2017 24th Asia-Pacific Software Engineering Conference

Improving Bug Localization with an Enhanced Convolutional Neural Network

Yan Xiao, Jacky Keung, Qing Mi, Kwabena E. Bennin
Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong
Email: yanxiao6-c, Qing.Mi, kebennin2-c@my.cityu.edu.hk, Jacky.Keung@cityu.edu.hk

***Abstract*—*Background:* Localizing buggy files automatically speeds up the process of bug fixing so as to improve the efficiency and productivity of software quality teams. There are other useful semantic information available in bug reports and source code, but are mostly underutilized by existing bug localization approaches. *Aims:* We propose DeepLocator, a novel deep learning based model to improve the performance of bug localization by making full use of semantic information. *Method:* DeepLocator is composed of an enhanced CNN (Convolutional Neural Network) proposed in this study considering bug-fixing experience, together with a new rTF-IDuF method and pre-trained word2vec technique. DeepLocator is then evaluated on over 18,500 bug reports extracted from AspectJ, Eclipse, JDT, SWT and Tomcat projects. *Results:* The experimental results show that DeepLocator achieves 9.77% to 26.65% higher F-measure than the conventional CNN and 3.8% higher MAP than a state-of-the-art method HyLoc using less computation time. *Conclusion:* DeepLocator is capable of automatically connecting bug reports to the corresponding buggy files and successfully achieves better performance based on a deep understanding of semantics in bug reports and source code.**

***Index Terms*—bug localization, convolutional neural network, word2vec, TF-IDF, deep learning, semantic information**

I. Introduction

Bug localization is a significant task during software maintenance process. To locate a newly reported bug, assigned developers need to carefully analyze the bug report and review a lot of source codes. The bug resolution activity requires a considerable large amount of time and effort. To reduce maintenance costs and substantially improve the efficiency and productivity of the whole software team, several automated bug localization approaches and tools have been proposed by researchers to localize the potential buggy files for developers. These approaches extract essential information from bug reports and source code.

The existing bug localization techniques can be categorized into four main groups. The first group comprises approaches which use traditional features related with program analysis information [1], such as the passing or failing execution information with test cases. Spectrum-based fault localization techniques [2][3] are typical examples of these approaches. These methods extract only static features from the source code or execution information, which is time-consuming. The second group comprises the IR (Information Retrieval)-based approaches that search and rank buggy files for a given bug report. These approaches measure the textual similarity between bug reports and source code files (names of classes or methods). This kind of similarity focuses more on the term weights [4]. Also, ML (Machine Leaning)-based approaches have been proposed. These approaches adopt machine learning models that are trained to match the topics of bug reports with those of source files [5] or classify the source files into multi-class using former fixed files [6].

The approaches in the above two groups focus on the term weights of natural language texts and do not consider the semantics of the source code during the bug localization process. Ye et al. [4] report that there are significant differences between natural language texts used in bug reports and code tokens in source code files. Generally, programs have well-defined syntax and there is semantics hidden deeply in the source code [7]. According to some previous researches, the semantic information is important for code suggestion [8] and code completion [9], which is also useful for bug localization. To bridge the semantic gap, deep learning has been introduced into the domain of bug localization for semantic parsing. Deep learning is known to perform excellently in the area of NLP (Natural Language Processing) and image processing [10][11]. However, the recently proposed deep learning based model for bug localization combines deep neural networks (DNNs) and IR techniques [12]. The final results are influenced by the performance of IR to some extent. Most importantly, many DNN models are used together, which makes it very complex and difficult to accurately adjust the weights.

In order to extract the underlying semantic information from the bug reports and source code, this paper proposes a CNN-based model for bug localization, namely DeepLocator. CNN is able to extract the local feature by convolving filters and has good performance in semantic parsing related NLP problem [13]. Besides semantic information, DeepLocator aims to correlate the bug reports to the corresponding buggy files instead of textual similarity used in IR-based approaches. Moreover, DeepLocator is based on an enhanced CNN proposed in this study, which enables deep learning to fully utilize and demonstrate its natural characteristic.

The main contributions of this paper are listed as follows:

- We develop DeepLocator that consists of a new rTF-IDuF method, word2vec and a proposed enhanced CNN. The enhanced CNN makes use of the important bug-fixing experience (bug-fixing recency and frequency) besides the semantic information that can be extracted by conventional CNN from bug reports and source files.
- A set of experiments is conducted to validate the fea-

978-1-5386-3681-7/17 $31.00 © 2017 IEEE
DOI 10.1109/APSEC.2017.40

sibility and effectiveness of DeepLocator. We make our dataset publicly available[1] and provide the tool used to construct the dataset.

The remainder of the paper is structured as follows. Section II presents the related works. The preliminary knowledge related to DeepLocator is reviewed in Section III. This is followed in Section IV by a discussion of rTF-IDuF and an enhanced CNN. Section V gives a detailed description of the model design, followed by the presentation of experimental results in Section VI and discussion in Section VII. We conclude this paper in Section VIII.

II. Related Work

A. Bug Localization

In the literature, a number of techniques have been proposed to help localize bugs. The commonly used techniques are either based on information retrieval (IR) methods or machine learning (ML) methods.

IR: Wong et al. [14] proposed segmentation and stack-trace analysis techniques on top of BugLocator [15] to boost fault localization. Saha et al. [16] regarded summary and description of bug reports as two different query fields, combined respectively with classes, methods, variables and comments extracted from source files. However, these eight features were considered as equally important. Moreover, their evaluation used the fixed version of the project, which will lead to the inconsistency when considering future bug-fixing information.

To bridge these gaps mentioned above, Ye et al. [4] developed an adaptive ranking model supported by a learning-to-rank technique. Six features were used in this paper: surface lexical similarity, API-enriched lexical similarity, collaborative filtering score, class name similarity, bug-fixing recency and bug-fixing frequency. In [17], bug reports and source code were represented by word embeddings and the similarity between them were added as additional features. Then the weight for each feature was trained by the learning-to-rank technique based on the previously fixed bugs. However, this model used a hill-climbing algorithm with the linear weighted sum of features, which might ignore the nonlinear relationship. These IR-based techniques rely on the textual similarity between bug reports and source files, which cannot bridge the semantic gap.

ML: Kim et al. [6] applied Naive Bayes to build a two-phase recommendation model. Phase 1 was used to decide whether the bug was predictable or not. If not, there was no further prediction behavior. If predictable, Phase 2 applied Bayes model to recommend a set of files to fix. But they mainly focused on the name of fixed files. So it was hard for the model to recommend a file that had never been fixed before. Markov logic was used in [18] by combining statement coverage, static program structure information and prior bug information. But this paper did not take the contextual information (failure explanations) into account. BugSout, a machine learning model proposed by Nguyen et al. [5], was a topic-based model using an extended LDA (Latent Dirichlet Allocation) [19]. The bugs were related to their corresponding buggy files by their shared topics. But a tuning process was needed to find the right number of topics for different projects, which made the model not automated.

B. Deep Learning in Software Engineering

In recent years, many deep learning techniques have been introduced to software engineering research, such as code suggestion, API suggestion, defect prediction, effort estimation, program classification, bug localization and so on.

White et al. [7] proposed a deep architecture to model software language specific for sequential data and suggested many avenues for future work, especially for code suggestion. Wang et al. [20] used deep belief network (DBN) to detect features from tokens extracted from Abstract Syntax Tree (AST) of source code to find defective files. But the way they mapped the tokens to vectors was not reasonable.

Choetkiertikul et al. [21] proposed a deep learning based prediction system for estimating story points based on Long Short-Term Memory (to learn a vector representation for issue reports) and Recurrent Highway Network (to build a deep representation). DeepAPI was proposed by Gu et al. [22] to find API usage sequences given an API-related natural language query, which adopted the attention-based Recurrent Neural Network (RNN) Encoder-Decoder model with considering the API importance by IDF-based weighting to train API usage sequences and their corresponding annotations. But the training time of this model was very long, which is also the drawback of RNN.

In the area of software engineering, many files are written in natural languages, such as bug reports, API descriptions and annotations. Even codes themselves are also similar with natural language to some extent. That's why many mature techniques used in Natural Language Processing (NLP) can also be applied to some issues in software engineering to extract semantic information.

C. Deep Learning Related Bug Localization

Previous studies have proposed a number of features to represent the program source code. However, few methods try to extract the semantic information from source code. Deep learning can help to bridge this gap. Deep learning is a relatively new concept and there is little prior work about deep learning based bug localization. Huo et al. [23] proposed a convolutional neural network based model to extract program structure information and learn unified features from natural and programming languages leveraging both lexicon and program structure. Lam et al. [12] built a model named HyLoc that combined deep learning with information retrieval (IR) technique. There were about six deep neural networks (DNNs) used in this model, two for feature extraction, two for projection, one for relevancy estimation and one for feature combination. However, so many kinds of DNNs posed a risk of complex adjustment of weights and high cost for training and learning. Their experiments showed that only using DNN

[1]https://github.com/yanxiao6/BugLocalization-dataset

achieved poor performance and most improvements still benefited from IR techniques, which implied that DNN in their model was a subsidiary. Different from their work, we regard the problem of bug localization as a classification problem instead of IR problem.

III. Preliminaries

A. Term Frequency-Inverse Document Frequency (TF-IDF)

The TF-IDF technique is widely used in the area of text mining and information retrieval (IR). TF represents the frequency of a term appearing in a document and IDF denotes an inverse of the number of documents where the term appears in the entire corpus. TF-IDF can filter some common terms in the documents of the repositories. The formulas of TF-IDF are as follows [4]:

$$tf_{t,d} = f_{t,d} \quad idf_{t,d} = log(\frac{N}{df_t})$$
$$w_{tf-idf} = tf_{t,d} \times idf_{t,d} \tag{1}$$

where $f_{t,d}$ is the frequency of the term t appears in the document d, N refers to the number of total documents in the document collections and df_t reflects the number of documents in the corpus containing the term t. However, this kind of general corpus is not always available in the areas of research or practice. And they represent general behaviors so that ignoring some important personal information.

In order to address the aforementioned problems, Beel et al. [24] presented TF-IDuF (term frequency-user focused inverse document frequency) using personal document repositories instead of the general corpus with a simple proof. TF-IDuF is defined as [24]:

$$tf_{t,d} = f_{t,d} \quad idf_{t,d} = log(\frac{N_u}{n_{t,u}})$$
$$w_{tf-idf} = tf_{t,d} \times idf_{t,d} \tag{2}$$

where N_u refers to the number of total documents in the user collections and $n_{t,u}$ reflects the number of documents in the collections that incorporate the term t.

But Equation (1) and (2) are general equations for term weights. If there are large differences in frequencies, the term weights computed by (1) and (2) don't achieve good performance in practice [25].

B. Convolutional Neural Network (CNN)

CNN, inspired by biological processes and multilayer perceptron, is the first learning algorithm based on neocognitron [26] and receptive field [27]. CNN is a promotion of neocognitron, and neocognitron is a special case of CNN. CNN contains several convolutional layers that are often combined with pooling steps and then followed by a fully-connected layer similar with a standard neural network with multi-layers [28].

The process of convolution and pooling (subsampling) [29] is shown in Figure 1. In the process of convolution C, the input is convoluted with a trainable filter W and then added a offset b. Later, a non-linear function is used to get a reduced feature map c_i. Next is the subsampling procedure S. Mean-over-time polling or max-over-time pooling operation [10] is frequently used in this layer, which takes the mean ($c = mean(c_i)$) or maximum value ($c = max(c_i)$) as the feature corresponding to this particular filter. The convolutional layer is used to detect features. Then in pooling layer, original features are subsampled to reduce the number of parameters, which can also alleviate the over-fitting problem [29].

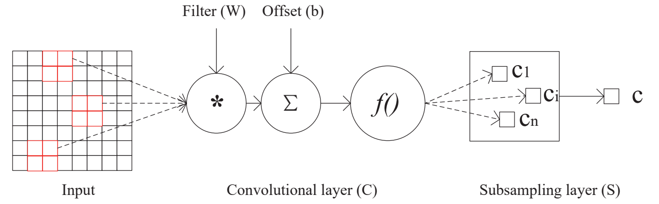

Fig. 1. The Process of Convolution and Subsampling of CNN.

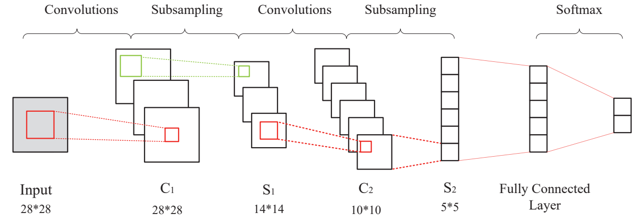

Fig. 2. The Net Structure of CNN.

The basic structure of CNN [11] is shown in Figure 2. C layer and S layer are based on the aforementioned convolutional and pooling procedure respectively. The C layer is used to capture features. The input of every neuron in C layer is linked with the local receptive field of former layer and then extract the local feature by convolutional operations. As soon as the local feature is extracted, its positional relationship with other features will also be known. The S layer is a feature mapping layer. Each computational layer has several feature maps and every feature map is a 2-dimension surface. The weight of each neuron in this surface is the same. The final *softmax* function operates the classification.

IV. Enhanced model for bug localization

As mentioned in III-A, the performance of general TF-IDuF (term frequency-user focused inverse document frequency) is not satisfactory in practice. In addition, even though the conventional convolutional neural network (CNN) has good performance in semantic parsing related Natural Language Processing (NLP) problems [13], it cannot learn the bug-fixing experience by itself. Therefore, we will enhance TF-IDuF and conventional CNN for the task of bug localization in this section.

A. revised TF-IDuF (rTF-IDuF)

It is known that the logarithm variant of TF could improve the performance of TF-IDF in practice given in Croft's experiments [25]. Specifically it can dampen effects of large differences in frequencies:

$$tf_{t,d} = log(f_{t,d}) + 1 \tag{3}$$

Inspired by Croft's work [25], we propose a rTF-IDuF to further improve the model performance:

$$w_{tf-idf} = (log(f_{t,d}) + 1) \times log(\frac{N_u}{n_{t,u}}) \tag{4}$$

where $f_{t,d}$ is the frequency of the term t appears in the document d, N_u refers to the number of total documents and $n_{t,u}$ reflects the number of documents in user collections that contain the term t.

In reality, the complete corpus is not always available. Also, the bug reports for different projects are usually different in terms of writing style and contents. Therefore, it is convenient and reasonable to apply rTF-IDuF that focuses on personal collections.

B. Enhanced CNN

This section proposes a novel technique to enhance CNN for bug localization. Besides the semantic information, conventional CNN model contains no other important information related to bug localization, especially the fixing history of some source files. Furthermore, the bug-fixing recency and frequency are verified to be useful for bug localization by [12] and [4].

The change history of source files for bugs contains useful information to find fault-prone files [30]. This implies specifically that those source files fixed recently are more likely to be buggy files than those fixed long time ago (bug-fixing recency) and those source files that have been fixed many times prone to still contain bugs than those seldom fixed or even never fixed (bug-fixing frequency).

In order to improve the performance of bug localization, we enhance the conventional CNN by adding bug-fixing recency and frequency in the fully connected layer as two penalty terms to the cost function.

The conventional CNN is trained to minimize the following mean cross-entropy error function [22]:

$$\begin{aligned} L(\theta) &= \frac{1}{N}\sum_{i=1}^{N} cost_i \\ cost_i &= -\sum_{j=1}^{T} t_{ij} log(y_{ij}) \end{aligned} \tag{5}$$

where N is the size of a batch of multiple samples and T is the number of classes. $cost_i$ is the cost function of sample i. t_{ij} is the true value of class j of sample i and y_{ij} is the output probability of class j of sample i.

Then we propose a new cost function by considering bug-fixing recency (r_i) and bug-fixing frequency (f_i), which is defined in the following:

$$cost_i = -\sum_{j=1}^{T} t_{ij} log(y_{ij}) - \omega_1 r_i - \omega_2 f_i \tag{6}$$

where ω_1 and ω_2 are initialized by random values from a truncated normal distribution and tuned in the training phase similar with other weights. r_i and f_i are the scaled values of the bug-fixing recency and frequency respectively.

We use similar equations as given in Ye's work [4] to implement the bug-fixing recency and frequency of a source file. For a sample i with source file s and a new bug report r, the bug-fixing recency is defined as:

$$R_i = \frac{1}{r.month - s.month + 1} \tag{7}$$

where $r.month$ represents the month in which bug report r was generated and $s.month$ denotes the month when the source file s was fixed lastly before bug report r was created. $r.month$ and $s.month$ take the year into account. For example, if $r.month$ is Jan. this year and $s.month$ is Dec. last year, R_i is 0.5. Therefore, the bug-fixing recency R_i is the inverse of time interval between the time of creating the bug report and the last fixing time of the source file.

The bug-fixing frequency F_i is expressed by the number of fixing times of the source file s before the bug report r is submitted.

R_i and F_i may have wide value ranges that makes them incomparable with each other, which may result in detrimental effects. R_i and F_i both in training and testing set are scaled as follows:

$$\begin{aligned} r_i &= \begin{cases} 0 & if\, R_i < R_{min} \\ \frac{R_i - R_{min}}{R_{max} - R_{min}} & if\, R_{min} \le R_i \le R_{max} \\ 1 & if\, R_i > R_{max} \end{cases} \\ f_i &= \begin{cases} 0 & if\, F_i < F_{min} \\ \frac{F_i - F_{min}}{F_{max} - F_{min}} & if\, F_{min} \le F_i \le F_{max} \\ 1 & if\, F_i > F_{max} \end{cases} \end{aligned} \tag{8}$$

where R_{min} (F_{min}) and R_{max} (F_{max}) are the minimum and maximum values of R_i (F_i) in the training set. When testing, R_i (F_i) may be larger than R_{max} (F_{max}) or less than R_{min} (F_{min}) that are obtained in the training phase. So the first and last judgments are necessary.

The scaled bug-fixing recency (r_i) and frequency (f_i) are then used in Equation (6) to obtain the enhanced CNN.

V. DeepLocator

In this section, we describe DeepLocator, a deep learning based model that localizes the buggy files for bug reports automatically. DeepLocator adapts an enhanced CNN proposed in IV-B for the task of bug localization. Figure 3 provides the overall workflow of DeepLocator. In the training phase, bug reports and source files are firstly pre-treated by rTF-IDuF as discussed in IV-A and Abstract Syntax Tree (AST) detection

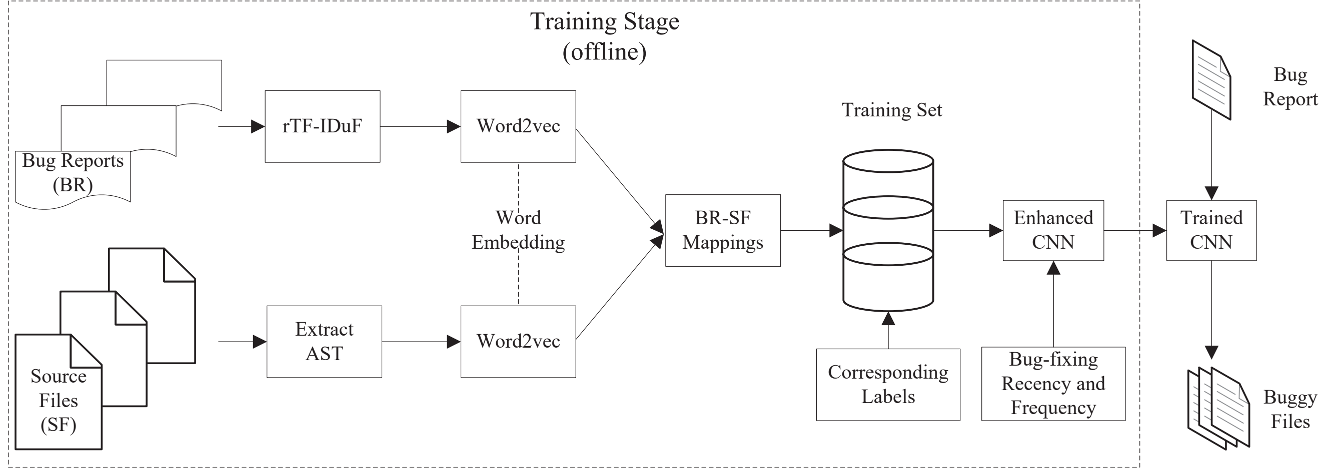

Fig. 3. The Overall Workflow of DeepLocator.

respectively. Word embedding is then applied to transform the words into vectors. The mappings between bug reports and source files, as well as corresponding labels (buggy or not buggy), are obtained to train the enhanced CNN. The training stage can be done offline. When we receive new bug reports, the already-trained enhanced CNN will efficiently utilize the bug reports to obtain the location of buggy files. The details of DeepLocator are discussed in the following sections.

A. Data Preprocessing

Typically, there are summaries and descriptions for each bug in the bug reports. Because most of the summaries are very short or sometimes even vacant, we first combine the summary and description of each bug into one document. Next, we apply proposed rTF-IDuF to filter some common words (such as I, the, to) to shorten the length of each sentence, which is a standard data preprocessing to eliminate redundant words [4][15][6].

The well-defined programming syntax can be represented by AST [9] that has been successfully used to extract the programming patterns of source code [31]. The two main AST nodes we extract from the source code are [20]: a) nodes of method invocations and class instance creations, e.g., method names. b) declaration nodes, e.g., method declarations.

All words are further preprocessed because developers most often combine words to create a new "word" when defining the classes or methods in the source code, which is also possible to be mentioned in bug reports. According to CamelCase Naming Convention [32], combined words can be split into separated real words based on capital letters. For example, "WorkbenchActionBuilder" is split into "Workbench", "Action" and "Builder". Besides, we pay attention to some special capital words, such as URI, IDE and so on, which should not be separated. Lastly, we change all the capital letters into lower cases to meet the rules of word embedding.

B. Word Embedding-word2vec

Words themselves cannot be the input of CNN directly. Thus the pre-treated words should be embedded into vectors. We first find the maximum sentence lengths from bug reports and preprocessed AST nodes respectively. Then they are padded by <PAD> tokens to be separately same with their maximum sentence lengths. In this paper, word2vec with Skip-gram model [33] is applied to transform the preprocessed bug reports and AST nodes into vectors. There may be some words absent in the set of pre-trained words of word2vec. These words are initialized randomly and fine-tuned during training [13]. Ye et al. [17] verified that word embeddings trained on Wiki corpus and project (Eclipse and Java) specific corpus had similar performance. But Wiki corpus has a larger size of vocabulary and words (96 times and 548 times the size of project specific corpus respectively). So the pre-trained word2vec used in our model is based on Wiki corpus.

C. Enhanced CNN Training and Prediction

Now we begin building the enhanced CNN. Firstly, we construct the training dataset. Besides the real corresponding buggy files, we only select the files that were ever buggy files as the negative samples due to the imbalanced dataset. We consider them as potential buggy files.

As we discussed in Section IV-B, an enhanced CNN is adopted for the main training part, which contains one convolutional layer followed by a max pooling layer. We also add a dropout layer that is frequently used to regularize CNN. The dropout layer only enables a fraction of the neurons randomly, which prevents neurons from co-adapting and at the same time forces neurons to learn useful features individually. The following Figure 4 is the structure of CNN used in this paper. The sentence contains pre-treated bug reports and AST nodes of source code.

This model is trained by gradient descent algorithm together with Adam (Adaptive Moment Estimation) optimizer, which uses an adaptive learning algorithm. Compared to Adadelta

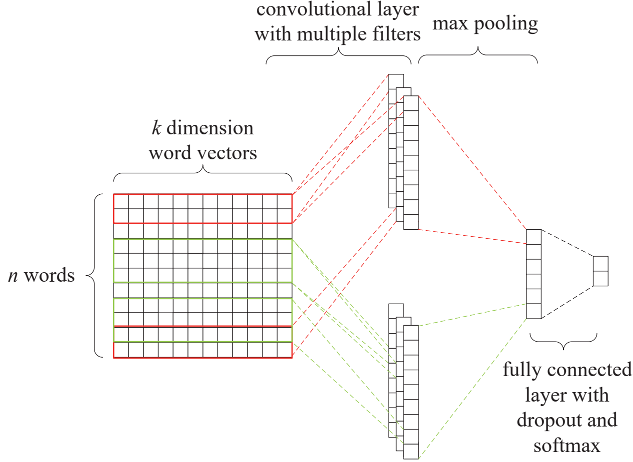

Fig. 4. The Architecture of CNN for a Sentence. n is the number of words in the sentence and k is the vector dimension of each word that is tuned experimentally (details are shown in Figure 5).

[34] that keeps an exponentially decaying average of past squared gradients, Adam can also store the exponentially decaying average of past gradients. What's more, Adam has shown better performance than other algorithms that automatically adjust learning rates [35].

After the trained CNN is available, the bug reports have been related to buggy files instead of matching them by textual similarity. When a new bug report comes, it is paired with each source file and then input into the trained CNN to classify buggy files. The probability of being the buggy file is used to give priority to more buggy-prone files.

VI. Experiments

Several experiments are conducted to evaluate the performance of DeepLocator. We describe the datasets used, the experimental setup and performance metrics in VI-A. The research questions are listed in VI-B, followed by the results in VI-C.

A. Experimental Settings and Evaluation Metrics

As some source files may be modified or even deleted, for an old bug, we may fail to find the original mapped source files without before-fix versions. So we collect dataset (Table I) with before-fix versions and relate them to each bug based on the publicly available mappings of bug reports and the corresponding commit history provided by [4]. More statistical descriptives about the dataset are left out due to space limitation. We use tensorflow to construct the model and all experiments are run on a server with CPU Intel Xeon CPU E5-4620 2.20GHz (32 cores), 128 GB RAM.

The chronologically sorted bug reports of each project are split into 70% as the training set (older bugs) and 30% as the testing set (newer bugs). This splitting strategy is adopted by many previous studies [6][36] and we further validate the effect of different training sizes on performance in RQ4.

TABLE I
Subject Projects.

Project	Time Range	# of Bug Reports	# of Fixed Buggy Files	avg. # of Buggy Files per Bug
AspectJ	03/2002-01/2014	593	1,151	4.0
Eclipse UI	10/2001-01/2014	6,495	6,228	2.7
JDT	10/2001-01/2014	6,274	5,002	2.6
SWT	02/2002-01/2014	4,151	1,415	2.1
Tomcat	07/2002-01/2014	1,056	1,038	2.4

In order to evaluate the performance of DeepLocator, three metrics are adopted: Precision rate (P), Recall rate (R) and F-measure (F), which are widely used in classification tasks [6]. Their definitions are expressed as follows.

$$P = \frac{true\,positive}{true\,positive + false\,positive} \tag{9}$$

$$R = \frac{true\,positive}{true\,positive + false\,negative} \tag{10}$$

$$F = \frac{2 * Precision * Recall}{Precision + Recall} \tag{11}$$

where true positive represents the number of predicted buggy files that are truly buggy files, while true negative denotes the number of predicted non-buggy files that are truly buggy-free files. False positive represents the number of predicted buggy files that are actually non-buggy files, while false negative denotes the number of predicted buggy-free files that are actually buggy files. A good classification model desires to obtain higher P and R. As a comprehensive measurement, F is a trade-off between P and R.

The above three metrics are used to evaluate the effectiveness of DeepLocator. We also use another metric, Mean Average Precision (MAP) that is widely used in existing bug localization approaches [12][4][17], to compare DeepLocator with the state-of-the-art IR-based techniques.

B. Research Questions

Specifically, we aim to answer the following questions:

RQ1: What is the effect of pre-trained word2vec on DeepLocator? Iyyer et al. [37] reported that word vectors initialized by unsupervised neural language models had better performance than those randomly initialized in Natural Language Processing (NLP). In order to validate the effectiveness of pre-trained word2vec in our model, we compare the accuracy of word vectors obtained by pre-trained word2vec and random initialization respectively. We also analyze the influence of the dimension of word embedding on accuracy.

RQ2: What is the effect of model settings on DeepLocator? Before building DeepLocator, we need to analyze two main settings: the size of filters and the number of convolutional layers.

RQ3: What is the performance of DeepLocator? Our evaluation can be divided into four steps. First, we compare

the conventional CNN with two classification models (i.e., SVM classifier [38] and Naive Bayes classifier [6]). Then, we evaluate whether the enhanced CNN improves the accuracy of DeepLocator against the conventional CNN. The only difference between the two CNNs is that the enhanced CNN considers bug-fixing recency and frequency but the conventional CNN does not. Further, we aim to examine the training time and prediction time overhead of DeepLocator. Lastly, since Lam et al. [12] has validated that HyLoc has comparable or even better performance than existing approaches [4][5][17][15], we use HyLoc as our competitor and analyze the results. The same input and output are used for the models.

RQ4: What is the effect of larger dataset on accuracy of DeepLocator? Some related works of bug localization, such as [12] and [4], split the datasets into 10 folders. Ye et al. [4] claimed that the increase of training datasets had little effect on the accuracy. But most deep learning models require more data to obtain the features hidden deeply in the datasets. In order to evaluate the impact of different sizes of training dataset on DeepLocator, the chronologically sorted bug reports of the large projects (Eclipse UI and JDT) are split into 10 folders. $fold_1$ consists of oldest bug reports and $fold_{10}$ contains newest bug reports. We keep $fold_{10}$ as the unchanged testing dataset. The 9 training datasets are used incrementally. To begin, only one folder ($fold_9$) is used as training dataset and then two folders ($fold_9 \cup fold_8$), and so forth, until all 9 folders ($fold_9 \cup fold_8 \cup ... \cup fold_2 \cup fold_1$) are used as training datasets.

C. Results and Analyses

RQ1: Impacts of Pre-trained Word2vec on Accuracy.
We only show the results of Project AspectJ. Note we also experiment with other projects; the results are qualitatively similar and omitted due to space limitation. As shown in Figure 5, the accuracy by using pre-trained word2vec is always higher than using random initialization, which indicates that word vectors initialized by the unsupervised neural language model (word2vec) perform better than those initialized randomly in bug localization.

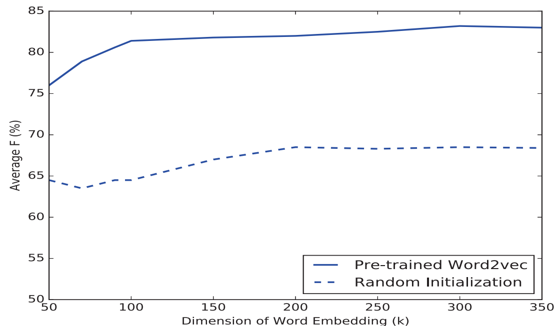

Fig. 5. Performance of Word2vec and Random Initialization along the Dimension of Word Embedding (k).

Moreover, according to Figure 5, the word embedding dimension of woed2vec has little effect on the accuracy, while the increasing dimension of word vectors randomly initialized can improve the accuracy at the beginning. Compared to word2vec, word vectors randomly initialized contain fewer characteristics. So more vectors are needed to represent and distinguish words. But if more dimensions of word vectors are used, the dimension of input will increase dramatically, which augments the training and predicting time. Therefore, the word vector dimension in our experiments is set as 100.

RQ2: Accuracy Under Different Model Settings.
We first evaluate how to choose the size of filters. According to [39], when the size of n-gram is from 4 to 8, there are more possibilities to improve the performance of expression. To make the figure distinct, the accuracy of three projects (AspectJ, SWT and Tomcat) along different filter sizes are showed in Figure 6. Although the accuracies of these three projects fluctuate when the filter size is between 4 and 8, they are still higher than the accuracy when the filter size is from 1 to 3 and from 9 to 12. If the filter size is very small, e.g. 1-3, the model fails to extract the semantic features because each sentence is split into one or two words that have less semantic information. On the other hand, if larger filter size is used, in other words, more words are in a group, it is harder for the model to extract semantic features from a large group. So in our experiments, the filter size is chosen from 4 to 6 whose accuracies are relatively high.

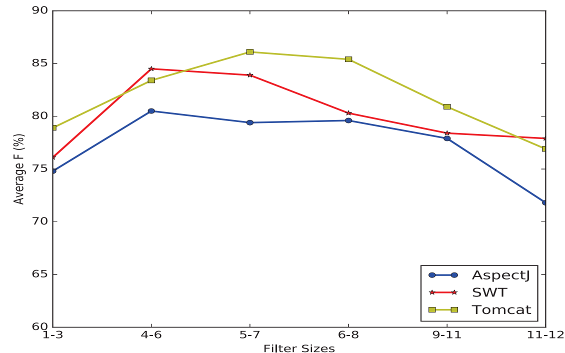

Fig. 6. Performance of Different Filter Sizes.

TABLE II
ACCURACY AND TIME CONSUMPTION ON DIFFERENT NUMBER OF LAYERS.

Project	One-layer		Two-layer	
	F (%)	Time (s)	F (%)	Time (s)
AspectJ	79.92	8.69	81.16	692.26
Eclipse UI	84.13	8.83	85.58	689.83
JDT	82.28	8.78	83.21	702.18
SWT	83.61	7.93	84.71	682.61
Tomcat	85.04	7.69	85.73	696.67

Next, we compare the average F and the average time consumption for each batch of CNN by using one convolutional layer and two convolutional layers. When using a two-layer CNN, the features between words are convolved again. So a

TABLE III
ACCURACY OF TWO MODELS.

Project	Model	P (%)	R (%)	F (%)	F_max (%)	F_min (%)
AspectJ	DeepLocator	77.37	80.23	79.92	81.21	77.94
	CNN	57.83	53.49	55.57	60.42	52.79
Eclipse UI	DeepLocator	82.37	84.58	84.13	85.96	79.19
	CNN	55.35	61.62	59.29	67.17	49.22
JDT	DeepLocator	83.43	79.92	82.28	85.41	78.48
	CNN	58.79	61.46	60.85	65.49	54.23
SWT	DeepLocator	82.33	84.98	83.61	84.92	79.95
	CNN	73.37	75.49	73.84	76.39	69.98
Tomcat	DeepLocator	83.35	87.21	85.04	86.29	83.13
	CNN	58.01	58.62	58.39	63.59	52.81

deeper semantic information is extracted. In Table II, it can be observed that the performance of the two-layer CNN is a bit better than the one-layer CNN, yet the cost time is much higher. So DeepLocator uses a one-layer CNN.

RQ3: Performance of DeepLocator.

Firstly, we compare conventional CNN with other techniques. Due to space limitation, we report the highest F of the two classifiers (SVM and Naive Bayes) that are 46.71% and 53.57% respectively, which is lower than the average F of CNN (66.61%). Then table III reports the average values of Precision, Recall, and F-measure for each bug paired with all source files, maximum F and minimum F of DeepLocator and CNN. We find that the P, R and F of DeepLocator are all higher than CNN. In comparison with CNN, DeepLocator achieves from 9.77% to 26.65% higher average F. Because CNN does not have the long-term memory, it cannot learn the features about bug-fixing recency and frequency, which are included in DeepLocator. Furthermore, the relative interval between F_max and F_min of CNN is larger than DeepLocator, which shows the standard deviation of CNN's accuracy is larger than DeepLocator. Based on the above results, DeepLocator is more robust and performs better.

TABLE IV
PREDICTION TIME (SECONDS) PER BUG REPORT OF THREE MODELS.

Prediction Time	AspectJ	Eclipse UI	JDT	SWT	Tomcat
DeepLocator	25.7	29.9	45.1	9.1	7.9
CNN	26.4	28.3	43.6	9.2	7.7
HyLoc	144.0	126.0	198.0	108.0	60.0

Then we report the time overhead of DeepLocator. Because more weights in DeepLocator need to be adjusted in training phase than CNN, the average time consumption of DeepLocator per batch in training is a bit higher (about 0.03 seconds) than CNN. Due to space limitation, Table IV presents the average prediction time for one bug report of DeepLocator, conventional CNN and the existing deep learning related model (HyLoc) [12]. It is observed that the average prediction time of DeepLocator and conventional CNN are very similar. Especially the prediction time for one bug report of DeepLocator is much lower than HyLoc, which indicates that DeepLocator is more suitable in practice.

Finally, we analyze the comparison between DeepLocator and HyLoc. In Table V, the average MAP of DeepLocator is 3.8% higher than the average MAP of HyLoc. If we analyze the results, we observe that some buggy files given low scores by HyLoc can be predicted by DeepLocator. For example, HyLoc gives one buggy file of Bug 54450 in Project Tomcat Rank 50, which is a relatively low score. However, DeepLocator predicts it as a buggy file because "resource" in bug reports is paired with "context" related words (getContext, configureContext, processContextConfig) in source files many times. Another example is Bug 399401 in Project Eclipse UI. The rank of its buggy file is 38. However, "perspective" is paired with "view" and "visible" related words (getViewLayout, isPartVisible), so DeepLocator can relate them by training on older bug reports if these pairs have ever appeared. Therefore, even tokens in source files are mismatched with words used in bug reports, DeepLocator can also relate them, which indicates that DeepLocator can learn correlations between them if they appear frequently as pairs.

TABLE V
MAP OF DEEPLOCATOR AND HYLOC ON FIVE PROJECTS.

MAP	AspectJ	Eclipse UI	JDT	SWT	Tomcat
DeepLocator	0.34	0.42	0.45	0.40	0.54
HyLoc	0.32	0.41	0.34	0.37	0.52

RQ4: Impact of the Size of Training Dataset on Accuracy.

Figure 7 shows the average F of both projects (Project Eclipse UI and JDT) by using different training sets. We observe that initially adding more training data improves the F values. But when more data is involved, especially when more older bug reports are used, the accuracy decreases. This is because older bugs introduce more interference to the model, which also confirms the necessity of using bug-fixing recency as important features.

VII. DISCUSSION

A. *Effectiveness of rTF-IDuF*

In this section, we analyze the difference between TF-IDF and rTF-IDuF. Some common words may have special meanings in different collections. And bug reports for different projects are usually different in terms of writing style and content. It is therefore not appropriate to consider them in the same perspective. For example, "aspect" exists in both Project AspectJ and Eclipse UI. There are many "aspect" in AspectJ

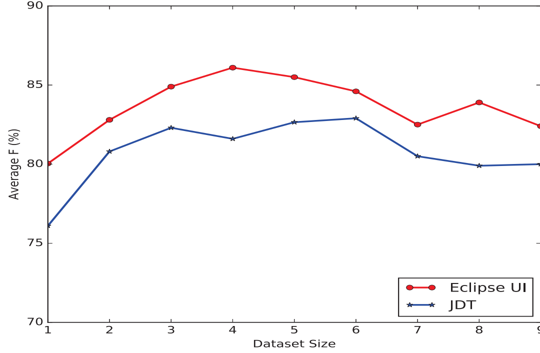

Fig. 7. Performance of Different Training Set Sizes.

and most of them represent the name of this project. But there are few "aspect" in Eclipse UI and they denote important information just as shown on the right of Figure 8. In other words, "aspect" in Bug 201616 of Project Eclipse UI should be more important than in Bug 415266 of Project AspectJ, which implies the term weight of "aspect" in Bug 201616 should be larger than in Bug 415266.

AspectJ **Bug ID:** 415266 **Summary:** Bug 415266 LTW not working when JMX is enabled. **Description:** When I enable JMX remote management on a JVM along with AspectJ load-time weaving (LTW), our Aspect doesn't appear to get woven in. This are the JVM arguments ...	**Eclipse UI** **Bug ID:** 201616 **Summary:** Bug 201616 [Min.Max] Need to be able to change the orientation of a minimized stack. **Description:** We currently use the aspect ratio of the existing view stack to set the orientation to be used when showing the view as fast views. While this works in most cases there are enough ...

Fig. 8. Bug Reports that Contain "aspect" in Project AspectJ and Project Eclipse UI.

We calculate the term weights of "aspect" using TF-IDF and rTF-IDuF. TF-IDF uses the whole corpus of available bug reports, while rTF-IDuF just considers the bug reports from the same project. The term weights of "aspect" computed by TF-IDF is 7.36 and 3.09 respectively for Bug 415266 in AspectJ and Bug 201616 in Eclipse UI, while rTF-IDuF is 1.48 and 6.99 respectively. They are different, which validates that the word "aspect" has different importance in different projects. When considering the whole corpus using TF-IDF, the term weight of "aspect" in Bug 201616 (3.09) is less than in Bug 415266 (7.36), which is not consistent with above observations. Actually, there are many words like "aspect" that have different meanings in different collections. Therefore, rTF-IDuF which considers the "user" collection and not the entire corpus is appropriate, practically feasible and effective.

B. *Why does DeepLocator work?*

The major challenge of bug localization is the semantic gap between bug reports and source code files. Textual similarity used in most existing techniques [4][12][15][6] is based on the term frequency rather than semantic information of words and phrases. Unlike textual similarity, DeepLocator correlates bug reports to the corresponding buggy files based on a deep understanding of semantics.

Abstract Syntax Tree (AST) is applied to represent syntax and extract programming patterns of source code, which makes the best of the information in source code. Then DeepLocator uses word embedding to map words into semantic vector space where similar words are close to each other. So embedded word vectors containing the information of semantic similarities benefit DeepLocator a lot. What's more, unlike the studies in [4][17] using a linear weighted sum of features, DeepLocator uses an enhanced CNN proposed in this study to correlate bug reports to buggy files including both linear and nonlinear relationships. Besides, CNN model performs well in the semantic parsing field because of the convolving filters [13], which helps DeepLocator to extract hidden semantic information between bug reports and source files.

C. *Threats to Validity*

The experimental results demonstrate the feasibility of DeepLocator, however, we do acknowledge there are still some potential threats to validity of our approach and experiments. Firstly, the proposed approach could be affected by stemming and removal of stop words process, which will be investigated in future. Secondly, the analysis on the choice of filter size (RQ1) is related to the writing style of developers. For example, if developers in a project team prefer to use very long phrases to express bug reports, the filter size should also be longer. We analyze the results of the dataset and provide a general conclusion. This conclusion may be inadequate. Thirdly, we implement HyLoc according to the algorithm provided by [12] and obtain similar results. But the results are not perfectly the same. Therefore, we choose the best ranked results to compare with our results during the results analysis. Finally, all dataset used are obtained from Java projects. The results may not generalize to other projects written in other languages. In the future, we intend to improve the model and use this model in other projects written in different languages.

VIII. Conclusions and Future Work

DeepLocator, a deep learning based model that consisted of an enhanced CNN proposed in this study, together with a new rTF-IDuF method and the pre-trained word2vec technique, is proposed to improve the performance of bug localization. The experimental results show that DeepLocator performs better (36.29% and 29.43% improvements) than existing classification models and outperforms HyLoc with less computation time. DeepLocator bridges the semantic gap by using Abstract Syntax Tree to parse the syntax of source code, word embedding to obtain semantic similarities and the enhanced CNN to learn the correlations between bug reports and source code.

We intend to enhance DeepLocator to be more sensitive to buggy file orders and comparable with more bug localization

techniques [23]. In future, we will also investigate the performance of DeepLocator using other TF-IDF weight schemes [4][40].

IX. Acknowledgement

This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (No. 11208017 and 11214116), and the research funds of City University of Hong Kong (No. 7004683).

References

[1] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2005, pp. 273–282.

[2] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 831–848, 2006.

[3] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[4] 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*. ACM, 2014, pp. 689–699.

[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 *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 2011, pp. 263–272.

[6] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE Transactions on Software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.

[7] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*. IEEE, 2015, pp. 334–345.

[8] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 858–868.

[9] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 837–847.

[10] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *Journal of Machine Learning Research*, vol. 12, no. Aug, pp. 2493–2537, 2011.

[11] B. Kwolek, "Face detection using convolutional neural networks and gabor filters," *Artificial Neural Networks: Biological Inspirations–ICANN 2005*, pp. 551–556, 2005.

[12] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (n)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 476–481.

[13] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[14] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 181–190.

[15] 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 *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 14–24.

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

[17] 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*. ACM, 2016, pp. 404–415.

[18] S. Zhang and C. Zhang, "Software bug localization with markov logic," in *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 424–427.

[19] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[20] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 297–308.

[21] M. Choetkiertikul, H. K. Dam, T. Tran, T. Pham, A. Ghose, and T. Menzies, "A deep learning model for estimating story points," *arXiv preprint arXiv:1609.00489*, 2016.

[22] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 631–642.

[23] X. Huo, M. Li, and Z.-H. Zhou, "Learning unified features from natural and programming languages for locating buggy source code." in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2016, pp. 1606–1612.

[24] J. Beel, S. Langer, and B. Gipp, "TF-IDuF: a novel term-weighting sheme for user modeling based on users' personal document collections," in *Proceedings of the iConference 2017*, Wuhan, China, Mar. 22-25 2017. [Online]. Available: http://ischools.org/the-iconference/

[25] W. B. Croft, D. Metzler, and T. Strohman, *Search engines: Information retrieval in practice*. Addison-Wesley Reading, 2010, vol. 283.

[26] K. Fukushima and S. Miyake, "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition," in *Competition and Cooperation in Neural Nets*. Springer, 1982, pp. 267–285.

[27] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *The Journal of Physiology*, vol. 160, no. 1, pp. 106–154, 1962.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1097–1105.

[29] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the 25th International Conference on Machine Learning*. ACM, 2008, pp. 1096–1103.

[30] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 432–441.

[31] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, 2009, pp. 383–392.

[32] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "To camelcase or under_score," in *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*. IEEE, 2009, pp. 158–167.

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

[34] M. D. Zeiler, "Adadelta: An adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[35] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[36] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International Conference on Machine Learning*, 2016, pp. 2091–2100.

[37] M. Iyyer, P. Enns, J. Boyd-Graber, and P. Resnik, "Political ideology detection using recursive neural networks," in *Proceedings of the Association for Computational Linguistics*, 2014, pp. 1113–1122.

[38] S. Shivaji, J. E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve bug prediction," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 600–604.

[39] V. Kešelj, F. Peng, N. Cercone, and C. Thomas, "N-gram-based author profiles for authorship attribution," in *Proceedings of the Conference Pacific Association for Computational Linguistics, PACLING*, vol. 3, 2003, pp. 255–264.

[40] C. D. Manning, P. Raghavan, and H. Schütze, "Scoring, term weighting and the vector space model," *Introduction to Information Retrieval*, vol. 100, pp. 2–4, 2008.