Multi-Dimension Convolutional Neural Network for Bug Localization

Bei Wang, Ling Xu, Meng Yan, Chao Liu and Ling Liu, *Fellow, IEEE*

Abstract—Software bugs remain frequent in the life cycle of software development and maintenance. Automatic localization of buggy source code files is critical for timely bug fixing and improving the efficiency of software quality assurance. Various bug localization techniques have been proposed using different dimensions of features. Recent studies have shown that different dimensions of features may play different roles in bug localization. Unfortunately, how to effectively merge these dimensions of features for improving bug localization has rarely been investigated. This paper presents a Multi-Dimension Convolutional Neural Network (MD-CNN) model for bug localization automatically based on a bug report. Our approach has dual-novelty. First, we identify and extract five statistical dimensions of features. Second, we design a Convolutional Neural Network (CNN) model that takes our five statistical dimensions of features as the input and iteratively learns the complex and non-linear relationship between the features and the bug locations. The MD-CNN bug localization model is verified using six large-scale open source projects. The experimental results show that our MD-CNN outperforms the existing representative bug localization techniques in terms of the Mean Average Precision (MAP) and the number of bugs successfully localized in the top 1, 5, and 10 matched source code files.

Index Terms—Bug localization, feature extraction, convolutional neural network, software quality assurance

1 Introduction

Large-scale software systems are the complex artifacts created by team efforts. Software bugs remain frequent in the lifecycle of software development and maintenance. Bug tracking systems (e.g., Bugzilla and JIRA) are used to report and manage bugs. Developers or users can submit bug reports to bug tracking systems upon discovering abnormal behavior of a software project [20].

Bug localization refers to the task of locating the potential buggy source code files in a software project given a bug report. Developers who assign to resolve a bug report usually need to find the locations of the source code files in order to fix the bug. Generally speaking, in order to locate a reported bug, a developer needs to analyze the bug report and reviews a large number of source code files. Unfortunately, for a large-scale project, not only the number of source code files is at the scale of several thousands, the number of bug reports is often very large too. For example, until May 14th, 2019, Eclipse project has received more than 547,200 bug reports. It is prohibitively expensive and labor intensive to identify potentially buggy files for all incoming bug reports manually. Therefore, automated localization of buggy files can significantly improve the efficiency of bug fixing and speed up the productivity of software maintenance.

- *Bei Wang, Ling Xu, Meng Yan are with the School of Big Data and Software Engineering, Chongqing University, P.R. China, E-mail: {bwang2013, xuling, meng}@cqu.edu.cn*
- *Chao Liu is with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. E-mail: liuchaoo@zju.edu.cn*
- *Ling Liu is with the College of Computing, Georgia Institute of Technology, USA. E-mail: ling.liu@cc.gatech.edu*

Manuscript received ; revised

Several automated bug localization techniques have been proposed to help developers focus on potentially buggy files [29], [30], [43], [49]. We can broadly categorize the existing approaches into two groups: dynamic with runtime test cases and static without execution traces.

The **dynamic** approaches usually locate bugs by collecting and analyzing program data, breakpoints, or the execution traces of the system [5], [13], [31]. These approaches rely on passing or failing execution traces of a set of test cases under certain input conditions. Spectrum-based fault localization [1], [46], [47] and model-based fault localization [8], [19] are two well-known dynamic approaches. The Dynamic approach is time-consuming and expensive, its accuracy highly relies on the quality of the test suite, and most of the test suite do not have enough code coverage to locate bugs [4]. For real-world programs, we may have no test cases.

The **static** approaches do not require execution traces and can be applied to the system at any stage of the software development. They only rely on bug reports and source code files to localize bugs. The traditional information retrieval (IR) techniques are widely used for bug localization [29], [32], [49]. These IR-based approaches usually rely on the text similarities between bug reports and source code files to compute a ranked list of source code files based on their similarities to a given bug report. However, the performance of these approaches is limited by the semantic gap between a bug report and the source code files, because bug reports and source code files may not always share common textual tokens or synonyms [16], [43]. To address the mismatch of text similarity, several enhancement techniques have been proposed using other dimensions of features, such as the structured Information Retrieval [32], [48], semantic information [16], [18], bug-fixing history [37], are proposed to improve the performance of bug localization. Although

these approaches demonstrate the possibility of combining multiple features compared to the traditional IR approaches that rely on only text-similarity, how to effectively combine multiple dimensions of features for bug localization has rarely been investigated.

Despite the success of automated bug localization in existing empirical studies, the reported accuracy has been in the range of 10% to 40% and remains to be too low to be practically useful. [2] has shown that there is a significant lexical mismatch inherent between the natural language text in the bug report and the programming language in the source code files. To bridge the semantic gap, machine Leaning (ML) techniques are used. These ML approaches typically adopt the trained models to match the topics of bug reports with those of source files or classify the source files into multiple pre-defined class labels using previously fixed source files as classification labels [14]. For example, [36] used n-gram language model to generate a list of probable bugs. [44] used a learning to rank approach for adaptive ranking based on 19 features from different software artifacts, such as bug reports, source code, API description and Bug-fixing History. However, it is unclear which combinations of features are effective for a bug localization model. Some features may incur misclassification and hurt the localization performance.

Deep learning is known to perform well for some natural language processing (NLP) problems [7], credit scoring [25], [26], and medical problems [27], [45]. In the software engineering area, Huo, et al. [12] used an NP-CNN model to extract program structure and learn unified features from bug report and source code. Xiao, et al. [41], [42] combined CNN with IR based methods to locate buggy files. These deep learning techniques improve the performance by 5% to 10% by considering the textual semantics features, but they suffer from ignoring API documents and Bug-fixing History.

In this paper, we make the following two major contributions: identifying the core dimensions of features that are critical for effectively bug localization and proposing a multi-dimension deep learning model that can capture the complex and non-linear correlations among the core dimensions of features. In particular, We propose a Multi-Dimension Convolutional Neural Network model for bug localization, denoted by MD-CNN, which fuses multiple statistical dimensions of features between bug reports and source files.

We evaluate the MD-CNN model on 22,747 bug reports from six open source projects: AspectJ, Eclipse, Birt, SWT, JDT, and Tomcat. The experimental results show that our MD-CNN outperforms the existing representative bug localization techniques in terms of the Mean Average Precision (MAP) and the number of bugs successfully localized in the top 1, 5, and 10 matched source code files.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 presents an overview of our MD-CNN model. Section 4 describes the statistical extractions of five important bug-source correlation dimensions of features. Section 5 presents the design of our iterative CNN model training to learn the non-linear relationships among five statistical dimensions of features independently extracted from bug reports, source code files, and bug fixing history. We describe our experimental setup, metrics, and evaluation objectives and report our experimental evaluation results in Section 6 and conclude the paper in Section 7.

2 RELATED WORK

2.1 IR-based Bug Localization

Most of the existing approaches for bug localization are IR-based [20], [29], [43], [49], such as Vector Space Model (VSM) [21], [33], Latent Semantic Indexing (LSI) [22], [28] and Latent Dirichlet Allocation (LDA) [20], Unigram Model (UM) [29] and Cluster Based Document Model (CBDM) [29]. [29] presents a good survey to evaluate many standard IR-based bug localization methods, and they show that LDA and LSA did not outperform VSM and UM. [30] proposed BugLocator model using a revised Vector Space Model (rVSM), which performs bug localization based on previously fixed similar bugs and document length. [32] extracts and models code structures like classes, methods, variables and comments from source files, demonstrating the effectiveness for bug localization. [43] proposed an adaptive ranking algorithm using a learning-to-rank technique to derive the weights for combining features using a linear weight function. [23] combined the textual similarity between a bug report and a source file and the structural similarity between the stack trace and the code elements. The structural similarity was measured by computing the minimum distance between a stack trace and a code element. [38] proposed AmaLgam+, a method that locates buggy files by integrating five data sources: version history, similar reports, structure, stack traces, and reporter information.

The main drawback of the IR-based approaches is the use of a linear combination of multiple similarity features, even though the weights are trained on historical solved bug reports using adaptive learning techniques.

2.2 ML-based Bug Localization

Machine learning (ML) based techniques have gained popularity for bug localization. [40] proposed an approach to use a trained BP neural network to localize faults utilizing information about the testing coverage of the statements in the program. [24] proposed BugScout, which used an extended LDA to estimate the topics of a new report and compare it with the topics of all source files. [14] used Naive Bayes to build a two-phase recommendation model, which wipes out "unpredictable" bug reports in phase 1 and applies Bayes model to assign a set of source files to a bug report in phase 2, using only the names of fixed files as labels in the training process, incapable of handling a new bug report that had never been fixed before. [44] used learning to rank with multiple ranking features from bug reports and source files. [12] proposed a unified framework to combine the LSTM and CNN model based on program structure and sequence nature of source code. [16] built HyLoc that combines deep learning with IR techniques using six deep neural networks (DNNs), two for feature extraction, two for projection, one for relevancy estimation and one for feature combination. It suffers from the high cost of training and the difficulty of adjusting the weights of the six models during training. [42] proposed an enhanced CNN by considering

bug-fixing experience with a new rTF-IDuF method and a pre-trained word2vec technique.

Existing machine learning based approaches for bug localization focus more on extracting semantic information in bug reports but ignored many useful IR-base features. For example, historical bug reports and file revision history provide useful hints that can help bug localization. In this paper, we combine the best of both worlds by selectively utilizing the conventional IR techniques to perform statistical feature extractions and then construct feature matrices as the training input dataset and configure the multi-layer convolutional neural network to produce our MD-CNN model. We show through extensive experiments on six datasets that our MD-CMM outperforms existing representative approaches.

3 Solution Approach: An Overview

In this section, we first give a brief overview of the bug localization and then describe the assumptions we make for the design and implementation of our MD-CNN model to bug localization. We will present each of the five dimensions features and statistical feature extraction methods in the next section.

Background. A software *bug* is a coding defect or mistake in a software module, which may cause an unintended behavior at runtime of the software execution. A *bug report* is a text document that is intended to provide information to help in fixing a bug found by a developer or a user. A *source code file* (source file) is a software program that is written in a programming language, such as Java, C or C++. A software project refers to the development of a software product. A large number of bug reports may be generated during both the development and the deployment life cycle of a software product. Upon receiving a new bug report, the developers of the software project may need to locate the buggy source code files that contain the bug, then reproduce the bug and perform code reviews to find the cause, finally fix the bug. The bug localization is a task of ranking all the source files with respect to the probability of containing the cause of the bug, and Automated bug localization is critical for the productivity of the developers to fix the newly reported bug.

Assumption. In the development of the MD-CNN bug localization model, we assume that for a repository of software projects, there are three repositories available: a repository of historical bug reports, a repository of source code files, and a repository of bug fixing history. Let SF and BR denote the set of N_s source files and the set of N_b bug reports, i.e., $|SF| = N_s$ and $|BR| = N_b$. Each source file $s \in SF$ contains a number of fields, such as the source file identifier, the class, the method, the variable, the comment, and/or API documents, and so forth. Each bug report $b \in BR$ consists of the bug identifier and the bug text that can be further divided into the summary and the description. Each bug fixing history record consists of the bug identifier, the time stamp t_b when the bug report is fixed, and the set of source file identities associated with this bug b.

MD-CNN Design Overview. The development of MD-CNN bug localization model consists of two phases: model training and model deployment. Figure 1 presents the overall design framework.

MD-CNN Model Training. There are two primary tasks in the model training phase. The first task is to perform the statistical feature extraction. The goal of this task is to prepare the training dataset by preprocessing the raw input data from a set of historical bug reports stored in the bug tracking system, which correspond to the target software system, a set of source code files of the target software system, including the API documents of the classes and the interfaces used in the source code, and a repository of bug fixing history, and generate a set of statistical features that capture the varying types of relationships between bug reports and source code files. In the first prototype of MD-CNN, we extract five independent features from these three software repositories: (1) text similarity between bug report and source file; (2) bug-source relationship by similarity of bug reports; (3) bug-source relationship by recently-fixed source files; (4) bug-source relationship by class name Similarity; (5) bug-source relationship by structural similarity. Given a bug report, each of these five features will produce a ranking score for each of the N_s source files.

The second task of our model training phase is to construct a feature matrix of size $5 \times N_s$ for each bug report in the training set, say N_b. We provide N_b training inputs, each is represented as a feature matrix of N_s columns and five rows, and corresponds to a bug report in the training set. In each feature matrix, a column corresponds to a source file in the training set with the five statistical features (ranking scores in the range of $[0, 1]$ as its row values, which are the output generated by the feature extraction task for each pair of source file and bug report. We train the MD-CNN model for bug localization by taking the collection of N_b feature matrices as the training input, and configuring a CNN model with varying number of kernels in convolutional layers to capture the complex and non-linear relationships among different combinations of the five statistical features across the N_s source files and the N_b bug reports.

The output layer is a concatenation of the deep semantics learned by different ways of combining the features across N_b bug reports and N_s source files and their bug fixing history. The trained model is generated in multiple epochs iteratively, and each epoch consists of multiple iterations. The model training is concluded after the validation phase, which performs accuracy testing of the trained model on the validation dataset. Usually $90\% : 10\%$ or $80\% : 20\%$ ratio is used to split the input dataset into training dataset and validation dataset.

Model Deployment. In the deployment phase, the pre-trained MD-CNN model for automated bug localization prediction will be performed upon request. When a new bug report is received, the model prediction will be triggered and it will first parse the bug report against all N_s source files to generate a feature matrix. Then this feature matrix will be sent as a query input to the pre-trained MD-CNN model to perform bug localization. The output of the model prediction is the ranked list of source files, of which those source files with high probability scores are considered as the most buggy files as they are the best matches to the new bug report.

Let $f_{MD-CNN}(\theta, r)$ denote the trained MD-CNN model with r as the query with the new bug report and θ as the model parameters, which include the number of hidden

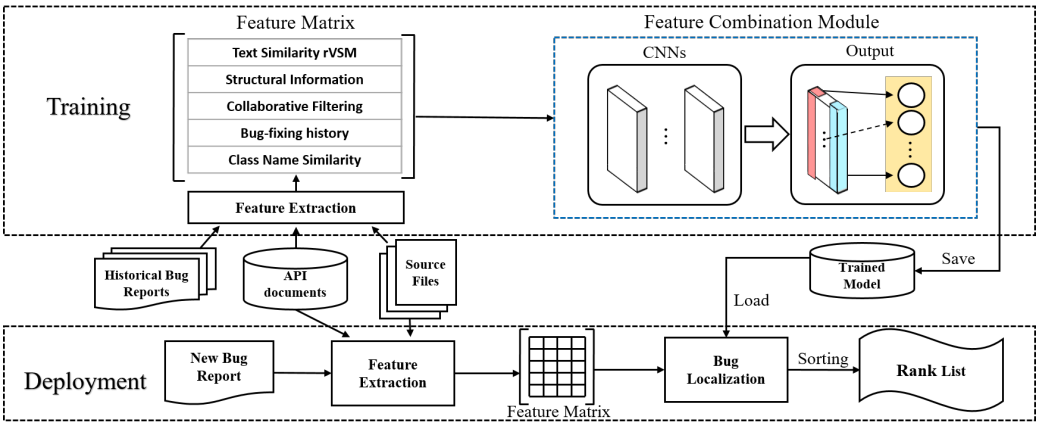

Fig. 1. The Overall Framework of MD-CNN

layers used by the MD-CNN model, the number of kernel filters W_i $(i > 1)$ used for learning the varying patterns of hidden and non-linear relationships among different features across different source files and across different bug reports, with the goal to produce the best matching of a bug report to the correct source files with high confidence. The output of $f_{MD-CNN}(\theta, r)$ is a probability vector of size N_s for the query r with the top k highest scores as the top k best source files that match the new bug report r. We use a number of accuracy metrics to measure the effectiveness of the bug localization using MD-CNN by comparing it with existing representative approaches. We next provide a more detailed description of the two tasks in our model training phase in Section 4 and Section 5, and then present our experimental setup and evaluation results in Section 6.

4 Statistical Feature Extraction

The first task of developing the proposed MD-CNN model for automated bug localization is to extract important dimensions of features using statistical analysis techniques, aiming to capture different types of observable features between each pair of a bug report and a source file, denoted by (b, s), $\forall b \in BR$, $\forall s \in SF$. For each pair (b, s), we extract k features from them and build the feature vectors $Socre(b, s) = [Score_i(b, s)]_{1 \le i \le k}$.

In the first prototype development of MD-CNN, we identify five important features that capture the observable features between a bug report and a source file: (1) the textual similarity to the source files, (2) the normalized cosine similarity to historical bug reports, (3) the normalized cosine similarity to recent buggy source files, (4) the class name similarity to the source files, and (5) the structural similarity to the source files. We describe the statistical method to extract and represent each of these five features in the remaining of this section. Each feature extraction algorithm will take the data input from the three repositories and output a similarity score for each pair (b, s). Table 1 provides a summary.

Upon completion of the feature extraction task, the MD-CNN model training task will construct the training dataset, consisting of N_b feature matrices, each of size $5 \times N_s$, to train our MD-CNN model iteratively to learn the hidden and non-linear relationships among N_b bug reports and N_s source files based on the five features that are extracted to characterize each source file based on a given bug report in the training set. The goal of the trained MD-CNN model is to produce a ranked list of source files for each bug report with high prediction accuracy.

TABLE 1
Features Used in the MD-CNN Model

Dimension	Formula
Text Similarity	$Score_{t-sim}(b,s) = g(n_t) \times cos(b,s) = \frac{1}{1+e^{\gamma mm(n_t)}} \times \frac{\vec{b} \cdot \vec{s}}{\|\vec{b}\| \|\vec{s}\|}$
Similar Bug History	$Score_{cf-sim}(b,s) = \sum_{i=1}^{k} \frac{1}{i} sim - rank(b, B(s))$
Bug-fixing History	$Score_{h-sim}(b,s) = \sum_{s \in H_m} \frac{1}{1+e^{-\frac{12 t_{elapse}(s,b)}{m} + w(s)}}$
Class Name Similarity	$Socre_{c-sim}(b,s) = \begin{cases} max\_len(cn) & if\ cn \in s.class \cap b.class \\ 0 & otherwise \end{cases}$
Structural Similarity	$Score_{s-sim}(b,s) = \sum_{b_p \in b} \sum_{s_p \in s} sim(b_p, s_p)$

4.1 Text Similarity

In general, a bug report b is expressed in a natural language, and a source code file s is expressed in a programming language. To compute text similarity between each pair (b, s), we first view bug reports and source code files as text documents under the vector space model (VSM) by employing tokenization to transform each bug report or each source file into a bag of words using the vocabulary V of all text tokens from the repository of source files and the repository of bug reports. The typical tokenization toolkit, including the NLTK package[1], the Porter stemmer[2], is used. Thus, we represent each source file s and each bug report b in the format of term vector of size $|V| = n$, denoted by $\vec{b} = [w_{i_1}, w_{i_2}, \cdots, w_{i_n}]$ and $\vec{s} = [w_{j_1}, w_{j_2}, \cdots, w_{j_n}]$ respectively, where n is the total number of the terms in V, $w_{i_1}, \ldots, w_{i_n}$ denote the term weight for terms in document b and $w_{j_1}, \ldots, w_{j_n}$ denote the term weight for terms in document s. We also represent BR as the set of bug reports (i.e., $\vec{b} \in BR$), SF as the set of source files (i.e., $\vec{s} \in SF$), and the documents $D = BR \cup SF$. One example term weight function is the term frequency tf, which gives the frequency of the term $v_i \in V$ appeared in the respective document, be it a bug report b or a source file s. The most representative weight function in the classical VSM is the TF.IDF model. The term weight w is computed based on both the term frequency (tf) and the inverse document frequency (idf).

To address the bias problem introduced due to long and short documents, we adopt the rVSM (revised Vector Space Model) by incorporating an improved term-frequency variant as larger source code files may have higher probability of containing a bug [7], and [49] has shown that rVSM has

1. http://www.nltk.org/
2. https://sourceforge.net/projects/porterstemmer/

shown better performance than the classic VSM for bug localization. Let d_t denote the number of documents that contain the term t and $d_t \leq |D|$. Let f_{td} denote the number of occurrences of a term t in a document d ($d \in D$). In rVSM, the logarithm variant of $tf(t,d)$ is used to help smooth the impact of high frequency terms. We compute the term frequency $tf(t,d)$, the inverse document frequency $idf(t,d)$ in (1) and compute the term weight w_t in each document vector of size n by (2).

$$\begin{aligned} tf(t,d) &= log(f_{td}+1) \\ idf(t,D) &= log(\frac{|D|}{d_t}) \end{aligned} \tag{1}$$

$$w_{t\in d} = tf(t,d) \times idf_{t,D} = log\,(f_{td}+1) \times log(\frac{|D|}{d_t}) \tag{2}$$

After the tokenization preprocessing of all the input bug reports and source files, and the completion of tf-idf weight computation, we use the standard cosine similarity in Equation (3) to compute the text similarity between a bug report and a source file, which is the inner product of the two vectors $\vec{b}$ and $\vec{s}$, denoted by $\vec{b} \cdot \vec{s}$, normalized by their Euclidean norm.

$$cos(b,s) = \frac{\vec{b} \cdot \vec{s}}{\|\vec{b}\|\|\vec{s}\|} \tag{3}$$

However, using the cosine similarity directly to measure the text similarity between a bug report and a source file may introduce bias. This is because bugs are often localized in one method or some small code fragment of a large source code file. For a large source file, the normalized Euclidean norm will be large, which will lead to a very small cosine similarity, even though the bug report is highly relevant to the source file. This motivates us to introduce Equation (4) to compute the text similarity score between a bug report b and a source file s.

$$\begin{aligned} Score_{t-sim}(b,s) &= g(n_t) \times cos(b,s) \\ &= \frac{1}{1+e^{\gamma_{mm}(n_t)}} \times \frac{\vec{b} \cdot \vec{s}}{\|\vec{b}\|\|\vec{s}\|} \end{aligned} \tag{4}$$

where n_t denotes the total number of distinct terms in the source file s, $g(n_t)$ models the length of document s, and $\gamma_{mm}(n_t)$ denotes the Min-Max normalization method to normalize the value of n_t as the input to the exponential function e^x. e^x is a function that takes into account of larger documents during the ranking of source files for a given bug report.

To compute the text similarity feature, we extract the text of summary, description, and comments from each bug report and extract the string-literal in addition to comments and identifiers from each source file. We argue that API documentation in the source file should be included in the text similarity computation as well. In principle, the text in a bug report may have a number of common words that also appear as the technical terms in those buggy source code files that found to match the bug report. In practice, we may find that this is not always true for a fair number of bug reports and source files. For example, Figure 2 shows an example from the Eclipse project. Bug report 407505 describes a defect that the hidden editor area will be shown when Maximize/Restore is called, and **MinMaxAddon.java** is known to be relevant to this bug report. Indeed, we

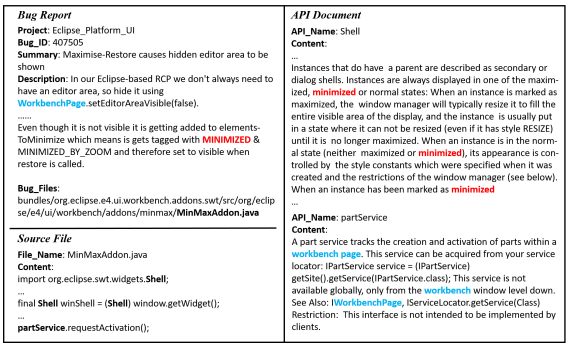

Fig. 2. Example of a bug report and a corresponding source file in the left box, compared to including the API specification in the source file text similarity computation shown in the right box. The colored text indicates some common word tokens shared by the bug report and the source file with its associated APIs.

observe from Figure 2 that the bug report 407505 and the corresponding source file MinMaxAddon.java do not have any tokens in common, and consequently their cosine similarity is zero, failing to capture the actual relevance of the bug report to this source file. However, there are several ways to bridge the lexical mismatches between natural language texts in bug reports and technical terms in source code. For example, the API specification of the classes and various interfaces [43], [44] can be a good additional channel to use in the text similarity computation. In the right box of Figure 2, the API document for the classes **Shell** and **partService** has some tokens, such as *minimized* and *workbench page*, which also appear in the bug report 407505. By obtaining the text description of the classes and interfaces through term extraction from their API speciation, we can further enhance the coverage of text similarity based linkage of a bug report with those potentially relevant source files. Thus, our feature extraction and tokenization procedure is also employed for each class and each method in a source file. We extract the textual content of their API descriptions for classes and methods mentioned in both source files and bug reports and integrate these additional tokens for the text similarity computation and feature extraction.

4.2 Similarity to Historical Bug Reports

There are a large number of historically fixed bug reports in the bug reporting system repository. Many similar bug reports may be relevant to the same source code files. Thus, we examine the bug fixing history to extract those previously fixed bug reports that are textually similar to the current bug report. Based on this set of fixed bug reports, we can locate those source files that are associated with the historical bug reports because it is highly likely that these source files are also relevant to the current bug report. It can be useful to rank them based on their textual similarity score to the current report. Figure 3 shows the examples of three bug reports that are relevant to the same class file *AjBuildManager.java*. We also observe that the three bug reports share numerous keywords, such as *aspectjrt*, *classpath*, *compile*, *etc*. These bug reports are similar and related to the same buggy source code file.

Motivated by these observations, one can leverage the fact that one source file may correspond to multiple bug reports to compute the similarity between a new bug report b and a source file s. Let $br(b,s)$ denote the set of historical

Source code file: **AjBuildManager.java**	
Bug ID: 272591	Summary: couldn't find **aspectjrt.jar** on **classpath**
	Description: I am using the **aspectj** runtime **jar** that is in the spring source bundle repository. The have renamed their jar to match their naming conventions and it is causing the warning to occur. Their bundle is named com.springsource.org.**aspectj.**runtime-1.6.3.RELEASE.jar. It would be nice if this **warning** was not printed out in this case.
Bug ID:34951	Summary: NPE **compiling** without **aspectjrt.jar**
	Description: **Compiling** spacewar without specifying **aspectjrt.jar** on the **classpath** causes a NPE. Expected an error message "**aspectjrt.jar** required". Steps to reproduce: 1) install latest 2) cd doc/examples3) java -jar ../../lib/aspectjtools.jar -verbose @spacewar/debug.lst Result :NPE in attached log
Bug ID: 112830	**Warning** "couldn't find **aspectjrt.jar** on **classpath**"
	The **compiler** makes this warning if "**aspectrt.jar**" file has a different name like "**aspectrt-1.3.jar**", which is the case when **compiling** with maven.

Fig. 3. Bug reports that are similar with a single source file

bug reports associated with a source file s and are fixed before the current bug report b. Thus, Equation (5) computes the textual similarity between the current bug report b and the summaries of all the bug reports in $br(b, s)$.

$$Score_{hbs-sim}(b, s) = cosine(b, br(b, s)) \quad (5)$$

Although [44] implements this approach using the conventional collaborative filter (CF) method, we argue that using a simple sum of the similar bug reports in Equation (6) may introduce bias and inaccuracy.

Thus, in our MD-CNN model, we propose to use Equation (6) instead. It improves Equation (5) from two aspects. First, we propose to normalize the CF score of the similar historical bugs for each source file. Let B_s denote the set of bug reports for which the source file s was fixed before the current bug report b was received. k is an integer from 1 to n. $cosine(b, b')$ is the cosine similarity between the vector b and the vector $b' \in B_s$. Let $sim-rank(b, B(s))$ be the similarity ranked list in descending order. Let k be a system-supplied parameter ($3 \leq k \leq n$). By default, $k = 3$ is used in our first prototype. The pseudocode is provided in Figure 4.

$$Score_{cf-sim}(b, s) = \sum_{i=1}^{k} \frac{1}{i} sim - rank(b, B(s)) \quad (6)$$

1. Suppose a new bug report is b, a set of past bug reports is $\{b_1, b_2, b_3 \ldots b_n\}$, where b_i is respect to the same source code file f_1 which was fixed before b was reported.
2. Similarity between b and a set of past bug reports $\{b_1, b_2, b_3 \ldots b_n\}$ is $\{s_1, s_2, s_3 \ldots s_n\}$. The relevance score between b and b_i is computed using cosine similarity$s_i = sim(b, b_i)$, s_i is a normalized value using sigmoid function.
3. Let $\{s_1, s_2, s_3 \ldots s_n\}$ *is* sorted in descending order. Then, s_1 is the biggest, and s_2 is the second, and so on.
4. Inspired by the reinforcement learning, we define the total CF score S is as follows: $S = \sum_{i=1}^{k} \alpha s_i$, where α is a attenuation factor, $\alpha = \frac{1}{i}$.
5. Then, $S = s_1 + \frac{1}{2}s_2 + \cdots + \frac{1}{k}s_k$, $k = 3$ is used in our first prototype($if\ n < 3, k = n$).

Fig. 4. The main steps of the improved collaborative filtering method

4.3 Similarity to Recent Buggy Source Files

This third feature extraction leverages the change history data of source code in the version control systems. When a bug was located in the source files, the developer needs to fix the buggy files. The fixing of bugs in a buggy source file may introduce new bugs before the bug is fixed. [15] reported that software faults do not occur in isolation, but rather in bursts of several related faults. This motivates us to utilize the bug-fixing history to identify and predict fault-prone source files [37]. Figure 5 shows three bug reports for the project JDT. The last two bug reports occurred after the first bug 272354 was reported. We observe that the second bug report 272418 was only five hours after the first bug report 272354 was received and the third one 274041 is 13 days after the second one was reported, and all three of them were located in the same buggy source file *P2Utils.java*. This shows that a buggy source file may generate new bug reports either before or after the previously reported bugs have been fixed. We conjecture that a source file is more likely to contain faults if it has recently been changed by fixing bugs. Put differently. It is highly likely to locate new bugs from the most recent buggy files.

Bug_ID : 272354
Report_Time : 2009-04-15 14:21
Modify_Time : 2009-04-16 03:00
Buggy_Files :
org.eclipse.jdt.junit.core/src/org/eclipse/jdt/internal/junit/buildpath/**P2Utils.java**
org.eclipse.jdt.junit/src/org/eclipse/jdt/internal/junit/buildpath/**P2Utils.java**

Bug_ID : 272418
Report_Time : 2009-04-15 19:57
Buggy_Files :
org.eclipse.jdt.junit.core/src/org/eclipse/jdt/internal/junit/buildpath/**P2Utils.java**

Bug_ID : 274041
Report_Time : 2009-04-28 10:07
Buggy_Files :
org.eclipse.jdt.junit.core/src/org/eclipse/jdt/internal/junit/buildpath/**P2Utils.java**
org.eclipse.jdt.junit/src/org/eclipse/jdt/internal/junit/buildpath/**P2Utils.java**

Fig. 5. Three bug reports corresponding to the same source code file

Inspired by the defect prediction technique called Time-Weighted Risk (TMR), which provides the bug-fixing commits on Google systems [17], we introduce time based decaying method to leverage the most recent fault-prone source files in the bug-fixing history. Let BFH denote the repository of bug fixing history with each entry representing the bug report b, the buggy source file s fixed at time t, i.e., $(b, s, t) \in BFH$ and H_m denote the set of buggy source files that are found in m days before receiving the bug report b at t_b. $H_{m,b,t_b} = \{s | s \in SF, (b, s, t) \in BFH, m \geq (t_b - t)\}$. m is set empirically at the system initialization and configuration time. We will include the experiments on the performance impact of the parameter m in Section 6. Let $t_{elapse}(s, b)$ is the number of days that have elapsed between a bug-fixing commits and a newly submitted bug report b. Equation (7) defines the similarity to the recent buggy files. $w(s)$ denotes the shortest time between a bug-fixing commit for the source file s and the current bug report b, showing the importance of the source files in terms of recency. It is defined in Equation (8). The larger the $w(s)$ value is, the smaller the output of the similarity score.

$$Score_{h-sim}(b, s) = \sum_{s \in H_m} \frac{1}{1 + e^{-\frac{12 t_{elapse}(s,b)}{m} + w(s)}} \quad (7)$$

$$w(s) = min_{s \in H_m, t_{elapse}(s,b) \leq m} t_{elapse}(s, b) \quad (8)$$

Consider Figure 5 again and assume that the third bug report 274041 is new, and the w value of the source file *P2Utils.java* is 13 because the bug was recently found in *P2Utils.java* about 13 days ago (2009-04-15). If we set m to be 60, then $t_{elapse}(s, b)$ of the source file *P2Utils.java* is 13, 13, and 56 respectively since the bugs were also found in the same source file on 2009-03-03. So the similarity score to the recent buggy file *P2Utils.java* for the bug report 274041 is 0.141.

There are other techniques to make defect prediction. [15] proposed BugCache method which use locality information of the previous defect and maintains a relatively short list of most bug-prone source code files (or methods). Our experiments show that recent bug-fixing history is more effective than bug-fixing frequency in training our MD-CNN bug localization model.

4.4 Class Name Similarity

In many bug reports, the class names are found in the summary or the description, which provides a useful indicator of the corresponding class files, which may be relevant to the bug report. For example, the description after the tokenization of the bug report 255600 for the SWT project contains the class names *ViewerAttributeBean*, *viewer*, *attribute* and *bean*, but only the longest name *ViewerAttributeBean* is the relevant buggy source file. Also when the class name is longer, it is more specific [44]. Thus, we design the fourth feature extraction method, which computes the statistical similarity between a bug report b and a source file s by checking whether the name of each class in the source code file is also included in the bug report. For all the class names present in the bug report, we use the maximum length of the class name as the similarity value for the $Score_{cn}(b, s)$, and if no common class names exist, the score is set to zero.

Let $s.class$ denote the set of class names in a source file s, and $b.word$ denote the set of words in the bug report b. Let $max\_len(b, s)$ denote the longest class name of the source file s which appeared in the bug report b, i.e., $max\_len(b, s) = \{length(cn) \,|\, cn \in s.class \cap b.word, \forall cn' \in s.class \cap b.class, cn' \neq cn, length(cn') \leq length(cn)\}$. We compute the class name similarity by Equation (9).

$$Socre_{c-sim}(b, s) = \begin{cases} max\_len(b, s) & if\ cn \in s.class \cap b.class \\ 0 & otherwise \end{cases} \tag{9}$$

4.5 Structural Similarity

Recall the discussion in the text similarity section, when a bug report is relatively small compared to the large source code files, if the bug reported only matches one text segment of the source file, then the cosine similarity can be misleading because the large source file can lead to the large Euclidean norm of b and s, resulting in very small similarity value, even though the source file is highly relevant to the bug report. One straightforward approach to address this problem is to introduce the structural similarity between b and s. For example, we can segment the source code file into methods to compute per-method based cosine text similarity to the bug report [18]. Another recent proposal is BLUiR [32]. Inspired by BLUiR parsers, we partition each bug report b into two segments: $b.summary$ and $b.description$, and partition each source file s into four structural segments: $s.class$, $s.method$, $s.variable$ and $s.comment$. Each of these text segments are represented as a vector of size $n = |V|$ following the cosine similarity computing procedure (4), and then sum the eight similarity scores. The structural similarity feature can be computed as follows:

$$Score_{s-sim}(b, s) = \sum_{b_p \in b} \sum_{s_p \in s} sim(b_p, s_p) \tag{10}$$

where b_p is one of the two text segments (partitions) in a bug report b, s_p is one of the four text segments (partitions) in a source file s, and $sim(b_p, s_p)$ is the cosine similarity of the two vector representations of b_p and s_p. The output of the structure similarity feature is a set of N_s scores, one for each source file in the source file repository SF ($|SF| = N_s$).

5 MD-CNN Modeling

5.1 Feature Scaling with Min-Max Normalization

For a given feature ξ, we set ξ_{min} and ξ_{max} as the minimum and the maximum observed values in the training dataset. If a feature value ξ in the test dataset is larger than ξ_{max}, we set ξ to be ξ_{max}; and if $\xi < \xi_{min}$, then we set $\xi = 0$. For $\xi_{min} \leq \xi \leq \xi_{max}$, if $\xi > 1$, then we need to employ the min-max normalization to scale ξ to the value range of $[0, 1]$ by $\frac{\xi - \xi_{min}}{\xi_{max} - \xi_{min}}$.

5.2 Feature Combination

Different types of features are extracted by employing different statistical analysis techniques to compute the similarity scores between a bug report b and a source file s. In the first prototype of MD-CNN, we provide five types of features: text similarity ($t\text{-}sim$), similarity to historical bug reports ($cf\text{-}sim$), similarity to recent buggy files ($h\text{-}sim$), class name similarity ($c\text{-}sim$) and structural similarity ($s\text{-}sim$). Each of these features serves as a useful indicator for linking a bug report to the most relevant source files for bug localization. Thus, a careful combination of these features can significantly improve the performance of bug localization. Existing methods [18] use weighted linear combination by learning how to derive the proper weight value for each of the different feature types. However, linear models fail to capture the hidden and non-linear relationship among the different types of features across bug reports and source files. The table in Figure 6 shows the five similarity scores between bug report 263837 and three source files, which shows the sample bug report 263837 with three class structural similarity ($s\text{-}sim$) but ranks the highest similarity compared to historical bug reports ($cf\text{-}sim$) and the class-name similarity feature ($c\text{-}sim$). This indicates that given a bug report, the same source file under different features has different importance, and the same feature has different importance for different source files, demonstrating non-linear relationships among the different features. For example, for the feature of Text Similarity, the values of *BceClassWeaver.java*, *BceTypeMunger.java* and *BcelWeaver.java* are 0, 0.01 and 0.86 respectively. The most important feature for *BcelClassWeaver.java* is the similar bug history (0.87), but the most important feature for *BcelClassWeaver.java* is the text similarity (0.86). This motivates us to replace the linear model for combining the bug-source file similarity features for bug localization by using deep neural networks such as convolutional neural network, which is known to capture the latent and non-linear relationships among the features and holds the potential to outperform the linear feature integration model for bug localization.

Project: AspectJ
Bug_ID: **263837**
Summary: Error during Delete AJ Markers
Description: Error sent through the AJDT mailing list. I believe this is an LTW weaving error, so not raising it against AJDT.
Bug_Files: weaver/src/org/aspectj/weaver/bcel/**BcelClassWeaver.java**
weaver/src/org/aspectj/weaver/bcel/**BcelTypeMunger.java**
weaver/src/org/aspectj/weaver/bcel/**BcelWeaver.java**

scores after normalization	*t_sim*	*cf_sim*	*h_sim*	*c_sim*	*s_sim*
BcelClassWeaver.java	0	0.87	0.03	0.43	0
BcelTypeMunger.java	0.01	0.59	0.8	0	0
BcelWeaver.java	0.86	0.71	0.81	0.43	0.08

Fig. 6. An example of the non-linear relationship between the features

5.3 MD-CNN Model Training and Model Prediction

Motivated by the success of Convolutional Neural Networks (CNNs) for learning complex non-linear relationships between inputs and outputs [10], we take a two phase approach to develop a CNN-based non-linear feature combinator, aiming to compute the ranking scores for all source code files for each given bug report in the bug reporting repository and output the ranked list of source code files upon receiving a new bug report from the prediction query API. The first phase is the model training, which takes the training datasets and produces the trained model. The second phase is the model prediction, which takes a new bug report as a query input, transforms it into its feature matrix format, and feeds it to the trained model. The output is the bug localization result, which is a ranked list of source files by the descending order of the probability likelihood of the source files matching the new bug report.

Feature Matrix. To construct our MD-CNN model through the CNN training, we first need to create the training input matrix for training. Each input matrix is a two-dimensional feature matrix, which corresponds to a bug report, and consists of five rows and N_s columns. Each row represents one of the five features captured from our statistical feature extraction step (the file names determine the order of the source files in each row). Each column represents the five similarity scores (the order of such five similarity scores in each column is the same as the No. in Table 1) between the bug report and one of the N_s source code files. In particular, due to the source files that will change according to the change of project's version, we take a subset of source files that contains all source files that have appeared (i.e., including deleted ones). Each feature matrix can be viewed as a feature map of the bug report $b \in BR$ over the entire source code file repository BF through five different observable features. By using all the bug reports in the training set, we want to train the MD-CNN model to learn several latent and non-linear relationships among the training dataset (the N_b feature matrices of size $5 \times N_s$, such as how is the bug report correlated to different source code files in the entire collection SF, how is a source file correlated to different bug reports, and what types of hidden relationships can be extracted from the historical bug fixing history about bug reports and source files. Such latent relationship representation will be captured and expressed by the MD-CNN model through iterative DNN training and the trained model will be deployed for future bug localization prediction.

Convolutional Neural Network Structure. A typical CNN contains several convolutional layers that are often combined with pooling steps (subsampling) and then followed by a fully connected layer. Figure 7 shows the basic structure of our CNN training. The input matrix is of size $5 \times N_s$ (N_s is the total number of source files in the training sets. Given that each column of an input feature matrix represents the statistical features of a source file with respect to a given bug report. Features in different columns are independent of one another in terms of how features are extracted. Thus, the width of a neuron and of a convolution kernel is 1, which is different from image and video classification and recognition systems. To learn the non-linear relationship for different combinations of the features, we use multiple kernel filters of varying sizes. For five features, we set four sizes of kernel filters to combine 5, 4, 3, and 2 features, respectively. Thus, for the first convolution layer, we use four types of kernel filters with different region sizes (e.g., 2, 3, 4, and 5). For each convolution layer, the input of every neuron is convoluted with multiple trainable kernel filters, defined by W_i. A non-linear function, such as sigmoid or Rectified Linear Units (ReLU), is used to increase the network's non-linear properties, followed by the pooling (subsampling) procedure. A max-over-time (or mean-over-time) pooling operation is used in the pooling step to reduce the number of parameters and alleviate the over-fitting problem [34]. The size of kernel filters in this pooling layer is 4×1. Followed by the first pooling layer, we splice the results into a new feature matrix. To learn more complex feature correlation relationships between bug reports and source files, the second convolution layer uses four filters with a size of 2×1. The second pooling layer operates similarly as the first one with the size of filters in this max pooling layer set to be 3×1. Finally, the *sigmoid* function is used in the fully connected layer to increase the non-linear properties of the network. Our CNN is trained to minimize the following mean binary cross-entropy lost function [9]:

$$Lost(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{T} W * t_{ij} log(y_{ij}) + (1 - t_{ij}) log(1 - y_{ij})) \quad (11)$$

where N is the size of a batch of multiple samples, and T is the number of classes (i.e., the file names of source files). t_{ij} is the ground truth of class j of sample i, and y_{ij} is the output probability of class j of sample i. Since the number of positive samples (i.e., the buggy files) and negative samples is extremely unbalanced, we set W to solve such problem. W is the weight based on the proportion of positive and negative samples in the training data. The early stopping strategy is used that checks every two epochs to avoid overfitting, and the learning rate and batch size are set to 1e-3 and 32, respectively. The number of layers and parameters of the above CNN network are optimized by the brute force method.

6 Experimental Evaluation

We evaluate the effectiveness of MD-CNN by conducting experiments on six open source software projects and compare its performance with three existing representative bug localization approaches and a Deep Neural Networks (DNN) approach. We first describe the experimental setup, including the datasets used, the performance metrics, and the evaluation objectives in Section 6.1 and then report the evaluation results in Section 6.2.

6.1 Experimental Setup

6.1.1 Dataset.

Data Collection. For comparison, we use the same collection of datasets provided in [44]. It contains a total of 22,747 bug reports from six popular open-source projects: Eclipse Platform UI, JDT, Bir747t, SWT, Tomcat, and AspectJ. Table 2 describes the datasets in detail. All of the projects use Bugzilla as the issue tracking system and GIT as a version control system (earlier versions are transferred from CVS/SVN to GIT). All of the bug reports, source code

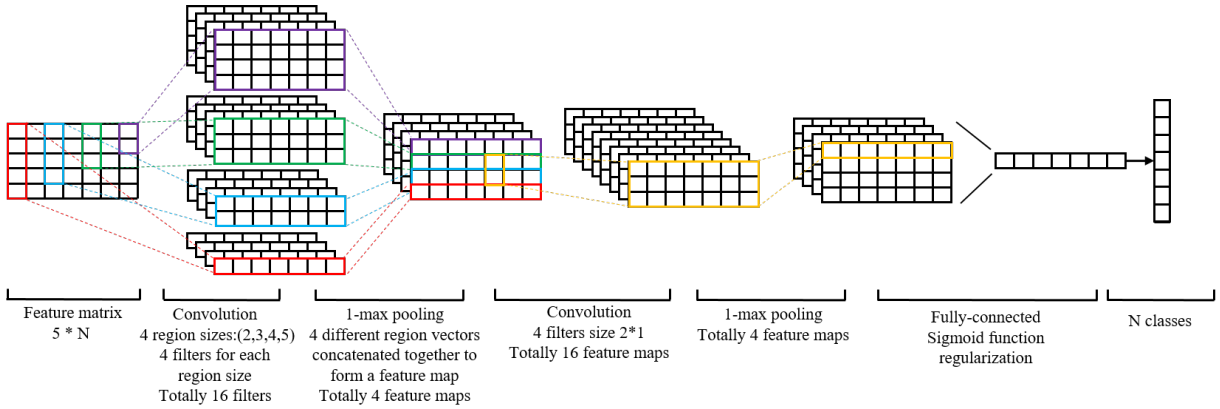

Fig. 7. The architecture of convolutional neural network

repositories, buggy files, and API specifications are publicly available at *http://dx.doi.org/10.6084/m9.figshare.951967*.

TABLE 2
Benchmark Datasets

Project	Time Range	#bug reports	#source files	#API entries
Eclipse	10/01–01/14	6495	3454	1314
JDT	10/01–01/14	6274	8184	1329
Birt	06/05–12/13	4178	6841	957
SWT	02/02–01/14	4151	2056	161
Tomcat	07/02–01/14	1056	1552	389
AspectJ	03/02–01/04	593	4439	54

Training, Validation, and Testing Data. The bug reports of each project are chronologically sorted by the bug IDs. The first 80% of the bug reports (older bugs) were used to train our model (Use the ratio 90%: 10% to split it into training dataset and validation dataset), and the remaining 20% (newer bugs) used as the testing set.

6.1.2 Evaluation Metrics.

We use three popular metrics to evaluate the effectiveness of our MD-CNN model: Accuracy@k, Mean average precision (MAP) and Mean Reciprocal Rank (MRR).

Accuracy@k: This metric measures the percentage of the bug reports that have found at least one buggy source files in the top k (k= 1,5,10,20) ranked files returned. A bug report b belongs to the top k accuracy ranked list if the top k query results contain at least one correct source file from which the bug is successfully located. The higher value the metric has, the better performance the bug localization approach offers.

MRR (Mean Reciprocal Rank): This metric measures the mean of the reciprocal rank for all queries [35]. The reciprocal rank of a query is the multiplicative inverse of the position of the first correctly located buggy file in the ranked list of returned files for a query (i.e., a new bug report). Let Q denote the set of queries (i.e., bug reports) in the test set, and $rank - pos(q)$ denote the position of the first correctly located buggy file for the query q in the ranked list of returned files. We define MRR as follows:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank - pos_{q_i}} \tag{12}$$

MAP (Mean Average Precision): This metric measures the mean of the Average Precision (AvgP) scores across all bug report queries. MAP is the most commonly used IR metric to evaluate and compare the ranking approaches [21].

$$MAP = \sum_{i=1}^{|Q|}\frac{AvgP(q_i)}{|Q|} \tag{13}$$

For a single query $q \in Q$, we can compute the average precision $AvgP(q)$ for q overall N source files as follows: Let $top - SF(k)$ denote the number of buggy files correctly located in the top k ranked list of the N source files returned and $flag(k)$ denote whether the file in the rank k is the correct buggy file: correct if $flag(k) = 1$ and incorrect if $flag(k) = 0$. Let $BuggyF(k)$ denote the total number of buggy files in the top k returned files.

$$AvgP(q) = \sum_{k=1}^{|N|}\frac{top - SF(k)flag(k)}{BuggyF(k)} \tag{14}$$

The higher the Accuracy@k, MRR, and MAP values are, the better performance and higher effectiveness the bug localization approach will offer.

6.1.3 Evaluation Plan.

Our evaluation plan consists of four evaluation objectives. First, we evaluate the effectiveness of our MD-CNN by comparing it with existing representative bug localization systems. Second, we evaluate the importance of different features on the overall performance of our MD-CNN. Third, we evaluate the impact of training data on the performance of our MD-CNN. Finally, we study the impact of multiple system parameters on the performance of MD-CNN.

Performance and Effectiveness of MD-CNN. To evaluate the performance of MD-CNN, we conduct an experimental comparison of MD-CNN with the following three representative baseline approaches on all six datasets in Table 2:

- **Learning to Rank (LR)** [44] leverages some domain knowledge through functional decompositions of source code files into methods, API descriptions, the Bug-fixing History, and the code change history.
- **BugLocator (BL)** [49] is a well-known bug localization technique, which ranks the source files based on textual similarity, the size of source files, and information about previous bug fixing results.
- The standard VSM method **VSM** [21] ranks the source files based on their textual similarity with a bug report.

- **Deep Neural Networks (DNN)** [11]. The Deep Neural Networks (DNN) is used to combine our five statistical dimensions of features.

To provide stable accuracy and reduce the errors incurred by the randomness of CNN structure and hyper-parameter settings, we train the MD-CNN model for 10 times using each dataset, and record the mean value of the results. In addition to the three metrics, Accuracy@k, MAP and MRR, we also apply the Wilcoxcon signed-rank test [39] at 95% significance level on 18 paired measurement values, which correspond to the performance metrics of Accuracy@k, MAP, or MRR on six datasets. We use the Cliff's delta (δ) [6] to quantify the amount of difference between the two models. $|\delta|$ ranges from 0 to 1, where $|\delta| = 0$ means that the results of the two models overlap completely without any difference, and $|\delta| = 1$ means that one model outperforms another model on all datasets in terms of the performance metrics. The interpretation of different delta values is given in Table 3.

TABLE 3
Different cliff's delta and effectiveness level [6]

Cliff's Delta ($\|\delta\|$)	Effectiveness Level
$0.000 \leq \|\delta\| < 0.147$	Negligible
$0.147 \leq \|\delta\| < 0.330$	Small
$0.330 \leq \|\delta\| < 0.474$	Medium
$0.474 \leq \|\delta\| \leq 1.000$	Large

Performance Impact of Each Feature on MD-CNN. We have extracted five statistical features from observable properties in the repository of bug reports, the repository of source code files, and the repository of bug fixing histories. We plan to compute the MAP of each feature on all six datasets and estimate the impact of each feature on the test performance of MD-CNN. We also use the greedy algorithm [3] to sort the five features on each dataset and measure the performance of combining features by adding the best performing feature to the model one at a time.

Impact of Model Parameters on MD-CNN. Although several parameters may impact on the performance of our MD-CNN model, in this paper, we report the impact of the following three parameters on the bug localization performance of MD-CNN using MAP measurements on all six datasets. They are the cf-sim feature (similarity to the historical fixed bugs) and the h-sim feature (similarity to the recent-fixed source files), and the number of convolutional layers.

6.2 Evaluation Results

6.2.1 Performance of MD-CNN.

Table 4 compares MD-CNN with three existing representative baseline approaches LR [44], BL [49] and VSM [21] and DNN feature combination approach using Accuracy@k metric. We make four observations.

(1) MD-CNN consistently outperforms the four baseline approaches in terms of Accuracy@k for k=1, 5, 10. For example, MD-CNN successfully locates 45.3% bugs in AspectJ, 44.8% bugs in Eclipse, 46.7% bugs in Tomcat in terms of top-1 accuracy.

Comparing to the learning-to-rank (LR), MD-CNN increased the Accuracy@1 by an average of 25.99%, the improvement at Accuracy@5 is from 8.5−39.4%, the improvement at Accuracy@10 is from 10−45.5%.

(2) Even at Accuracy@20, MD-CNN outperforms VSM and BL on all six datasets and outperforms LR on 4 out of 6 datasets with comparable performance on the other two datasets (JDT and Tomcat).

(3) Table 4 also list the p-value and Cliff's Delta for the comparison of MD-CNN with the three representative baseline methods. When the p value is less than 0.05, the comparison is significantly different. For Accuracy@1, 5, 10 and 20, MD-CNN demonstrates significant improvement over VSM (all four values $p < 0.01$) with large effect size (all four $\delta = 0.944$). Similarly, MD-CNN is significantly better than BL for Accuracy@1, 5, 10 and 20 with all four $p < 0.05$ and large effect size ($\delta > 0.7$). Finally, MD-CNN outperforms LR on Accuracy@1 ($p < 0.05$, $\delta = 0.722$), and shows small improvement over LR on Accuracy@5,10 ($p > 0.05$) with large effect size (both $\delta > 0.5$).

(4) DNN feature combination approach outperforms the Learning to Rank (LR) in most cases. Such result shows that our five statistical dimensions of features can capture the varying types of relationships between bug reports and source code files which conducive to accurately locate the buggy files. And our MD-CNN consistently outperforms the DNN feature combination approach demonstrates that our CNN model is more suitable for feature combination than DNN.

Next, we compare the performance of MD-CNN with LR, BL, VSM, and DNN in terms of MAP and MRR. Table 5 shows the results. We observe two interesting facts. First, MD-CNN consistently outperforms all three existing baseline approaches on all six datasets. AspectJ achieve MAP and MRR score of 0.41 and 0.46 respectively, and Birt achieve MAP and MRR score of 0.22 and 0.25, respectively, better than all three existing methods. For SWT, MD-CNN has the MAP score of 0.53, 562.5% higher than VSM, 39.4% higher than BL, and 32.5% higher than LR. Second, MD-CNN has the high average MRR, about 43.4% higher than the average MRR for BL, and 11.8% higher than the average MRR for LR. The consistently high MAP and MRR for MD-CNN also suggest that the overall ranking of the buggy source files located by MD-CNN is more effective than those of LR, BL, and VSM. And Table 5 also shows that MD-CNN has significant improvement over VSM in both MAP and MRR for all 6 datasets (both $p < 0.01$, $\delta > 0.889$). The improvement of MD-CNN over BL is more substantial than the improvement over LR for both MAP and MRR measures. Furthermore, the performance of the DNN feature combination approach is second only to MD-CNN.

Table 6 shows the training time on dataset and test time for one bug report of MD-CNN. In particular, we use TensorFlow to construct the CNN model, and all experiments are run on a server with Intel Xeon CPU E5-2650 and NVIDIA GPU TITAN V. We can observe that the training time is acceptable (we use early stopping strategy) and test time is reasonable(below one minute for one bug report). Since MD-CNN must reset the commit version of project continuous in feature extraction step, the feature extraction cost many time in training and test. Especially, due to the boost of GPU, the time cost by the trained CNN network (for feature combination) is much less than the time for feature extraction.

Overall, our MD-CNN has significantly improved over

TABLE 4
Performance comparison (Accuracy@k, k=1,5,10,20) of four representative baseline methods (VSM, BL, LR, DNN)

Dataset	Accuracy@1					Accuracy@5					Accuracy@10					Accuracy@20				
	VSM [21]	BL [49]	LR [44]	DNN	MD-CNN	VSM [21]	BL [49]	LR [44]	DNN	MD-CNN	VSM [21]	BL [49]	LR [44]	DNN	MD-CNN	VSM [21]	BL [49]	LR [44]	DNN	MF-CNN
AspectJ	0.116	0.251	0.374	0.402	**0.453**	0.209	0.404	0.523	0.567	**0.59**	0.285	0.48	0.637	0.738	**0.752**	0.40	0.505	0.738	0.785	**0.801**
Birt	0.043	0.111	0.124	0.173	**0.197**	0.096	0.259	0.289	0.371	**0.403**	0.117	0.321	0.381	0.513	**0.530**	0.178	0.399	0.489	0.568	**0.609**
Eclipse	0.185	0.271	0.397	0.402	**0.448**	0.337	0.538	0.654	0.713	**0.735**	0.422	0.616	0.743	0.805	**0.822**	0.523	0.710	0.821	0.833	**0.844**
JDT	0.097	0.181	0.334	0.427	**0.439**	0.201	0.39	0.635	0.714	**0.720**	0.287	0.502	0.729	0.800	**0.811**	0.396	0.604	**0.832**	0.827	0.831
SWT	0.044	0.198	0.313	0.418	**0.464**	0.118	0.381	0.624	0.739	**0.752**	0.199	0.496	0.75	0.852	**0.867**	0.433	0.612	0.835	0.878	**0.893**
Tomcat	0.208	0.351	0.419	0.434	**0.467**	0.487	0.651	0.715	0.743	**0.776**	0.599	0.716	0.802	0.827	**0.859**	0.680	0.815	**0.898**	0.856	0.871
Average	0.116	0.227	0.327	0.376	**0.412**	0.241	0.437	0.573	0.641	**0.663**	0.318	0.522	0.673	0.756	**0.773**	0.435	0.608	0.769	0.791	**0.813**
Improved%	+255.17	+81.49	+25.99	+9.57	-	+175.1	+51.71	+15.71	+3.43	-	+143.08	+48.08	+14.86	+2.25	-	+86.9	+33.72	+5.72	+2.78	-
p-Value	<0.01	<0.05	<0.05	-	-	<0.01	<0.05	>0.05	-	-	<0.01	<0.05	>0.05	-	-	<0.01	<0.05	>0.05	-	-
δ	0.944	0.778	0.722	-	-	0.944	0.778	0.5	-	-	0.944	0.889	0.667	-	-	0.944	0.778	0.167	-	-

TABLE 5
Performance comparison (MAP and MRR) with four representative baseline methods (VSM, BL, LR, DNN)

Dataset	MAP					MRR				
	VSM [21]	BL [49]	LR [44]	DNN	MD-CNN	VSM [21]	BL [49]	LR [44]	DNN	MD-CNN
AspectJ	0.12	0.22	0.38	0.40	**0.41**	0.16	0.32	0.44	0.46	**0.46**
Birt	0.05	0.14	0.17	0.21	**0.22**	0.07	0.18	0.21	0.23	**0.25**
Eclipse	0.20	0.31	0.44	0.47	**0.48**	0.25	0.37	0.51	0.54	**0.54**
JDT	0.12	0.23	0.40	0.45	**0.45**	0.15	0.30	0.47	0.53	**0.53**
SWT	0.08	0.38	0.40	0.51	**0.53**	0.09	0.44	0.46	0.56	**0.57**
Tomcat	0.33	0.43	0.52	0.54	**0.55**	0.36	0.48	0.55	0.59	**0.60**
Average	0.15	0.285	0.385	0.43	0.44	0.18	0.348	0.44	0.485	0.492
Improved%	+193.3	+54.4	+14.3	+2.3	-	+173.3	+41.4	+11.8	+1.4	-
p-Value	<0.01	<0.05	>0.05	-	-	<0.01	>0.05	>0.05	-	-
δ	0.944	0.667	0.5	-	-	0.889	0.667	0.388	-	-

TABLE 6
Training and Test Time of MD-CNN (in minutes)

Project	Training time on the dataset (Average)		Test time for one report (Average)	
	Feature Extraction	Feature Combination	Feature Extraction	Feature Combination
AspectJ	89	177	0.46	0.07
Birt	138	181	0.68	0.08
Eclipse	196	193	0.39	0.07
JDT	210	199	0.82	0.08
SWT	129	186	0.25	0.06
Tomcat	68	170	0.15	0.05

VSM and BL across all Accuracy@k, MAP, and MRR for all six data sets. Compared with LR, MD-CNN has a good improvement in most evaluation metrics. Compared with the DNN combination approach, MD-CNN consistently has a slight improvement. The overall ranking of the buggy source files located by MD-CNN is more efficient than LR, BL, VSM, and DNN.

6.2.2 Impact of Each Feature on MD-CNN.

TABLE 7
The MAP of each feature on six projects

Feature	AspectJ	Birt	Eclipse	JDT	SWT	Tomcat
Text Similarity	0.264	0.157	0.352	0.312	0.344	0.457
Similar Bug History	0.090	0.178	0.212	0.372	0.413	0.307
Bug-fixing History	0.247	0.049	0.089	0.042	0.135	0.037
Class Name Similarity	0.133	0.050	0.197	0.146	0.167	0.094
Structural Similarity	0.093	0.076	0.194	0.126	0.105	0.196
MD-CNN (5 combo)	**0.41**	**0.22**	**0.48**	**0.45**	**0.53**	**0.55**

Table 7 shows the performance of each of the five features on all 6 datasets. For example, for AspectJ, the system achieves the best MAP of 0.2644 using the feature $\{Text\,Similarity\}$, and performs the second place for MAP metric of 0.2469 when using the feature $\{Bug\text{-}fixing\,History\}$. However, when using the same feature $\{Bug\text{-}fixing\,History\}$ for Tomcat, the system achieves the least MAP value of 0.0373. This suggests that each feature plays a different role in different projects. From Table 7, we observe that the feature $\{Text\,Similarity\}$, which measures the lexical similarity between bug reports and source code, is the most critical feature for the projects AspectJ, Eclipse and Tomcat. In comparison, the feature $\{Similar\,Bug\,History\}$, which measures the similarity between a new bug report and the previously fixed bug reports, is the most critical feature for the project Birt, JDT, and SWT. For all six projects (datasets), Text Similarity and Similar Bug History are more critical features by examining the overall performance. Other features provide complementary information that further improves the bug localization performance.

TABLE 8
The importance of features using greedy algorithm (Feature NO.1: Text Similarity; Feature NO.2: Similar Bug History; Feature NO.3: Bug-fixing History; Feature NO.4: Class Name Similarity; Feature NO.5: Structural Similarity;)

Dataset		First	Second	Third	Fourth	Fifth
AspectJ	Feature	1	3	4	2	5
	MAP	0.264	0.359	0.387	0.405	0.412
	Improved	-	+36.0%	+7.8%	+4.7%	+1.7%
Birt	Feature	2	1	4	5	3
	MAP	0.178	0.195	0.209	0.216	0.220
	Improved	-	+9.6%	+7.2%	+3.3%	+1.9%
Eclipse	Feature	1	2	4	5	3
	MAP	0.352	0.415	0.441	0.466	0.479
	Improved	-	+17.9%	+6.3%	+5.7%	+2.8%
JDT	Feature	2	1	4	5	3
	MAP	0.372	0.413	0.431	0.444	0.452
	Improved	-	+11.0%	+4.4%	+3.0%	+1.8%
SWT	Feature	2	1	4	3	5
	MAP	0.413	0.475	0.507	0.523	0.534
	Improved	-	+15.0%	+6.7%	+3.2%	+2.1%
Tomcat	Feature	1	2	5	4	3
	MAP	0.457	0.494	0.524	0.543	0.548
	Improved	-	+8.1%	+6.1%	+3.6%	+0.9%

In the next set of experiments, we evaluate the importance of features using the greedy algorithm. Table 8 shows the result. For each project, the first row represents the order of the features when being combined. We observe that the performance of the MD-CNN model is improved when an additional feature is added. This demonstrates that every feature is useful, effective, and necessary. At the same time, we also observe that some features are particularly effective for certain data sets. For example, the performance of feature $\{Similar\,Bug\,History\}$ on AspectJ increase the MAP value by 36%, and the MAP performance of $\{Similar\,Bug\,History\}$ on SWT is 0.475, which has surpassed the LR approach [44] of 0.40 as shown in Table 5.

Our experimental results show that these five dimensions features contribute differently to each dataset in terms of the accuracy of bug localization. These results can be used to determine the trade-off between localization accuracy and system complexity.

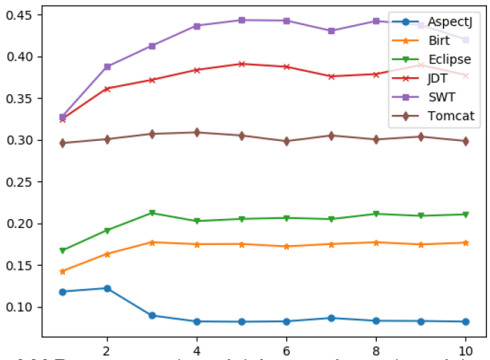

Fig. 8. The MAP measure (y-axis) for varying k (x-axis) on Similar Bug History feature (cf-sim)

6.2.3 Impact of Model Parameters on MD-CNN.

Although there are many parameters involved in training MD-CNN, we identify three parameters to show how the setting of these parameters may impact the performance of MD-CNN. The first parameter is related to the cf-sim feature, which computes the feature by similar bug histories using the set of k bug reports associated with a source file (recall Equation (6)). We want to study the setting of this k value on the performance of MD-CNN. Figure 8 shows the measurement of MAP by varying the k from 2 to 10. We observe that the MAP value for the projects JDT and SWT are gradually increasing when the value of k increases from 2 to 6. But the MAP value of AspectJ is declining as the value of k increases from 2 to 6. Also, we observe that when $k = 3$, we obtain the best performance for Birt and Eclipse platform UI dataset. By default, we set $k = 3$ in our prototype.

The next set of experiments is designed to compare the performance of the feature of similarity by recently fixed source files, which computes the feature by using the set of bug fixing files m days before the submission of the current bug report (recall Equation (7)). We want to study the setting of this k value on the performance of MD-CNN. To compare the MAP performance of MD-CNN with different settings of parameter k, we define the settings of k by the following five scenarios:ios:

A. k>3 , α=1
B. k =3 , α=1
C. k>3 , α=i
D. k>3 , α=i^2
E. k =3 , α=i

where due to an average of 80% of the buggy files in all datasets correspond to no more than 3 historical bug reports, k=3 is used in our prototype and set it in scenarios B, E. To compare the necessity of the parameter k, we do not set the value of the parameter k in scenarios A, C and D. Also, α is the weights of sim-rank in Equation (6), and i means the ranking of similarity between buggy file with its corresponding bug report (as shown as Figure 4). As shown in Table 9, more than 50% of the total number of buggy files corresponds to only one historical bug report (One file corresponds to one report) in each dataset. The number of buggy files which correspond to two historical bug report accounts for 11.0% to 21.8% of the total. We used parameter α as an attenuation factor to reduce the imbalance effect of multiple similarity accumulations on the CF score. And to verify the effect and appropriate size of the attenuation factor, we set α=1 (This means no attenuation factor is used) in scenarios A and B, α=i (According to the proportion of the buggy files in Table 9) in scenarios C and E, and α=i^2 in scenarios D.

TABLE 9
Number of buggy files which correspond to bug reports of 1, 2, 3, 4 or more respectively(#Bug Reports indicates the number of bug reports that buggy file correspond to)

#Bug Reports	AspectJ	Birt	Eclipse	JDT	SWT	Tomcat
1	763	2002	2907	2747	633	564
2	116	754	1252	636	176	168
3	52	354	561	303	65	79
4	30	191	305	201	52	35
>4	96	536	723	766	319	127
Total files	1057	3837	5748	4653	1245	973

Table 10 shows the results when the parameters are set in the scenarios of A to E. Obviously, except AspectJ, the performance of scenario A is worse than the others. The performance of scenarios B, C, D, E were somewhat mixed. For example, the performance of scenario D is best on AspectJ, Eclipse, and Tomcat, but it is worse than C on SWT. We choose scenario E to train MD-CNN because the performance of scenario E is more stable even though it is not the best by MAP measurement on some projects.

TABLE 10
The MAP of Similar Bug History when using different parameters on the six projects

Scenario	AspectJ	Birt	Eclipse	JDT	SWT	Tomcat
A	0.0801	0.0997	0.1672	0.212	0.3578	0.2601
B	0.0736	0.176	0.1973	0.3845	0.4235	0.3067
C	0.0914	0.1502	0.2087	0.3523	0.4614	0.3025
D	0.0916	0.1756	0.2135	0.3795	0.4322	0.3089
E	0.0898	0.1775	0.2124	0.3718	0.4126	0.3072

Figure 9 shows the measurement of MAP by varying the m days from 30 to 180. We observe that the m value has a high impact on AspectJ, but has relatively less effect on the other five projects. When $m = 90$ and $m = 120$, the MAP values are comparable. This motivates us to set m to be 90 as the system default in our prototype.

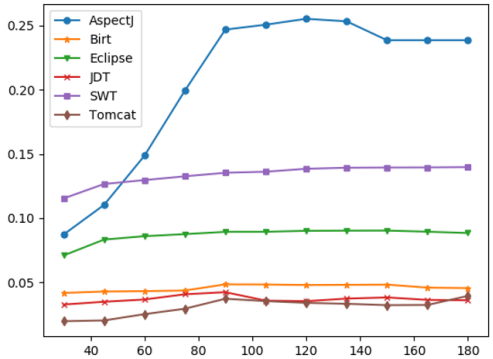

Fig. 9. The MAP of *h-sim* with varying m settings

The third parameter we evaluate is the number of convolution layers in MD-CNN and its impact on the effectiveness of MD-CNN for bug localization. We vary the number of convolution layers in MD-CNN and compute MAP and MRR for all the bug reports in all six projects. Figure 10 shows the results. We make two observations. First, the MAP scores are the lowest when the network structure of MD-CNN has only one convolution layer. As we add the

second convolution layer, the MAP scores are increased significantly for all datasets. By increasing the number of convolution layers above 2, the MAP scores are increasing very slowly. For all six datasets, the setting of 4 to 6 convolutional layers is sufficient. This is one of the motivations for the design of our CNN structure in Figure 7.

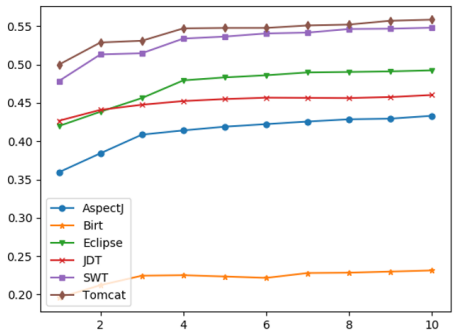

Fig. 10. MAP scores of six projects with varying number of convolution layers in MD-CNN

7 Conclusion

Bug localization is a challenging and time-consuming process. Automated bug localization techniques are desirable in practice. Although several automatic bug localization methods have been proposed in recent years, the low accuracy of these approaches makes them difficult to use.

In this paper, we present a Multi-Dimension Convolutional Neural Network (MD-CNN) model for bug localization, which creates a deep learning model to automatically localize the most-buggy source files based on a bug report. MD-CNN effectively merges five statistical dimensions of features and iteratively learns the complex and non-linear relationship between the multiple dimensions of features and the bug locations. We have evaluated our model on 22,747 bug reports and 26,526 source files from six different software systems. Compared with the existing representative bug localization techniques, MD-CNN uses fewer features but outperforms them in terms of Top 1, Top 5, Top 10, and MAP scores. MD-CNN trained the CNN network on each dataset separately, so it has a strong generalization ability on different projects. Our proposed MD-CNN also has a disadvantage, i.e., if any new files are added to the projects in the future, our model needs to be trained again for better performance.

In the future, we plan to improve the performance of our approach further and investigate more aspects of bug reports and source code, such as the inherent structural information of a program better to represent the high-level semantics of the source code. We also plan to localize methods and classes instead of buggy files for automatic program repair.

Acknowledgment. The work described in this paper was partially supported by the National Natural Science Foundation of China (Grant No. 61772093), Fundamental Research Funds for the Central Universities (Grant No. 2019CDYGYB014), and Ling Liu's research is partially supported by USA NSF CISE grant No. 1564097 and an IBM faculty award.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAICPART-MUTATION '07, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society.

[2] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 308–318, New York, NY, USA, 2008. ACM.

[3] A. Bouchet. Greedy algorithm and symmetric matroids. *Mathematical Programming*, 1987.

[4] S. Chakraborty, Y. Li, M. Irvine, R. Saha, and B. Ray. Entropy guided spectrum based bug localization using statistical language model. *CoRR*, abs/1802.06947, 2018.

[5] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, New York, NY, USA, 2005. ACM.

[6] N. Cliff. *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[7] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537, Nov. 2011.

[8] A. Feldman and A. van Gemund. A two-step hierarchical algorithm for model-based diagnosis. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 827–833. AAAI Press, 2006.

[9] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep api learning. 2016.

[10] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, Nov 2012.

[11] G. E. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 2006.

[12] X. Huo, M. Li, and Z.-H. Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 1606–1612. AAAI Press, 2016.

[13] 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*, ASE '05, New York, NY, USA, 2005. ACM.

[14] D. Kim, Y. Tao, S. Kim, and A. Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE Trans. Softw. Eng.*, 39(11):1597–1610, Nov. 2013.

[15] S. Kim, T. Zimmermann, E. J. Whitehead, Jr., and A. Zeller. Predicting faults from cached history. In *Proceedings of the 1st India Software Engineering Conference*, ISEC '08, 2008.

[16] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, pages 218–229, Piscataway, NJ, USA, 2017. IEEE Press.

[17] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr. Does bug prediction support human developers? findings from a google case study. In *Proceedings of the 2013 international conference on Software Engineering*. IEEE Press, 2013.

[18] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Trans. Softw. Eng.*, 32(10):831–848, Oct. 2006.

[19] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: Statistical model-based bug localization. *SIGSOFT Softw. Eng. Notes*, 30(5):286–295, Sept. 2005.

[20] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Inf. Softw. Technol.*, Sept. 2010.

[21] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[22] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *11th Working Conference on Reverse Engineering*, pages 214–223, Nov 2004.

[23] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen. On the use of stack traces to improve text retrieval-based bug localization. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 151–160, Washington, DC, USA, 2014. IEEE Computer Society.

[24] 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 *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 263–272, Washington, DC, USA, 2011. IEEE Computer Society.

[25] P. Plawiak, M. Abdar, and U. R. Acharya. Application of new deep genetic cascade ensemble of svm classifiers to predict the australian credit scoring. *Applied Soft Computing*, 2019.

[26] P. Plawiak, M. Abdar, J. Plawiak, V. Makarenkov, and U. R. Acharya. Dghnl: A new deep genetic hierarchical network of learners for prediction of credit scoring. *Information Sciences*, 2020.

[27] P. Plawiak and U. R. Acharya. Novel deep genetic ensemble of classifiers for arrhythmia detection using ecg signals. *Neural Computing and Applications*, pages 1–25, 2019.

[28] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, and G. Antoniol. Combining probabilistic ranking and latent semantic indexing for feature identification. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, pages 137–148, Washington, DC, USA, 2006. IEEE Computer Society.

[29] S. Rao and A. Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, New York, NY, USA. ACM.

[30] M. Renieres and S. P. Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, Oct 2003.

[31] D. Saha, M. G. Nanda, P. Dhoolia, V. K. Nandivada, V. Sinha, and S. Chandra. Fault localization for data-centric programs. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, New York, NY, USA, 2011. ACM.

[32] 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)*, pages 345–355, Nov 2013.

[33] Salton, Wong, Yang, and S. C. A vector space model for automatic indexing. *Communications of the Acm*, 1974.

[34] 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*, ICML '08, pages 1096–1103, 2008.

[35] E. M. Voorhees. The trec question answering track. *Nat. Lang. Eng.*, 7(4):361–378, Dec. 2001.

[36] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan. Bugram: Bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 708–719, New York, NY, USA, 2016. ACM.

[37] S. Wang and D. Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 53–63, New York, NY, USA, 2014. ACM.

[38] S. Wang and D. Lo. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process*, 10 2016.

[39] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1945.

[40] W. E. WONG and Y. QI. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 2009.

[41] 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*, 2019.

[42] Y. Xiao, J. Keung, Q. Mi, and K. E. Bennin. Improving bug localization with an enhanced convolutional neural network. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 338–347, Dec 2017.

[43] 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*, FSE 2014. ACM.

[44] X. Ye, R. Bunescu, and C. Liu. Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation. *IEEE Transactions on Software Engineering*, 42(4):379–402, April 2016.

[45] z. Yıldırım, P. Pławiak, R.-S. Tan, and U. R. Acharya. Arrhythmia detection using deep convolutional neural network with long duration ecg signals. *Computers in Biology and Medicine*, 2018.

[46] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.

[47] 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*, ISSTA 2017, pages 261–272, New York, NY, USA, 2017. ACM.

[48] X. Zhang, Y. Yao, Y. Wang, F. Xu, and J. Lu. Exploring metadata in bug reports for bug localization. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 328–337, Dec 2017.

[49] 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 *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012.

Bei Wang received the B.S. degree in software engineering from Chongqing University, Chongqing, China, in 2017. He is currently pursuing the Ph.D. degree in the School of Big Data & Software Engineering, Chongqing Univeristy, China. His current research interests include bug localization, deep learning, natural language processing and data mining.

Ling Xu is an Associate Professor at the School of Big Data & Software Engineering, Chongqing Univeristy, China. She received her B.S. degree in Hefei University of Technology in 1998, and her M.S. degree in software engineering in 2004. She received her Ph.D. degree in Computer Application from Chongqing University, P.R. China in 2009. Her research interests include mining software repositories, bug rediction and localization.

Meng Yan is now an Assistant Professor at the School of Big Data & Software Engineering, Chongqing Univeristy, China. Prior to joining Chongqing University, he was a Postdoc at Zhejiang Univeristy advised by Prof.Shanping Li and Dr. Xin Xia. he got his Ph.D degree in June 2017 under the supervision of Prof. Xiaohong Zhang from Chongqing University, China. His currently research foucuses on how to improve developers' productivity, how to improve software quality and how to reduce the effort during software development by analyzing rich software repository data.

Chao Liu received the M.S., B.S., and Ph.D. degrees in software engineering from Chongqing University, Chongqing, China, in 2011, 2014 and 2018, respectively, where he is currently a postdoc at Zhejiang University. His research interests include information retrieval, machine learning, deep learning, data mining, and program analysis.

Ling Liu is a professor in the School of Computer Science, Georgia Institute of Technology. She directs the research programs in the Distributed Data Intensive Systems Lab (DiSL). She is an elected IEEE fellow, a recipient of the IEEE Computer Society Technical Achievement Award in 2012, and a recipient of the best paper award from a dozen top venues, including ICDCS 2003, WWW 2004, 2005 Pat Goldberg Memorial Best Paper Award, IEEE Cloud 2012, IEEE ICWS 2013, ACM/IEEE CCGrid 2015, and IEEE Symposium on BigData 2016. In addition to serving as the general chair and PC chair of numerous IEEE and ACM conferences in data engineering, she has served on the editorial board of over a dozen international journals. Her current research is primarily sponsored by NSF, IBM, and Intel.