate the contribution of two matching modules. We test by removing one matching module separately.

6.3.1. Relevance Matching. The relevance matching model focuses on the lexical matching between the queries and code snippets. We remove the semantic matching module to evaluate the contribution of the relevance matching model.

6.3.2. Semantic Matching. The semantic matching model aims to capture the semantic correlation between the queries and code snippets. Similarly, we delete the relevance matching model to figure out the contribution of the semantic matching model.

6.3.3. IR Based on Collaborative Filtering. The IR model based on collaborative filtering aims to be a complement to the two matching model. The collaborative filtering method avoids the direct comparison between source files and bug reports that are not homogeneous but transforms into the comparison of bug reports.

We evaluated the effectiveness of relevance matching layer, semantic matching layer, and IR layer. Table 5 compared relevance matching layer, semantic layer, IR layer, both relevance layer and semantic layer, and SBugLocator (use three layers). The performance of the semantic matching layer is about twice that of the relevance matching layer, and the model achieved better accuracy with the addition of the semantic layer than with a single relevance layer. Compared with the relevance matching layer, Accuracy@10 improved 110% on average. Compared with the semantic layer added with the relevance layer, the model accuracy is improved by an average of 14%. It can be concluded that the semantic matching layer plays a great role in bridging the semantic gap, and relevance matching layer also fills the problem of semantic understanding deviation to a certain extent through the precise matching mechanism of keywords, which shows the effectiveness of the fusion of the two matching models. In addition, by comparing the Accuracy@10 scores of the two matching models before and after adding IR model, the accuracy increases by 13% on average, indicating that IR model has a certain complementary effect on matching model.

6.4. RQ4: How Effective Is the Focal Loss Function in the SBugLocator? To illustrate the superiority of focal loss function, we compare bug localization performance of focal loss function and traditional loss function cross-entropy loss.

The bug localization model using cross-entropy loss function is named CE-SBugLocator, and Table 6 shows the results obtained using test data with the two loss functions.

Table 6 shows the results of SBugLocator and CE-SBugLocator. SBugLocator that uses focal loss function shows evident improvements, especially on SWT and Tomcat, which means that the data imbalance problem of these two projects is more serious. For these four software projects (Eclipse UI, JDT, SWT, and Tomcat), the maximum Accuracy@1 of SBugLocator is 52%, and the maximum @1 of
7. Discussion

7.1. RQ5: Why Does SBugLocater Work? The main challenge to bug localization is the semantic gap between bug reports and source files. Most existing ways use textual similarity based on word frequency rather than semantic information about words and phrases. Unlike textual similarity, DeepLoc associates bug reports with corresponding buggy files based on a deep understanding of semantics, which explains why it performs best of the four comparison models. However, when extracting semantics, today’s popular pretrained models usually outperform the traditional models such as CNN and LSTM.

SBugLocater combines the text relevance matching score and the semantic information matching score, as they can dispose of bug localization problem under different situations. In addition, through the idea of indirect matching, namely collaborative filtering, it avoids the direct comparison between the two but transforms that into the direct comparison based on bug reports, which further improves the positioning performance of the model.

(1) Because of multiple components of some bug reports, this part of data is a big challenge to the way of employing models such as CNN to achieve coarse-grained feature extraction of the abstract representation of the whole bug report. Hence, relevance matching layer of SBugLocater computes an attention-based [43] fragment perceptual code for each

<p>| Table 6: Performance comparison between SBugLocater and CE-SBugLocater. |
|---------------------------------|---------------------------------|-----------------|-----------------|-----------------|</p>
<table>
<thead>
<tr>
<th>Project</th>
<th>Model</th>
<th>Accuracy@k (%)</th>
<th>MAP (k = 10)</th>
<th>MRR (k = 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse UI</td>
<td>CE-SBugLocater</td>
<td>39.3</td>
<td>63.8</td>
<td>70.1</td>
</tr>
<tr>
<td></td>
<td>SBugLocater</td>
<td>43.8</td>
<td>69.0</td>
<td>79.7</td>
</tr>
<tr>
<td>JDT</td>
<td>CE-SBugLocater</td>
<td>40.4</td>
<td>61.0</td>
<td>74.2</td>
</tr>
<tr>
<td></td>
<td>SBugLocater</td>
<td>45.3</td>
<td>70.6</td>
<td>82.4</td>
</tr>
<tr>
<td>SWT</td>
<td>CE-SBugLocater</td>
<td>36.6</td>
<td>63.2</td>
<td>70.9</td>
</tr>
<tr>
<td></td>
<td>SBugLocater</td>
<td>43.7</td>
<td>71.6</td>
<td>80.3</td>
</tr>
<tr>
<td>Tomcat</td>
<td>CE-SBugLocater</td>
<td>48.9</td>
<td>67.0</td>
<td>75.6</td>
</tr>
<tr>
<td></td>
<td>SBugLocater</td>
<td>52.0</td>
<td>74.3</td>
<td>83.1</td>
</tr>
</tbody>
</table>

CE-SBugLocater is 48.9%. For the two evaluation metrics of Accuracy@5 and Accuracy@10, SBugLocater has better bug localization performance than CE-SBugLocater. For SBugLocater, MRR values of four software projects (Eclipse UI, JDT, SWT, and Tomcat) are 0.530, 0.597, 0.576, and 0.602 respectively. For CE-SBugLocater, the four MRR values are 0.490, 0.510, 0.504, and 0.550, respectively. For MAP, the bug localization performance of SBugLocater is better than that of CE-SBugLocater. In summary, SBugLocater based on focal loss can better solve the problem of data imbalance and further improve the bug localization performance.
word in the bug report, using fine-grained matching signals to get the final global match score.

(2) In the meanwhile, on the basis of traditional global semantic analysis, the semantic matching layer abstractly represents the grammar structure of source codes by AST in allusion to the special structure of codes and extracts token sequences that can represent source codes to obtain enhanced semantic information of source files. This is helpful for the ALBERT pretrained model to extract semantic information from source files more accurately.

(3) SBugLocator also proposes another solution to the problem of semantic difference between bug reports and source files. The IR layer wisely avoids direct comparison between the two and uses collaborative filtering instead. The matching degree between the target bug report and the known bug report is calculated by the two with known corresponding relationship. The higher the matching degree is, the more possibly the source files corresponding to the known bug will be recommended.

(4) Due to the large length of source files, the extracted token sequences are usually several thousand in length. Therefore, the token sequences are split into multiple sequence slices in the same length through both matching layers, and the k scores with the largest weight ratio are selected by k-max pooling in the final pooling layer to obtain the matching score after being calculated averagely.

8. Threats to Validity

The experimental results demonstrate SBugLocator’s feasibility; however, we acknowledge some potential threats to the validity of our approach and experiments. Following the suggestions of Wohlin et al. [44], we discuss threats to internal validity, external validity, and construct validity.

8.1. Internal Validity. The proposed approach converts each sentence in a bug report into a vector, which could be affected by stemming and removal of stop words process. This potential will be investigated in a future study. Both bug reports and source files are transformed into vectors based on word embedding techniques. These techniques make texts from bug reports and source files into adequate input for matching model, retaining their semantic information and saving memory space. However, the performance of the proposed approach relies to some extent on the ability of word embedding. It would be best to test these techniques before adding them to the proposed model. Improving these techniques will also help to enhance our model. We leave this for future studies.

8.2. External Validity. Since the code for LR+WE, DeepLocator, and BugLocator is not published, we implemented them according to the algorithms provided by the papers and achieved similar results. However, the results are not quite the same. Fortunately, Ye et al. [7] and Lin et al. [9] provided results from the same data set. We choose the best results for each project and compare our results with theirs. Finally, the data sets used were obtained from Java projects. The results may not generalize to other projects written in other programming languages. In the future, we intend to refine this model for use in other projects written in different programming languages.

8.3. Construct Validity. This is due to the quality of bug reports [45]. If the bug report does not provide sufficient information or misleading information, the performance of the data link will be further affected. Another threat relates to the suitability of the evaluation metrics. We use Accuracy@k, MAP, and MRR evaluation metrics. These metrics have been used extensively in bug localization studies in the past. Finally, including test files may lead to biased evaluation results [46]. However, according to recent research [47], including test files does not interfere with bug localization if the correct prefix version is used. In subsequent work, we check out the source code files of the prefix version for each bug to ensure that the correct prefix version is used, and hence, including test files does not introduce bias in this study.

9. Conclusion

Most of the existing bug localization methods only pay attention to the correlation of words in bug reports and source files or the correlation degree of semantic information between the two. However, single-angle models tend to have poor performance in partial data that are not suitable. Most approaches focus on improving the semantic gap between bug reports and source files. However, the relevance matching model, which combines local matching and global matching and considers the exact matching of keywords, is rarely studied and applied to bug localization. To solve the problems, we propose a new bug localization framework SBugLocator, which combines relevance matching and semantic matching to meet the matching requirements of different data forms. The IR model based on collaborative filtering is complementary to the other two matching models as well. This has boosted performance of SBugLocator on various types of data sets. We have evaluated SBugLocator on four representative benchmark data sets. The experimental results show that SBugLocator’s performance is superior to the four most advanced localization models, which is living proof of the reliability of SBugLocator.

Data Availability

We use the data sets from five open-source Java projects [11], which are widely used in previous bug localization. In this study, four of them are selected as data sets and each of them consists of source files and bug reports. The four data sets are of different scales.
Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Nos. 61832014, 61902114, and 61977021) and the Key R&D Programs of Hubei Province (Nos. 2021BAA184 and 2021BAA188).

References


