An Imbalanced Deep Learning Model for Bug Localization

Bui Thi Mai Anh
Laboratory of Intelligent Software Engineering
School of Information and Communication Technology
Hanoi University of Science and Technology
Hanoi, Vietnam
anhbtm@soict.hust.edu.vn

Nguyen Viet Luyen
Laboratory of Intelligent Software Engineering
School of Information and Communication Technology
Hanoi University of Science and Technology
Hanoi, Vietnam
luyen.nv152333@sis.hust.edu.vn

***Abstract*—Debugging and locating faulty source files are tedious and time-consuming tasks. To improve the productivity and to help developers focus on crucial files, automated bug localization models have been proposed for years. These models recommend buggy source files by ranking them according to their relevance to a given bug report. There are two significant challenges in this research field: (i) narrowing the lexical gap between bug reports which are typically described using natural languages and source files written in programming languages; (ii) reducing the impact of imbalanced data distribution in model training as a far fewer of source files relate to a given bug report while the majority of them are not relevant. In this paper, we propose a deep neural network model to investigate essential information hidden within bug reports and source files through capturing not only lexical relations but also semantic details as well as domain knowledge features such as historical bug fixings, code change history. To address the skewed class distribution, we apply a focal loss function combining with a bootstrapping method to rectify samples of the minority class within iterative training batches to our proposed model. We assessed the performance of our approach over six large scale Java open-source projects. The empirical results have showed that the proposed method outperformed other state-of-the-art models by improving the Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR) scores from 3% to 11% and from 2% to 14%, respectively.**

***Index Terms*—bug localization, deep neural network, imbalanced data-set, bootstrapping**

I. Introduction

During the software development and maintenance, bugs are inevitable. The effective management of bugs plays an essential role in the guarantee of software quality. Issue tracking systems such as Jira[*], Bugzilla[†], etc. are usually used to store and manage *bug reports* which provide the important information about a software failure including symptom description, environment context as well as specific constraints to reproduce the failure. To fix a newly reported bug, developers have to carefully analyze the bug report and review a large amount of source code files to locate potential buggy files. Thus, bug fixing activities require considerable time and effort, particularly in large-scale software projects with thousands of source files [1]. Automated bug localization is therefore desirable to reduce the maintenance cost and to speed up the productivity of the whole software development team.

[*]www.atlassian.com
[†]www.bugzilla.org

Several bug localization techniques have been proposed, those can be categorized into two directions: (i) *dynamic* and (ii) *static* bug localization. Different to dynamic approaches in which faulty files are located by analyzing execution traces and examining the runtime state of the software system [2], [3], static approaches localize bugs by exploring the relationship between bug reports and source files without requiring a runnable version of the system.

Considering a bug report as a *query* and source files as a set of *documents*, bug localization can be solved using Information Retrieval (IR) techniques [4]–[6]. These approaches measure the lexical similarity between a given bug report and all source files and produce a top ranked documents as results. However, the lexical mismatch between natural languages used to describe bug reports and programming languages used to write source code is one of the main factor that limits the performance of IR-based methods and causes a big challenge for bug localization models [7]. To overcome this issue, recent studies focus on exploiting not only lexical information but also semantic relations between bug reports and source files [8]. With the successful application of deep learning models in different disciplines, deep neural networks (DNNs) [9], recurrent neural networks (RNNs) [10], convolutional neural networks (CNNs) [7], [11] have been used to extract the semantic information from bug reports and source files. Liang et al. proposed to integrate hierarchical structures from source files into CNN model to leverage semantic relation of program statements [12]. Ye et al. [6] improved the performance of bug localization by using a learn to rank model to linearly combine lexical information with domain knowledge features. Dependencies between program entities have been also investigated to augment the accuracy of localizing bugs [1], [13].

We have observed that most studies on the application of deep learning models for bug localization focus on extracting the underlying semantic information from bug reports and source files (i.e., feature extraction). They suffer from a very high cost of training due to a high dimension of input data and might ignore some useful information come from lexical surface. On the other hand, almost existing models have neglected the impact of imbalanced data-set on the overall performance of bug localization. It is observable that there is a far fewer of source files that relate to a given bug report while the majority of them are not relevant. In this context, the skewed distribution of training data leads deep learning models towards the majority class (i.e., irrelevant source files), resulting in a poor performance on the minority class (i.e., relevant source files).

In this paper, we propose a deep neural network to capture a non-linear combination of essential features statistically extracted from bug reports and sources files. More concisely, we adopt revised vector space model proposed by Zhou et al. [4] to extract lexical features and compute the lexical similarity of each pair of bug report and source file. The semantic similarity is the second feature which captures the semantic relation between word embedding vectors of bug reports and source files. Finally, we take into account useful information from historical bug report and code change history to more effectively localize bugs. All these features are computed then fed into the DNN to speed up the training process. To address the problem of imbalanced learning, we propose a hybrid strategy which combines cost-sensitive learning approach with data manipulation method to rectify training data instance of the minority class. More precisely, we adapt the focal loss function [14] to our DNN model with the aim of reducing the dominant effect of the majority class by giving more weight to the minority one. Moreover, we improve the performance of imbalanced learning by adjusting the distribution of data instances during training phase. The idea is come from the fact that most deep learning approaches (including DNNs) split the training dataset into several mini-batches during training phase. It is not always the case that every mini-batch contain positive samples (i.e., from minority class), which might lead the training model towards the majority class. To overcome this phenomena, we apply a bootstrapping method with the aim of balancing generated mini-batches for each training iteration.

The rest of this paper is organized as follows: In Section II, related work in bug localization and imbalanced learning is discussed. Section III introduces principles about deep neural networks and focal loss function. Our proposed approach will be presented in Section IV. Sections V and VI discuss about the empirical settings, evaluation metrics and some results of our approach comparing with other state-of-the-art models. Finally, Section VIII concludes with some future directions.

II. Related Works

A. IR-based models

Early studies formalize the problem of bug localization as an information retrieval task in which source files are ranked following their relevance score to a given bug report. Lukins et al. [15] and the following-up work of Nguyen et al. [5] applied LDA (Latent Dirichlet Allocation) to compute the topic distribution of bug reports and source files, according to which a ranking list of source files is produced. Zhou et al. [4] proposed a revised Vector Space Model (rVSM) to compute the textual similarity between bug reports and source files regarding similar bugs and document length as bugs tend to be appear in large source code files. The combination of retrieval models and other features has been also explored in several researches. Ye et al. [6] adopted a learn to rank approach to linearly combine textual features with domain knowledge features using a weight function. Saha et al. [16] combined retrieval information with structured similarity measured by the distance between stack-traces and source code elements. Although these combinations help improve the performance of traditional IR approaches, they do not address the lexical mismatch between bug reports and source files.

B. Deep Learning models

To narrow the lexical gap, many studies focus on extracting semantic relationship between bug reports and source files using deep learning models. Lam et al. [9] used two deep neural networks: the first to extract the semantic relevancy score between bug reports and source files with the help of an auto-encoder to reduce the dimension, the second to learn the final score from the combination with retrieval features. Word embeddings [17], [18] have been adopted in several researches to take into consideration the semantic representation of bug reports and source files. Xiao et al. [8] used different CNNs to detect underlying semantic features from word embedding vectors of bug reports and source files. They considered particularly bug fixing recency and frequency as additional features to improve the loss function of the proposed CNN model. A multi-dimensional CNN model was proposed by Wang et al. [19] to learn the complex and non-linear relationship between five features extracted from bugs report and source files. The proposed model learns the deep semantic from a feature matrix of all bug reports and source files by combining different CNNs with appropriate configurations (i.e., filters). The most recent work of Yuan et al. [13] focuses also on a two-CNN model to capture the textual similarity from IR-based representation (i.e., VSM) as well as to leverage the dependency between different elements of source files. It is indisputable that deep learning models allow to extract the hidden semantic relation between bug reports and source files, resulting in a considerable improvement of performance on localizing bugs. However, most studies have ignored the impact of imbalanced learning which is a major reason for unsatisfactory performance of deep learning models [20].

C. Imbalanced Learning

The problem of imbalanced learning has drawn an increasing attention and attracted significant research efforts in machine learning and deep learning of many areas [21], [22]. There are two general approaches to deal with imbalanced classification: (i) data sample manipulation including over-sampling and upper-sampling and (ii) cost-sensitive learning [21]. The former may make the model susceptible to overfitting phenomena with over-sampling or may discard important samples for feature learning with under-sampling.

The latter aims to modify existing learning algorithm to give more weight to the minority class.

In the field of software engineering, cost-sensitive re-weighting approaches are often utilized to tackle the problem of imbalanced dataset. Wang and Yao [23] have proposed a dynamic version of AdaBoost.NC model for software defect prediction which adjusts automatically parameters by considering the imbalance degree between positive and negative data instances. An Artificial Bee Colony-based ANN (artificial neural network) model was proposed by Arar and Ayan [24] to optimize the learning weights by adding a cost sensitivity to the ANN error function during the training of the software defect prediction model. To the best of our knowledge, addressing the problem of skewed learning dataset is rarely investigated in the field of bug localization. Huo et al. [11] have proposed NP-CNN consisting of two CNNs to extract features from bug reports and source files. The fully connected layer is trained using an unequal misclassification cost according to the imbalance ratio between relevant and irrelevant source files to a given bug report. This approach has been also adopted by Wang et al. [19] in their proposed multi-dimensional CNN model for bug localization. Liu et al. [25] applied a focal loss function to their propose CNN model without considering the imbalanced degree in the training dataset. Inspired from state-of-the-art researches to tackle the problem of imbalanced classification, we propose a hybrid strategy which considers a cost-sensitive learning approach together with data sample manipulation. As will be shown in Section VI, the proposed strategy produces better performance on locating buggy files comparing to traditional cost-sensitive approaches.

III. Preliminaries

A. Deep Neural Network

Deep Neural Networks are computational models, inspired by biological processes, which consist of many simple processing units (also known as *neurons*) which can work in parallel and are arranged in interconnected layers [26]. An DNN contains an input layer, an output layer and might have multiple hidden layers in sequence. Feature vectors extracted from the input data are fed into the input layer. Each neuron in a deep neural network plays as a function that takes a weighted sum of all incoming signals. The output value of all neurons in the network is computed through the fed-forward mechanism from the input layer to the output layer, as described in Equation 1.

$$h_{kj} = \mathcal{F}\left(\sum_{i=1}^{N^{k-1}} w_k(i,j)x_i^{k-1} + b_{kj}\right) \tag{1}$$

where h_{kj} is the output of j^{th} node of the k^{th} layer, N^{k-1} is the number of nodes of the $(k-1)^{th}$ layer and x_i^{k-1} is the value of the i^{th} incoming node from the $(k-1)^{th}$ layer. $\mathcal{F}$ is the activation function such as *sigmoid* function [26]. b_{kj} is the bias value for the j^{th} node of the k^{th} layer. $w_k(i,j)$ is the weight of the connection from the i^{th} node of the $(k-1)^{th}$ layer to the j^{th} node of the k^{th} layer.

Weights of the connection between DNN layers can be learned by minimizing a *loss function* (akin to *cost function*) based on stochastic gradient descent [26]. As the localization of buggy files can be considered as a binary classification task in which source files are categorized into two classes: *relevant* and *not-relevant*, to a given bug report, the cross entropy (CE) loss function is typically adopted in most studies [7], [9], [19] (Equation 2).

$$\begin{aligned} \mathcal{L}_{CE} &= \frac{1}{N_b}\sum_{i=1}^{N_b} cost_i \\ cost_i &= -(\delta \log(y_i) + (1-\delta)\log(1-y_i)) \end{aligned} \tag{2}$$

where $\mathcal{L}_{CE}$ is the overall loss of a mini-batch of N_b samples, $cost_i$, y_i are the cost value and the output probability of sample i, respectively. δ has value 0 or 1 depending on negative (i.e., not-relevant to bug reports) or positive sample (i.e., relevant to bug reports). It can be observed that the CE loss minimizes the training error by considering an equal importance on all individual samples and classes. A skewed class distribution, therefore, might lead to a poor classification performance on minority class.

B. Focal Loss Function

The focal loss function was first introduced by Lin et al. [14] to tackle the problem of imbalanced learning by adding a modulating factor to the cross entropy loss. The formula of focal loss for binary classification can be described in Equation 3.

$$cost_i^{FL} = \begin{cases} -\alpha(1-y_i)^\gamma \log(y_i) & \delta = 1 \\ -(1-\alpha)y_i^\gamma \log(1-y_i) & \delta = 0 \end{cases} \tag{3}$$

The modulating factor $\gamma > 0$ adjusts the dominant effect of the majority class (i.e., the label $\delta = 0$). α-balanced variant factor is added to balance the proportion of positive and negative samples. The total loss of a training mini-batch is computed by taking the average summation of all sample-level focal loss. By design, the focal loss allows to focus more on the minority class and to reduce the weights of a large number of negative samples during training phase.

IV. Our Proposed Approach

A. Data Preprocessing

A bug report obtained from bug tracking systems typically consists of a summary and a description. We first combine them into one document as the textual representation of a bug report, which is then parsed into words using whitespaces and filtered to remove punctuation and standard stopwords (e.g., "a", "the"). The preprocessing of source files is subjected to the removal of programming language keywords (i.e., java keywords), numbers and punctuation. In order to dampen the redundancy of a huge mass of lines of code, we build the textual description of a source file by extracting only comments and identifiers of classes/methods as well as attributes. Compound words in bug reports and source files are split into individual words based on the capital letters if any. For example, `VirtualCamera` should be split into `Virtual` and `Camera`. Finally, all the words are converted to their stem to reduce derivationally related words.

B. Feature Engineering

Let $\mathcal{B} = \{r_1, r_2, ..., r_N\}$ denotes the set of bug reports and $\mathcal{S} = \{s_1, s_2, ..., s_M\}$ be the collection of source files of the software project where N and M are the total number of bug reports and source files, respectively. We extract five following features for a pair (r, s) with $r \in \mathcal{B}$ and $s \in \mathcal{S}$.

1) Lexical Similarity: Inspired from the Vector Space Model (VSM) technique which has been used in the works of Zhou et al. [4] and Ye et al. [6], we calculate the lexical similarity based on the frequency of tokens appearing in the bug report (*Term Frequency-TF*) and the number of source files where these tokens appears (*Inverse Document Frequency-IDF*). The lexical similarity between two VSM vectors V_s and V_r is computed using the cosine similarity as in Equation 4.

$$\mathcal{L}_{sim}(r,s) = \frac{\overrightarrow{V_r} \cdot \overrightarrow{V_s}}{\|\overrightarrow{V_r}\| \times \|\overrightarrow{V_s}\|} \tag{4}$$

The lexical similarity score favors large source files by incorporating the length of the source file s as large source files tend to be more error-prone than shorter ones [4].

2) Semantic Similarity: To address the lexical mismatch between bug reports and their related source files, we adopt the word embedding techniques [27] to explore the semantic relationship among words based on the context in which they appear. In our work, we adopt a pre-trained GloVe model[‡] to build word embedding vectors for bug reports and source files.

Given a document d of n words (either a source file s or a bug report r), it can be represented as a $n \times m$ matrix in which the row i encodes the i-th word of d. In order to easily measure the document similarity, we transform this matrix to a m-dimensional vector by taking the weighted average of n rows:

$$\vec{\mathcal{V}_d} = \frac{\sum_i w_i \vec{v_i}}{n} \tag{5}$$

where w_i, $\vec{v_i}$ denote the *tf-idf* weight and the word embedding vector of the i-th word and n is the total number of words in the document d. Given a bug report r and a source file s, the semantic similarity is calculated through the cosine similarity of their word embedding vectors as follows:

$$\mathcal{S}_{sim}(r,s) = \frac{\overrightarrow{\mathcal{V}_r} \cdot \overrightarrow{\mathcal{V}_s}}{\|\overrightarrow{\mathcal{V}_r}\| \times \|\overrightarrow{\mathcal{V}_s}\|} \tag{6}$$

3) Similar Bug Reports: As indicated in recent studies, similar bugs may be related to similar source files [6]. Therefore, the similarity between bug reports can be used as an indicator to find related faulty source files.

This score is calculated by measuring the similarity between the under-considering bug report and all the others extracted from the bug fixing history, those who were related to a given source file. Given a bug report $r \in \mathcal{B}$ and a source file $s \in \mathcal{S}$, let $\hat{\mathcal{B}}_s$ be the set of previous bug reports for which s was fixed, $\hat{\mathcal{B}}_s = \{\hat{r} | \hat{r} \in \mathcal{B}, s\ \mathit{was\ fixed\ to\ resolve}\ \hat{r}\}$. The similar bug report score $\mathcal{R}$ is defined as follows:

$$\mathcal{R}(r,s) = \frac{\sum_{\hat{r} \in \hat{\mathcal{B}}_s} \mathcal{L}_{sim}(r, \hat{r})}{|\hat{\mathcal{B}}_s|} \tag{7}$$

where $\mathcal{L}_{sim}(r, \hat{r})$ is the lexical similarity between r and $\hat{r}$, calculated by the cosine similarity between two VSM vectors V_r and $V_{\hat{r}}$. $|\hat{\mathcal{B}}_s|$ denotes the number of bug reports in $\hat{\mathcal{B}}_s$.

4) Code Change History: Recent studies has acknowledged that the changes performed in buggy source files to resolve previous bug reports may become reasons of new coming bug reports [6]. This motivates us to utilize the source code change history to effectively predict fault-prone source files. Indeed, a source file which was fixed very long time in the past or never changed is less probably to contain bugs than a file which was changed recently. We then evaluate the importance of a source file $s \in \mathcal{S}$ in term of fixing recency for a given bug report $r \in \mathcal{B}$ by calculating the elapsed time (in *days*) between the last commit day of s and the day at which r was reported.

$$\mathcal{H}(r,s) = \frac{1}{t_{elapse}(s,r)} \tag{8}$$

where $t_{elapse}(s,r) = \min_{s \in \mathcal{C}_s^\mu}(t_r^{report} - t_s^{commit})$, $\mathcal{C}_s^\mu$ is the set of different versions of s which were committed to the bug tracing system in μ days before receiving the bug report r at time t_r^{report}. The parameter μ is chosen depending on the fixing frequency of software projects. Equation 8 indicates that it is highly likely to recommend the most recent buggy files for a new bug report.

5) Bug Fixing Frequency: The frequency at which a source file has been fixed can also be used as an indicator to measure its error-proneness. Given a pair $< r, s >$ of bug report r and source file s, its bug-fixing frequency score is defined as the number of times the source file s has been fixed in a duration of μ days before the appearance of bug report r:

$$\mathcal{F}(r,s) = |\mathcal{C}_s^\mu| \tag{9}$$

where $|\mathcal{C}_s^\mu|$ denotes the length of set $\mathcal{C}$ which is also known as the number of times s was changed.

C. The Proposed DNN Model for Bug Localization

1) Feature Scaling: Features extracted from bug reports and source files are normalized to the same scale using min-max normalization Given a feature ψ, let ψ_{max} and ψ_{min} be the maximum and the minimum values of feature scores observed from the training dataset. Considering the testing dataset, it might be the case that the feature ψ may have values which are greater than ψ_{max} or less than ψ_{min} computed over the training dataset. For this reason, the value of ψ is scaled as follows:

$$\psi_{nor} = \begin{cases} 0 & \text{if } \psi < \psi_{min} \\ \frac{\psi - \psi_{min}}{\psi_{max} - \psi_{min}} & \text{if } \psi_{min} \le \psi \le \psi_{max} \\ 1 & \text{if } \psi > \psi_{max} \end{cases} \tag{10}$$

[‡]The GloVe model is trained on the Common Crawl corpus (https://commoncrawl.org)

where ψ is one of five aforementioned features (see Equation 4, 6, 7, 8 and 9).

2) Feature Combining by DNN: The five proposed features: *lexical similarity*, *semantic similarity*, *similarity to historical bug reports*, *code change history* and *bug fixing frequency* serve as different indicators to link a given bug report $r \in \mathcal{B}$ to its most relevant source files from $\mathcal{S}$. The performance of the bug localization model depends significantly on the way how these features are combined. In this study, we propose a DNN model to capture non-linear and complex relationships among the features for effective bug localization.

A pair bug report-source file, represented as a feature vector $v(r, s)$ of five elements, is fed into the DNN model with an input layer (of five nodes), k hidden layers (the number of node of each hidden layer is empirically optimized) and an output layer with one node to indicate the final relevance score of (r, s). We apply the focal loss function introduced in Section III-B to our proposed DNN model to tackle the problem of imbalanced dataset based on cost-sensitive learning approach.

3) DNN with Bootstrapping: The training process of deep learning models is typically speeded up by splitting the training dataset into several mini-batches. Each mini-batch contains an equal number of samples (called batch-size) which are randomly picked up from the training dataset. Because of a skewed class distribution of source files, it is possible that a mini-batch does not contain any positive samples, resulting in a poor classification performance on predicting relevant source files for a given bug report. To further improve the performance of our bug localization model on imbalanced dataset, we propose to apply a bootstrapping to the proposed DNN.

Let $\mathcal{N}_{neg}$ and $\mathcal{N}_{pos}$ be the number of negative and positive instances in the imbalanced training set, respectively. The mini-batch generation process is depicted in Algorithm 1.

Algorithm 1: Pseudo code of DNN with Bootstrapping

```
Input  : Negative set with $\mathcal{N}_{neg}$ negative instances,
         Positive set with $\mathcal{N}_{pos}$ positive instances
Output: $K$ mini-batches with equal size $N_{mb}$ for the
         DNN model
1  Divide $\mathcal{N}_{neg}$ negative instances into $K$ subsets
     $\mathcal{X} = X_1 \bigcup X_2 \bigcup ... \bigcup X_K$, each with $S_n$ negative
     instances
2  $Temp_{pos} \leftarrow$ Positive Set
3  foreach $i \in [1..K]$ do
4      $pos_i \leftarrow \emptyset$
5      foreach $j \in [1..S_p]$ do
6          randomly pick up an instance $t \in Temp_{pos}$
7          $pos_i \leftarrow pos_i \bigcup t$
8          $Temp_{pos} \leftarrow Temp_{pos} - t$
9      end
10     $X_i \leftarrow X_i \bigcup pos_i$
11 end
12 return The set $\mathcal{X}$ of $K$ mini-batches
```

Each generated mini-batch has an equal size $S = S_n + S_p$ where S_n, S_p are respectively the number of negative and positive instances in each mini-batch. First, all the negative samples of the training set is divided into $K = \lfloor \mathcal{N}_{neg}/S_n \rfloor$ distinct subsets $\mathcal{X} = X_1 \bigcup X_2 \bigcup ... \bigcup X_K$ where $X_i \bigcap X_j = \emptyset$ $(i \neq j)$ (line 1). Each subset X_i consists of S_n negative instances. The positive instances of each mini-batch are created by randomly picking up from the original set of all positive instances (lines 5-9). The set of S_p picked-up positive instances will combine with S_n negative instances to form a mini-batch (line 10). The same process is repeated until obtaining K mini-batches for training the DNN model.

The bootstrapping method ensures that each positive instance in the training set has an equal probability to be selected, while avoiding the chance of having no positive instances in some mini-batch.

V. Empirical Settings

A. Evaluation Metrics

The performance of our proposed approach will be assessed through three evaluation metrics which have been widely utilized in many studies.

- *Top-k accuracy* measures the percentage of bug reports at which at least one buggy file is correctly found in the top k rank. Typically, the model is measured by top-1, top-5, top-10 and top-15 files.
- *Mean Reciprocal Rank (MRR)* measures the average of the reciprocal rank for all bug reports. The reciprocal rank of a bug report is the inverse rank of its first correctly-located buggy file (Equation 11).

$$MRR = \frac{1}{N} \sum_{r \in \mathcal{B}} \frac{1}{first(r)} \tag{11}$$

where N is the number of bug reports in $\mathcal{B}$, $first(r)$ denotes the ranking position of the first relevant source file found in the ranking list, with respect to r.

- *Mean Average Precision (MAP)* is used to evaluate the mean of all *average precision* values ($AvgPre(r)$) (Equations 12, 13 and 14).

$$MAP = \sum_{r \in \mathcal{B}} \frac{AvgPre(r)}{N} \tag{12}$$

where N is the number of bug reports in $\mathcal{B}$. The average precision of a bug report r is computed as follows:

$$AvgPre(r) = \sum_{k \in K} \frac{Pre_k(r)}{|K|} \tag{13}$$

where K is the list of top ranking source files in descending order of f values, with respect to r, $|K|$ denotes the number of source files in K. The precision at k-th rank, $Pre_k(r)$ is computed as the number of relevant source files over the top k files:

$$Pre_k(r) = \frac{\text{\# of relevant source files in top } k}{k} \tag{14}$$

B. Benchmark Datasets

For the comparison with other state-of-the-art models, we evaluated the performance of our proposed approach on the same collection of datasets provided by Ye et al. [6]. There is totally 22,747 bug reports collected from six open-source projects including AspectJ, Tomcat, Eclipse UI, SWT, JDT and Birt. Table I shows the overall statistical information of these projects. Their source code is publicly accessible on GitHub.

TABLE I
BENCHMARK DATASETS

Project	Time Range	Number of Bug Reports	Number of Source Files
AspectJ	03/2002-01/2014	593	4439
Tomcat	07/2002-01/2014	1056	1552
Eclipse UI	10/2001-01/2014	6495	3454
SWT	02/2002-01/2014	4151	2056
JDT	10/2001-01/2014	6274	8184
Birt	06/2005-12/2013	4178	6841

C. Training and Testing Data

All the bug reports in each benchmark dataset is arranged in ascending order of their report timestamp. We then split the sorted bug reports into 10 equally sized folds for all projects excepting the smallest one, AspectJ. For AspectJ, we divide it into three folds. As such, $fold_1$ contains the oldest bug reports and the most recent bugs are in the last $fold_{10}$. The model will be trained on fold i and test on fold $i+1$. The final results are obtained by taking the averages of all folds.

In training dataset, we match a bug report will all the source files of the same project and compute the feature vector of each pair of bug report/source file. During training phase, the ground truth output of the proposed DNN is set to 0 for negative sample and 1 for positive one. For prediction, the relevance score is computed for all pairs of testing bug reports and source files and has value in the range $[0..1]$. A source file is considered as buggy file of a given bug report if the output score is greater than 0.5 and vice versa.

D. Model Settings and Hyper-parameter Turning

The proposed DNN model for bug localization, named `ImbalancedBugLoc`, was built on Keras framework. Through different empirical running on six projects, we propose two hidden layers for ImbalancedBugLoc. The first hidden layer has 300 nodes while the second one has 150 with the same activation function `ReLU`. The output layer of one node uses the `sigmoid` activation function. The model is trained on 30 epochs using the SGD to optimize the focal loss function (see Section III-B). The model hyper-parameters including the number of hidden layers, the number of nodes per hidden layer as well as the parameter α of the focal loss function are empirically determined. The detailed experimentation of the parameter α will be introduced in Section VI.

VI. Experimental Results

A. Evaluation Plan

We compare our proposed approach with two IR-based baseline models and three recent deep learning models in which there are two studies, SLS-CNN and MD-CNN, consider the imbalanced classification for bug localization, as follows:

- BugLocator [4] locates buggy files by using textual similarity combining with source file length and previous fixed bugs.
- Learn to rank - LR model [6] adopts a Support Vector Machine for ranking source files for a given bug report while leveraging the integration of domain knowledge features with IR-based model.
- DNNLoc [9] combines a deep neural network with IR-based model to explore the semantic information from bug reports and source files.
- MD-CNN [19] explores the non-linear combination of multi-dimensional features using different CNNs.
- SLS-CNN [25] proposes a unified feature CNN model with focal-loss function to solve the impact of data imbalance.

B. Results

We present our experimental results through the following research questions.

1) RQ1: How does ImbalancedBugLoc model perform comparing with some state-of-the-art models?: Our experiment was conducted on the same datasets using the same procedure and metrics, comparing to five state-of-the-art models aforementioned. Therefore, excepting BugLocator which was re-implemented, we used the public results on these other studies. Table II shows the performance comparison over top-k accuracy (with $k = 1..5, 10, 15$), MAP and MRR. It can be observed that our proposed model ImbalancedBugLoc generally outperforms other models regarding the accuracy. Indeed, ImbalancedBugLoc successful has located 52.5% bugs in AspectJ, 53.2% bugs in Tomcat, 48.5% bugs in Eclipse UI, 40.2% bugs on SWT and 53% bugs on JDT in terms of top-1 accuracy. Comparing to LR Model [6], our proposed model has increased the top-1 accuracy by average 16.95%. The improvement in terms of top-5 accuracy is from 10.9% to 38.3% while the improvement in terms of top-15 accuracy is from 8.8% to 26.8%. Comparing to MD-CNN and SLS-CNN with regard to the top-10 accuracy on three large-scale projects Eclipse, JDT and Birt, ImbalancedBugLoc is 2.2% higher than MD-CNN and 9.4% higher than SLS-CNN with Eclipse, improves 4.9% than MD-CNN and 12.6% than SLS-CNN with JDT, increases 10.2% comparing to MD-CNN with Birt while SLS-CNN did not experiment on Birt. Considering Tomcat and SWT, our proposed model outperforms MD-CNN in terms of top-1 and top-5 accuracy and is comparable with MD-CNN in terms of top-10 accuracy.

In terms of MAP and MRR, ImbalancedBugLoc achieves better performance on 5 projects excepting SWT when comparing to all state-of-the-art models. Regarding the SWT

TABLE II
PERFORMANCE COMPARISON WITH SIX STATE-OF-THE-ART MODELS

Project	Model	Top-k Accuracy (%)							MRR	MAP
		1	2	3	4	5	10	15		
AspectJ	ImbalancedBugLoc	**52.5**	**68.7**	**77.2**	**81.0**	**83.8**	**89.0**	**91.5**	**0.66**	**0.50**
	LR Model	20.2	32.1	38.5	41.3	45.5	61.1	68.2	0.33	0.25
	BugLocator (re-implement)	23.5	30.0	35.1	39.3	42.6	53.9	58.8	0.33	0.22
	DNNLoc	47.8	56.7	65.3	66.3	71.2	85.0	86.2	0.52	0.32
	MD-CNN	45.3	-	-	-	59.0	75.2	-	0.46	0.41
	SLS-CNN	-	-	-	-	-	-	-	-	-
Tomcat	ImbalancedBugLoc	53.2	**65.5**	**71.0**	**75.0**	**78.3**	85.6	**88.9**	**0.64**	**0.59**
	LR Model	46.2	54.2	59.8	62.3	66.5	74.7	80.1	0.55	0.49
	BugLocator (re-implement)	38.4	50.0	55.8	59.6	63.1	72.3	78.3	0.50	0.46
	DNNLoc	**53.9**	64.2	67.3	71.8	72.9	80.4	85.9	0.60	0.52
	MD-CNN	46.7	-	-	-	77.6	**85.9**	-	0.60	0.55
	SLS-CNN	53.5	-	-	-	72.5	80.0	-	0.61	0.54
Eclipse UI	ImbalancedBugLoc	**48.1**	**62.1**	**68.8**	**73.0**	**76.7**	**84.4**	**87.8**	**0.60**	**0.54**
	LR Mode	36.5	47.0	52.0	58.0	60.1	70.7	75.3	0.47	0.40
	BugLocator(re-implement)	28.6	39.1	46.4	51.3	54.9	66.6	72.6	0.41	0.35
	DNNLoc	43.1	57.3	64.8	70.0	73.2	81.9	86.0	0.57	0.50
	MD-CNN	44.8	-	-	-	73.5	82.2	-	0.54	0.48
	SLS-CNN	43.2	-	-	-	68.2	75.0	-	0.52	0.42
SWT	ImbalancedBugLoc	40.2	**54.9**	**64.2**	**69.3**	73.4	84.8	89.1	0.55	0.50
	LR Model	28.3	39.4	47.9	52.7	58.2	70.0	76.8	0.41	0.36
	BugLocator(re-implement)	15.4	25.0	31.9	37.7	42.5	57.9	67.0	0.29	0.26
	DNNLoc	35.2	49.2	58.4	63.6	69.0	80.3	85.3	0.45	0.37
	MD-CNN	**46.4**	-	-	-	**75.2**	**86.7**	-	**0.57**	**0.53**
	SLS-CNN	36.8	-	-	-	63.7	76.2	-	0.48	0.39
JDT	ImbalancedBugLoc	**53.0**	**65.6**	**71.7**	**76.2**	**79.0**	**86.0**	**89.2**	**0.64**	**0.55**
	LR Model	30.0	40.3	48.2	51.1	55.2	68.1	72.4	0.42	0.34
	BugLocator(re-implement)	18.6	26.5	31.7	35.3	39.0	49.6	55.5	0.29	0.23
	DNNLoc	40.3	49.1	57.2	61.6	65.0	74.3	79.4	0.45	0.34
	MD-CNN	43.9	-	-	-	72.0	81.1	-	0.53	0.45
	SLS-CNN	41.1	-	-	-	64.1	73.4	-	0.50	0.41
Birt	ImbalancedBugLoc	**28.3**	**39.3**	**45.7**	**51.0**	**53.6**	**63.2**	**69.2**	**0.40**	**0.32**
	LR Model	12.4	18.1	22.5	25.1	27.9	37.3	42.4	0.20	0.15
	BugLocator(re-implement)	11.5	17.4	21.2	24.7	27.0	37.1	44.6	0.20	0.15
	DNNLoc	25.2	30.4	34.7	38.4	42.2	50.9	57.2	0.28	0.20
	MD-CNN	19.7	-	-	-	40.3	53.0	-	0.25	0.22
	SLS-CNN	-	-	-	-	-	-	-	-	-

project, the performance of our model is less than MD-CNN by 3% on both MRR and MAP. As can be seen in Figure VI-B1, the performance of our model is comparable with MD-CNN in terms of top-5 and top-10 accuracy.

2) *RQ2: How do the focal loss and bootstrapping methods affect the performance of the model?:* In this section, we aim to evaluate the effectiveness of the hybrid strategy which combines focal loss function with bootstrapping method to tackle the problem of imbalanced datasets in bug localization. We first compare the two loss functions: the Cross Entropy (CE) loss and Focal Loss (FL). Table III shows the top-k accuracy of ImbalancedBugLoc performed on six projects with the use of cross entropy loss and focal loss. By using Focal Loss function, the improvement on JDT increases from 43.2% to 53.0% (up 9.8%) on the top-1 accuracy. Tomcat project has a slightly improvement in term of top-1, from 52.4% to 53.2% (increased only 0.8%). For remaining projects, the top-1 accuracy is improved by 3.2%, 3.5%, 5.1% 3.9% for AspectJ, SWT, EclipseUI and Birt, respectively. The experimental results have shown that the focal loss function allows to improve the performance of DNN model, particularly in large scale projects.

Additionally, we evaluate the influence of the parameter α on the effectiveness of the focal loss function. The Tomcat project is selected for this experimentation. Figure 2 shows the top-k accuracy of ImbalancedBugLoc according to different values of α. When the value of α is small (close to 0), the model has a low performance in terms of top-k accuracy. Indeed, the top-1 accuracy is only 48.9% comparing to 52.4% when using cross entropy function. As the value of α increases,

TABLE III
TOP-K ACCURACY OF IMBALANCEDBUGLOC MODEL WHEN USING CE AND FL FUNCTION

Project	Loss Function	Top-k Accuracy (%)			
		1	5	10	15
AspectJ	CE	49.3	80.3	88.6	90.9
	FL	**52.5**	**83.8**	**89.0**	**91.5**
Tomcat	CE	52.4	**79.3**	**85.9**	**88.9**
	FL	**53.2**	78.3	85.6	**88.9**
Eclipse UI	CE	44.0	73.6	82.6	86.5
	FL	**49.1**	**77.4**	**84.9**	**88.0**
SWT	CE	37.0	71.9	83.8	88.4
	FL	**40.5**	**72.0**	**84.1**	**88.9**
JDT	CE	43.2	72.0	81.0	85.1
	FL	**53.0**	**79.0**	**86.0**	**89.2**
Birt	CE	24.5	50.7	61.9	67.6
	FL	**28.3**	**53.6**	**63.2**	**69.2**

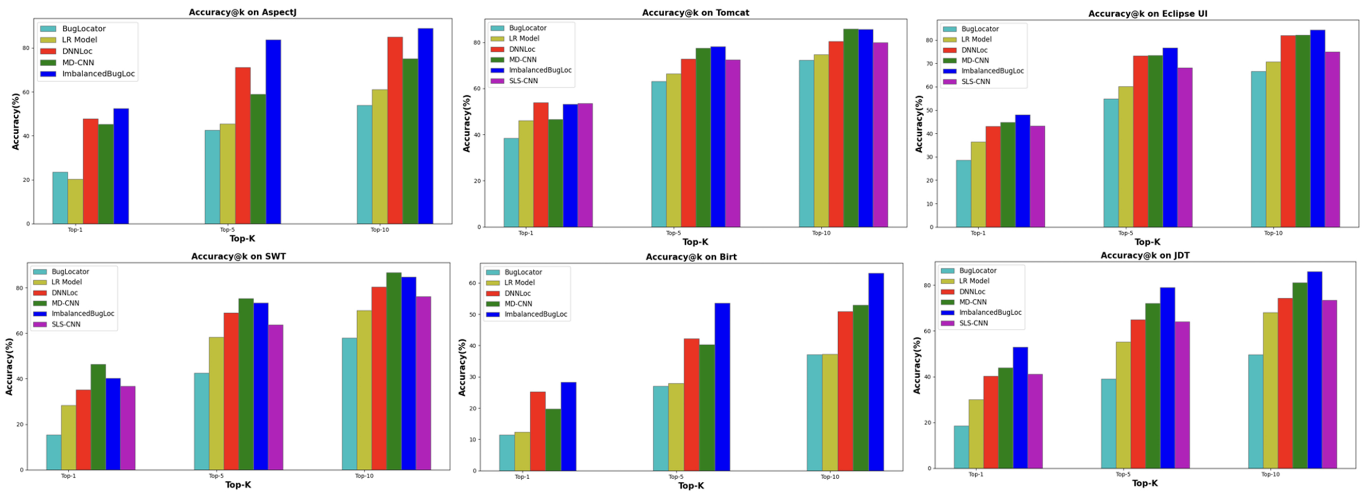

Fig. 1. Top K-Accuracy for benchmark datasets

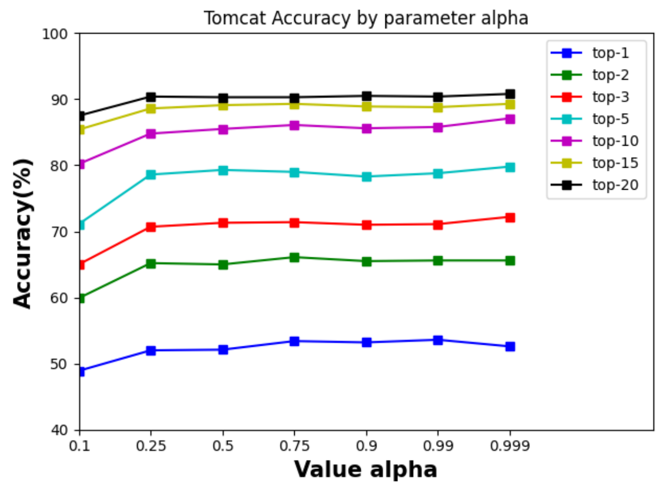

Fig. 2. Top-k Accuracy according to the value of α of the focal loss function

TABLE IV
TOP-K ACCURACY OF DNN WITH BOOTSTRAPPING

Project	Batch_size	Top-k Accuracy (%)				
		1	5	10	15	20
AspectJ	32	49.5	80.8	88.1	90.4	91.8
	Bootstrapping batch	**51.3**	**83.4**	88.6	91.3	**92.3**
	128	51.1	82.9	**88.8**	**91.5**	92.1
	256	49.0	76.7	83.5	85.9	87.2
Tomcat	32	51.5	78.0	85.4	88.5	89.8
	Bootstrapping batch	**52.7**	79.0	**85.7**	**89.1**	90.3
	128	**52.7**	**79.2**	85.6	88.6	90.3
	256	**52.7**	77.9	85.1	88.8	**90.6**
Eclipse UI	32	26.0	72.5	81.1	85.6	88.0
	Bootstrapping batch	**44.9**	73.5	82.5	86.5	88.5
	128	44.1	**73.9**	82.5	**86.6**	**88.7**
	256	44.0	73.6	**82.6**	86.5	88.6
SWT	32	27.5	68.2	80.3	86.0	89.3
	Bootstrapping batch	**38.2**	**72.5**	**83.9**	**88.9**	**91.5**
	128	37.5	72.0	83.6	88.3	91.1
	256	37.0	71.9	83.8	88.4	91.2
Birt	32	13.6	45.5	57.0	63.5	68.3
	Bootstrapping batch	**23.8**	**50.3**	**62.0**	**67.8**	**72.1**
	128	21.9	48.4	60.0	66.0	70.3
	256	21.7	48.2	59.7	65.9	70.1

the top-k accuracy of the model slightly increases. The more the value of α is close to 1, the more the model performs in terms of top k-accuracy. Indeed, when the value of $\alpha = 0.999$, the top-k accuracy of the model reaches the maximum value (top-1 = 53.6%). The reason is that when the value of α is close to 0, the model focuses more on irrelevant samples, thereby giving less accurate prediction results. We have also acknowledged that the value of α depends significantly on the skewed degree of the imbalanced dataset.

We finally assess the effectiveness of the bootstrapping method to the overall performance of ImbalancedBugLoc.

As can be seen in Table IV, the effectiveness of bootstrapping mini-batch size is compared with typical mini-batch size (i.e., 32, 128, 256). The experiment shows that the bootstrapping method results the best (or comparable) performance when comparing with using normal mini-batches (negative and positive instances are equally considered to be picked up). When the batch_size value is small, the model achieves a low prediction performance. When the batch_size value close to the bootstrapping mini-batch size, the model achieves better prediction results. It is reasonable because a larger batch size value might avoid the case in which there is no positive instance.

VII. THREATS TO VALIDITY

With regard to **internal validity**, the first threat comes from the quality of bug reports and source files. As one of the most effective features extracted from bug reports and source files represents their lexical match. Therefore, a high background knowledge about the source code of bug reporters allows to improve the performance of bug localization. The second threat to internal validity comes from the reproduction of the compared approaches. Apart from BugLocator which was re-implemented, we did not have the source code to replicate the other studies (LR Model [6], DNNLoc [9], MD-CNN [19], SLS-CNN [25]). We therefore have to rely on the published results of these works to make a comparison with our approach. Although all these approaches were applied on the same datasets, the difference in terms of configuration for training/predicting phase may still exist. The threat to **external validity** relates to the generalizability of our approach. Despite of the fact that we have conducted the experiments on six widely used benchmark datasets, our findings have not been validated on other projects which are implemented in other programming languages (such as C/C++).

VIII. Conclusion

In this paper, we have proposed an imbalanced deep neural network model to address one of the big challenge of bug localization, the imbalance of training datasets. The proposed strategy for the skewed class distribution takes advantages of two common approaches of imbalanced learning. On one hand, the proposed DNN is trained by optimizing a focal loss function which re-weights the cost of minority class while reducing the dominant affect of majority class. On the other hand, we further improve the performance of the DNN model by applying the bootstrapping method during iterative batch training. The empirical study have shown that our proposed approach produced a significant improvement of performance comparing to other related works.

References

[1] W. Zhang, Z. Li, Q. Wang, and J. Li, "Finelocator: A novel approach to method-level fine-grained bug localization by query expansion," *Information and Software Technology*, vol. 110, pp. 121–135, 2019.

[2] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 261–272.

[3] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization with code coverage representation learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 661–673.

[4] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 14–24.

[5] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 263–272.

[6] X. Ye, R. Bunescu, and C. Liu, "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*, 2014, pp. 689–699.

[7] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi, "Improving bug localization with word embedding and enhanced convolutional neural networks," *Information and Software Technology*, vol. 105, pp. 17–29, 2019.

[8] Y. Xiao, J. Keung, Q. Mi, and K. E. Bennin, "Bug localization with semantic and structural features using convolutional neural network and cascade forest," in *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, 2018, pp. 101–111.

[9] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 218–229.

[10] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi, "Machine translation-based bug localization technique for bridging lexical gap," *Information and Software Technology*, vol. 99, pp. 58–61, 2018.

[11] X. Huo, M. Li, Z.-H. Zhou *et al.*, "Learning unified features from natural and programming languages for locating buggy source code." in *IJCAI*, vol. 16, 2016, pp. 1606–1612.

[12] H. Liang, L. Sun, M. Wang, and Y. Yang, "Deep learning with customized abstract syntax tree for bug localization," *IEEE Access*, vol. 7, pp. 116 309–116 320, 2019.

[13] W. Yuan, B. Qi, H. Sun, and X. Liu, "Dependloc: A dependency-based framework for bug localization," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2020, pp. 61–70.

[14] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2980–2988.

[15] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[16] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 345–355.

[17] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[18] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[19] B. Wang, L. Xu, M. Yan, C. Liu, and L. Liu, "Multi-dimension convolutional neural network for bug localization," *IEEE Transactions on Services Computing*, 2020.

[20] S. Guan, M. Chen, H.-Y. Ha, S.-C. Chen, M.-L. Shyu, and C. Zhang, "Deep learning with mca-based instance selection and bootstrapping for imbalanced data classification," in *2015 IEEE Conference on Collaboration and Internet Computing (CIC)*. IEEE, 2015, pp. 288–295.

[21] Q. Dong, S. Gong, and X. Zhu, "Imbalanced deep learning by minority class incremental rectification," *IEEE transactions on pattern analysis and machine intelligence*, vol. 41, no. 6, pp. 1367–1381, 2018.

[22] Y. Cui, M. Jia, T.-Y. Lin, Y. Song, and S. Belongie, "Class-balanced loss based on effective number of samples," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 9268–9277.

[23] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, 2013.

[24] Ö. F. Arar and K. Ayan, "Software defect prediction using cost-sensitive neural network," *Applied Soft Computing*, vol. 33, pp. 263–277, 2015.

[25] G. Liu, Y. Lu, K. Shi, J. Chang, and X. Wei, "Convolutional neural networks-based locating relevant buggy code files for bug reports affected by data imbalance," *IEEE Access*, vol. 7, pp. 131 304–131 316, 2019.

[26] R. M. Cichy and D. Kaiser, "Deep neural networks as scientific models," *Trends in cognitive sciences*, vol. 23, no. 4, pp. 305–317, 2019.

[27] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 404–415.