Enhancing the Unified Features to Locate Buggy Files by Exploiting the Sequential Nature of Source Code*

Xuan Huo[1] and **Ming Li**[1,2]
[1]National Key Laboratory for Novel Software Technology, Nanjing University
[2]Collaborative Innovation Center of Novel Software Technology and Industrialization
Nanjing 210023, China
{huox, lim}@lamda.nju.edu.cn

Abstract

Bug reports provide an effective way for end-users to disclose potential bugs hidden in a software system, while automatically locating the potential buggy source files according to a bug report remains a great challenge in software maintenance. Many previous approaches represent bug reports and source code from lexical and structural information correlated their relevance by measuring their similarity, and recently a CNN-based model is proposed to learn the unified features for bug localization, which overcomes the difficulty in modeling natural and programming languages with different structural semantics. However, previous studies fail to capture the sequential nature of source code, which carries additional semantics beyond the lexical and structural terms and such information is vital in modeling program functionalities and behaviors. In this paper, we propose a novel model LS-CNN, which enhances the unified features by exploiting the sequential nature of source code. LS-CNN combines CNN and LSTM to extract semantic features for automatically identifying potential buggy source code according to a bug report. Experimental results on widely-used software projects indicate that LS-CNN significantly outperforms the state-of-the-art methods in locating buggy files.

1 Introduction

Software quality assurance is vital to the success of a software system. As software systems become larger and more complex, it is extremely difficult to identify every software defect before its formal release due to the inadequate software testing resources and tight development schedule. Thus, software systems are often plagued with bugs.

To facilitate fast and efficient identification and fixing of the bugs in a released software system, developers often allow users to submit bug reports to bug tracking systems, which are documents written in natural language specifying the situations in which the software fails to behave as it is expected or follow the technical requirements of the system. Bug reports are generated by the end-users of the software and then submitted to the software maintenance team. Once a bug report is received and verified, the software maintenance team would read the textual description of the bug report to locate the buggy potential source files in the source code, and assign appropriate developers to fix the bug accordingly. Unfortunately, for large and evolving software, the maintenance team may receive a large number of bug reports over a period of time and it is costly to manually locate the buggy potential source files based on bug reports.

Therefore, to alleviate the burden of software maintenance team, effective models for identifying potentially buggy source files for a given bug report automatically are highly desirable, and it has drawn significant attentions in software engineering community [Gay *et al.*, 2009; Zhou *et al.*, 2012; Ye *et al.*, 2014; Huo *et al.*, 2016]. The key for identifying buggy source files is to correlate the abnormal program behaviors written in natural languages with the source code written in programming languages that implement the corresponding functionality. Some of the existing methods treat the source code as natural language by representing both bug reports and source files based on bag-of-words feature representations, and correlate their relevance by measuring similarity in the same feature space. For example, [Gay *et al.*, 2009] represented both source code and bug reports using the vector space model (VSM) based on which the similarities between the buggy source files and a bug report are computed for localizing the corresponding buggy files. [Zhou *et al.*, 2012] proposed a revised vector space model (rVSM), where similar historical bug reports whose corresponding buggy files are further exploited to improve the bug localization results obtained by measuring the similarity between bug reports and source files. Recently, Huo *et al.* [2016] considered that the bug reports in natural language and source code in programming language should be processed in different ways. They employed a particular model based on convolutional neural network (CNN) to learn unified features from both bug reports and source code, which are shown effective in modeling source code and improving the performance on locating buggy source files.

Although learning unified features from bug reports and source code by CNN [Huo *et al.*, 2016] is able to overcome the difficulty in modeling natural and program-

*This research was supported by NSFC (61422304, 61272217).

ming languages with different structural semantics, the sequential nature of programming language, especially the long-term dependencies between statements, has not been well-modeled, which causes the loss of semantic information in the source code. The sequence of statements in source code specifies how statements interact with previous ones along the execution path and data stream transmission, and hence provides additional semantics to program functionality aside from lexical and structural terms. An example will make it more concrete: assume that a private string type variable `path` is initialized with a default value `DEFAULT_PATH`, and the following code "`if path==DEAFULT_PATH {path=getNewPath();} File f=File.open(path)`" may result in different behaviors if the previous default value has not been well considered. Although the CNN model proposed in [Huo *et al.*, 2016] employs filters sliding over statements along the execution path is able to reflect the sequential nature in the adjacent statements, it is also very important to consider the sequential nature of statements with long-term dependencies in order to better represent the program functionality and behaviors.

In this paper, we propose a novel unified framework based on deep neural network called LS-CNN (Long Short-term memory based on Convolutional Neural Network) to exploit the sequential nature of source code to enhance the unified features for bug localization. Such a method combines the LSTM and CNN models to enhance the unified features by exploiting the sequential nature of source code, such that the functional semantics of the program and correlations between bug reports and source code for identifying buggy files are carefully embedded. The key part of LS-CNN is the intra-language feature extraction network that combines LSTM and CNN for source code processing, where LSTM is designed to extract semantic features reflecting sequential nature from source code and handle long-term dependency between statements and CNN is designed to capture the local and structure information within statements. Experimental results on widely-used software projects indicate that exploiting sequential features is beneficial for bug localization and the proposed LS-CNN significantly outperforms the state-of-the-art bug localization methods.

The contributions of our work are summarized as follows:

- We propose a novel deep model LS-CNN to enhance the unified features by exploiting the sequential nature of source code for locating buggy source files.
- We design a particular framework to combine LSTM and CNN with respect to the program structure and sequence, which is able to capture the semantics of the program from both structural and sequential perspectives.

The rest of this paper is organized as follows. In Section 2, we discuss some related work. In Section 3, we present our proposed model. The experiments are discussed in Section 4, and finally in Section 5, we conclude the paper and issue some future work.

2 Related Work

To maintain software quality assurance, many bug localization approaches have been studied in recent years. Bug localization, which identifies and locates source files potentially responsible for the bug reported in bug reports, is an extremely vital but costly task in software maintenance. Most existing approaches treat the source code as documents and formalize the bug localization problem as a document retrieval problem, which calculate the relevancy between a bug report and a source file to identify buggy source code [Poshyvanyk *et al.*, 2007; Lukins *et al.*, 2008]. For example, Gay *et al.* [2009] employed Vector Space Model (VSM) based on concept localization to represent bug reports and source code as feature vectors, which are used to measure the similarity between bug reports and source files. Zhou *et al.* [2012] also proposed BugLocator approach using revised Vector Space Model, which is based on document length and similar bugs that have been solved before as new features. Recently, more information and features from bug reports and source code have been investigated for identifying bugs. Saha *et al.* [2013] utilized structured information from source code, such as class and method to enable more accurate bug localization. Wang *et al.* [2014] proposed AmaLgam that combines version history, similar report and structure to further improve bug localization performance.

Recently, deep learning models are very popular and have achieved enormous success in many natural language processing tasks. For example, Johnson and Zhang [2015] utilized convolutional neural network (CNN) to provide an alternative mechanism for effective use of word order for text categorization through direct embedding of small text regions. To overcome the difficulty in learning long term dynamics of Recurrent Neural Networks (RNNs) [Mikolov *et al.*, 2010] for text processing, [Sepp and Juergen, 1977; Graves, 2012; Donahue *et al.*, 2015] proposed long short-term memory (LSTM) by incorporating memory cells to learn when to forget previous states and when to update current states given new information. Furthermore, several studies have tried deep learning models on software engineering research recently [Lam *et al.*, 2015; Mou *et al.*, 2016]. For example, White *et al.* [2015] applied deep models for code suggestion and demonstrated their effectiveness at a real software engineering task compared to state-of-the-art models. To overcome the structural difference between natural and programming languages, Huo *et al.* [2016] designed particular convolutional operations for programming language and proposed a CNN-based model to learn unified features from bug reports and source code for identifying buggy source code.

3 Our Method

The goal of bug localization is to locate the potentially buggy source files that produce the program behaviors specified in a given bug report. Let $\mathcal{C} = \{c_1, c_2, \cdots, c_{n_1}\}$ denote the set of source files of a software project and $\mathcal{R} = \{r_1, r_2, \cdots, r_{n_2}\}$ denotes the collection of bug reports received by the software maintenance team, where n_1, n_2 denote the number of source files and bug reports, respectively. The learning task of identifying buggy source files aims to learn a prediction function

$f : \mathcal{R} \times \mathcal{C} \mapsto \mathcal{Y}$. $y_{ij} \in \mathcal{Y} = \{+1, -1\}$ indicates whether a source code $c_j \in \mathcal{C}$ is relevant to a bug report $r_i \in \mathcal{R}$, which can be obtained by mining software commit logs and bug report descriptions. We instantiate the learning task by proposing a novel deep model named LS-CNN (Long Short-term memory based Convolutional Neural Network), taking the raw data of bug reports and source code as inputs and learns a unified feature mapping $\phi(\cdot, \cdot)$ for a given r_i and c_j by combining LSTM and CNN, based on which the prediction can be made with a subsequent output layer. We will introduce the general framework of the LS-CNN model and explain the way to exploit the sequential nature of source code in the following subsections.

3.1 The General Framework of LS-CNN

The general framework of LS-CNN is shown in Figure 1. Specifically, LS-CNN consists of four parts: input layer, intra-language feature extraction layers, cross-language feature fusion layers and output layers. During the training process, pairs of source code and bug reports and their relevant labels are fed into the network, and the model is trained iteratively to optimize the training loss. For testing process, a new bug report containing bugs and its candidate source code are fed into the model, which outputs their relevant scores indicating which code have high relevance to the given bug report and are located as buggy.

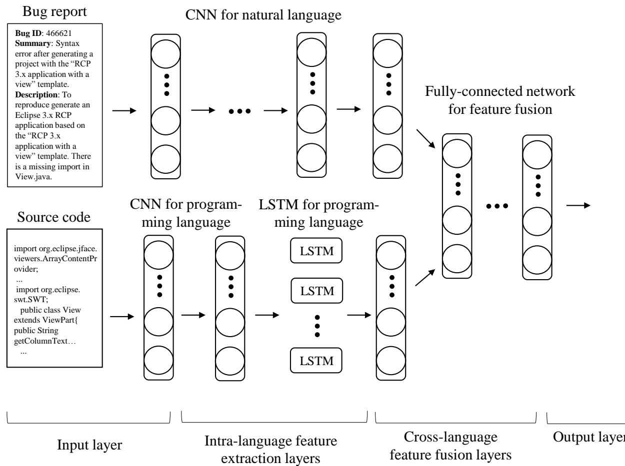

Figure 1: The general framework of LS-CNN.

The source code and bug reports are firstly encoded as feature vectors to feed into the network. We utilize one-hot encoding to represent each word as a k-dimensional one-hot vector, while k indicates the vocabulary size. After the encoding process in the input layer, a sentence of n words can be represented by vectors $X \in \mathbb{R}^{n \times k}$, which are then fed into subsequent layers. Such encoding directly transforms textual data to the raw binary representation with no requirement on domain knowledge for data representation and is shown to be effective in processing text data [Johnson and Zhang, 2015].

After the preprocessing of the input layer, the encoded data X_i^r of a bug report r_i and X_j^c of a source code c_j are then passed to intra-language feature extraction layers. In these layers, since bug reports in natural language and source code in programming language have different structures, we use different network for feature extraction. The intra-language network for natural language processing has been widely studied [Kim, 2014], we follow the standard approach to extract semantic features $\mathbf{h}^r$ from bug reports. Besides, we design a particular network that combines CNN and LSTM to extract the structural as well as sequential features $\mathbf{h}^c$ from source code, which is the key part of our model and will be introduced in the following subsection.

Then, the intra-language features from bug report and source code are further fused into a unified feature representation by the cross-language feature fusion layers, followed by a linear output layer mapping the unified features to $\mathcal{Y}$ which indicates whether source code c_j is related to bug report r_i.

3.2 Exploiting the Sequential Nature of Source Code

In this subsection, we introduce the structure of intra-language feature extraction network for source code, where a novel network structure is designed to combine CNN and LSTM and is able to extract sequential as well as structural information from source code.

To process source code and bug reports for bug localization, Huo *et al.* [2016] proposed a NP-CNN framework to learn unified features and designed a particular CNN network for source code processing, which preserves statement integrity and the structural information between statements could be captured. However, one important point to extract semantic features from the source code is that, the statements in programming language contain sequential nature, which means the previous statements may affect subsequent statements according to the execution path and statements may have long-term dependency via data stream transmission. Alhough CNN applies filters to slide over statements via execution path reflecting the sequential nature of statements to some extent, the sequential nature of source code, especially between statements that have long-term dependency relevance, has not been well considered by CNN. Thus, to represent the program functionality and behaviors better, a richer feature representation which captures sequential semantics should be exploited.

One question arises here: can we enhance the unified feature by considering sequential nature of source code? We extend CNN for source code processing [Huo *et al.*, 2016] by combining Long Short-Term Memory (LSTM) [Sutskever *et al.*, 2014; Zaremba and Sutskever, 2014] to exploit sequential features for bug localization, which has been demonstrated effective for maintaining sequential property in text processing. LSTM is a type of Recurrent Neural Network (RNN) that captures sequential relevance between statements. Note that, LSTM incorporates memory cells $\mathbf{c}$ that allow the network to learn when to forget previous hidden states and when to update hidden states given new information, which can exploit the sequential nature from source code and overcome the difficulty in learning long-term dynamics.

The structure of programming language feature extraction network is illustrated in Figure 2. The first convolutional layer aims to extract semantic features within statements to

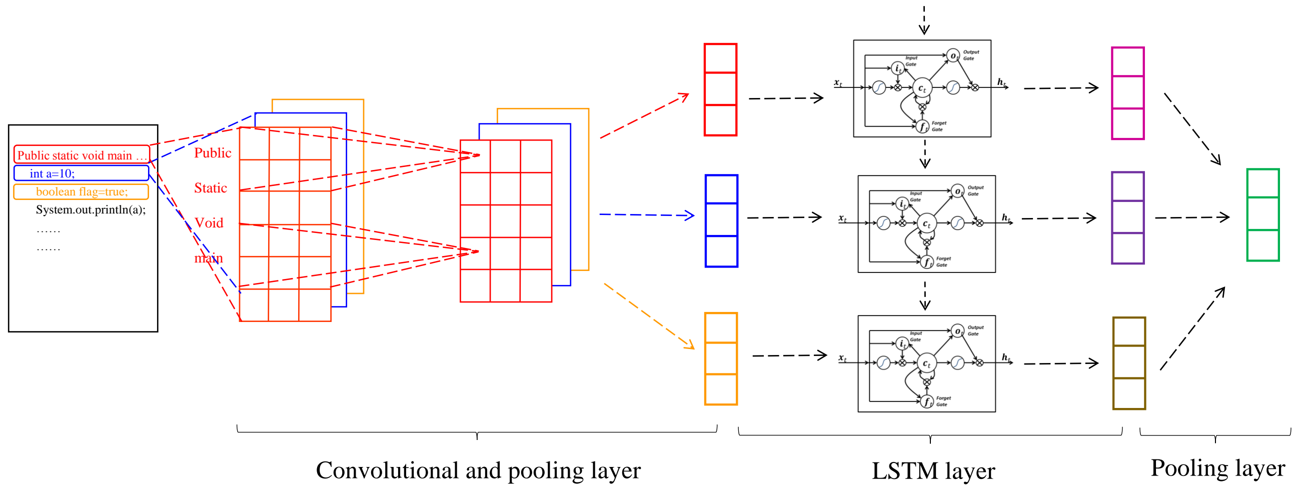

Figure 2: The overall structure of intra-language feature extraction network for programming language. The first convolutional and pooling layer aims to represent the semantics within statements while preserving their statements integrity. The subsequent LSTM layer is used to model sequential interactions between statements with respect to the program structure.

protect the integrity of programming language, which employs multiple filters sliding within statement and converts statements into new feature vectors. Applying *m* filters on a statement of n words will generate m feature maps $\mathbf{z}_t \in \mathbb{R}^{(n-d+1)}$, where d is the window size of filters. In order to extract high-level features with different granularity, the filters in the convolutional layer have different sizes. A max pooling operation is applied within each statement to extract the most informative of each feature map. After convolutional and pooling process, the network generates T feature maps $\mathbf{x} \in \mathbb{R}^m$, where T is the number of statements from a source file. Since the filters in the convolutional layer slide within statements and will not cross between statements, the most informative features of statements are extracted and the statements integrity is also well-preserved.

After processed by convolutional and pooling operations, the feature maps are then fed into the LSTM network. LSTM is a recurrent neural network (RNN) that takes words in a sequence one by one; i.e., at time t, it takes the t-th word as input. In our model, $\mathbf{x}_t \in \mathbb{R}^m$ in the input vectors in time step t, represents the feature maps of t-th statements generated by CNN. Therefore, the input vectors maintain the inherent sequential nature, which can be fed into LSTM that is specified for sequential inputs. The states of LSTM are updated given inputs $\mathbf{x}_t, \mathbf{h}_{t-1}$ and $\mathbf{c}_{t-1}$ as follows:

$$
\begin{aligned}
\mathbf{i}_t &= \sigma(W_{xi}\mathbf{x}_t + W_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i) \\
\mathbf{f}_t &= \sigma(W_{xf}\mathbf{x}_t + W_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh(W_{xc}\mathbf{x}_t + W_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c) \\
\mathbf{o}_t &= \sigma(W_{xo}\mathbf{x}_t + W_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o) \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)
\end{aligned}
\tag{1}
$$

where $\mathbf{h}_t$ denotes the output vector at the time step t, σ denotes the sigmoid function and $\odot$ denotes element-wise multiplication. LSTM overcomes the difficulty in learning long-term dynamics by incorporating memory cells $\mathbf{c}$ that allow the network to learn when to forget previous hidden states and when to update hidden states given new information, which is able to exploit sequential nature from source code to enrich the high-level semantic features.

A pooling layer that involves a mean pooling operation is connected to LSTM, which aims to fuse the outputs $\mathbf{h}_t$ from each time step. We generate the semantic feature $\mathbf{h}_i^c$ of source code c_i by averaging $\mathbf{h}_t$ from each time step: $\mathbf{h}_i^c = (\sum_{t=1}^T \pi_{it}\mathbf{h}_{it}^c) / \sum_{t=1}^T \pi_{it}$, where $\pi_{it} \in \{1, 0\}$ is the indicator variable, $\pi_{it} = 1$ if statement t of source code c_i is present in time step t, and $\pi_{it} = 0$ otherwise. The intra-language features $\mathbf{h}_i^c$ are then fed into the cross-language feature fusion layer for further fusion.

After processing from intra-language feature extraction layers, the generated middle-level intra-language features $\mathbf{h}^r$ from bug reports and $\mathbf{h}^c$ from source code are then fed into cross-language feature fusion layers, where a fully-connected network is employed for learning a unified features and followed by an output layer mapping to the predictions $\mathcal{Y}$. However, a reported bug may only relevant to one or a few source code, while a large number of source code are irrelevant and this imbalance nature should be considered. Similar to [Huo *et al.*, 2016] which introduced an unequal misclassification cost to handle imbalance problem, we randomly drop some negative instances in the cross-language feature fusion layer, which can decrease the computational cost and counteract the negative influence of the imbalance nature. Let $y_i^{(k)}$ denote the k-th label of instance $\mathbf{x}_i$ and $\tilde{y}_i^{(k)}$ denote its prediction, similar to traditional LSTM model, we use cross entropy error function in the output layer, and the parameters are learned by minimizing the following loss function using stochastic gradient descent (SGD) method:

$$
\mathcal{L} = -\sum_{i=1}^{N}\sum_{k=1}^{K}(y_i^{(k)} \ln \tilde{y}_i^{(k)} + (1 - y_i^{(k)}) \ln(1 - \tilde{y}_i^{(k)})) \tag{2}
$$

4 Experiments

To evaluate the effectiveness of LS-CNN, we conduct experiments on open source software projects and compare it with several state-of-the-art models for bug localization.

4.1 Experiment Settings

The datasets used in the experiments are extracted from four well-known open source software projects. The detailed information of the data sets are shown in Table 1.

Table 1: Statistics of data sets.

Data Sets	# source file	# bug reports	# total matches
AspectJ	6,994	1,653	2,812
PDE	5,340	4,034	18,116
Platform	6,112	6,330	26,231
JDT	6,842	5,532	31,556

All the projects and ground truth of bug reports and software code can be extracted from bug tracking system (Bugzilla) and version control system (Git) [Fischer *et al.*, 2003], which have been widely studied in previous work [Zhou *et al.*, 2012; Saha *et al.*, 2013]. To enrich the data sets and provide convincing experiments, we use multiple versions of each project for evaluation. *AspectJ*[1] is an aspect-oriented extension to the Java programming language and the data set PDE[2] (Plug-in Development Environment) is a tool to create and deploy features and plug-ins of Eclipse. Another project is *Platform*[3] that contains a set of frameworks and common services which make up Eclipse infrastructures. We also investigate project *JDT*[4] (Java Development Tools), which is an Eclipse project used for plug-ins support and development of any Java applications.

For each data set, 10-fold cross validation is repeated 10 times and we use Top k Rank (k=10) and AUC (Area Under ROC Curve) to measure the effectiveness of the LS-CNN model, which have been widely applied for evaluation in information retrieval based bug localization problems [Zhou *et al.*, 2012; Saha *et al.*, 2013; Kochhar *et al.*, 2016]. Besides, MAP (Mean Average Precision) are also used for evaluating the cost-effectiveness of identifying buggy source files, which is shown in Eq. (3):

$$\text{MAP} = \sum_{i=1}^{n_2} \sum_{j=1}^{n_1} \frac{\text{Prec}(j) * t(j)}{N_i}$$
$$\text{Prec}(j) = \frac{Q(j)}{j} \tag{3}$$

where n_1, n_2 indicate the number of candidate source code and retrieved bug reports, respectively. N_i is the number of relevant files to report i and $t(j)$ is the indicator vector presenting whether source code in the position j is relevant or not. Prec(j) is the precision at the given cut-off j and $Q(j)$ is the number of relevant source code in top j positions.

We compare our proposed model LS-CNN with following baseline methods:

- BugLocator [Zhou *et al.*, 2012]: a state-of-the-art bug localization method which employs the revised Vector Space model to measure the similarity between bug reports to locate potential buggy files related to a given bug report.
- BLUiR [Saha *et al.*, 2013]: a state-of-the-art bug localization model, which utilizes structured code information from bug reports and source files to enable more accurate bug localization.
- AmaLgam [Wang and Lo, 2014]: a state-of-the-art bug localization model which combines version history, similar bug reports and structure information for mining buggy source code.
- CNN [Kim, 2014]: a straight-forward CNN-based approach which merges textual bug report and source code together and feeds them directly to a CNN model.
- NP-CNN [Huo *et al.*, 2016]: a state-of-the-art CNN-based bug localization model, which employs two different CNNs to learn unified features from source code and bug reports for locating buggy source files.
- LSTM: a straight-forward LSTM based approach which merges textual bug reports and textual source code together and feeds them directly into a LSTM model.
- LSTM+: a variant of LS-CNN, which employs a straight-forward LSTM model for source code processing and a CNN model is for bug reports processing.

For BugLocator, BLUiR and AmaLgam, we use the same parameter settings suggested in their work [Zhou *et al.*, 2012; Saha *et al.*, 2013; Wang and Lo, 2014]. For LS-CNN, we follow the same experiment settings in [Huo *et al.*, 2016] to construct CNN model. We employ the most commonly used ReLU $\sigma(x) = \max(0, x)$ as active function and the filter windows size d is set as 3, 4, 5, with 100 feature maps each in CNN. The number of neuron dimension in LSTM is set the same as CNN. In addition, the drop-out method [Hinton *et al.*, 2012] is also applied in LS-CNN which is used to prevent co-adaption of hidden units by randomly dropping out values in fully-connected layers, and the drop-out probability p is set 0.5 in our experiments.

4.2 Experimental Results

For each data set, 10-fold cross validation is repeated 10 times and the average performance of all compared methods with respect to Top 10 Rank and MAP are tabulated in Table 2 and Table 3, and the performance with respect to AUC is depicted in Figure 3, where the best performance on each data set is boldfaced. Mann-Whitney U-test is conducted at 95% confidence level to evaluate the significance. If LS-CNN significantly outperforms a compared method, the inferior performance of the compared method would be marked with “$\circ$”, and “$\bullet$” if LS-CNN performs significantly worse.

[1]http://www.eclipse.org/aspectj/index.php
[2]http://www.eclipse.org/pde/
[3]http://projects.eclipse.org/projects/eclipse.platform
[4]http://http://www.eclipse.org/jdt/

From the results, we can find that LS-CNN achieves the best average performance in terms of Top 10 Rank. The Top 10 Rank of LS-CNN achieves 0.869, which improves state-of-the-art bug localization methods BugLocator (0.690) by 25.9%, BLUiR (0.709) by 22.5% and AmaLgam (0.729) by 19.2%. It is reasonable that LS-CNN performs better than traditional baseline models (BugLocator, BLUiR and AmaLgam) because LS-CNN costs more time and space to extract features during training process. Comparing with the best deep learning model NP-CNN, LS-CNN still improves the performance of NP-CNN (0.843) by 3.00%, which means the sequential nature between statements from source code can be well exploited by the LSTM network, leading to a better bug localization performance. Additionally, although LSTM+ also utilizes LSTM in source code processing, we find LSTM+ does not perform well in comparison to LS-CNN. The reason may be that the sequential nature within statements may not be as important as those between statements when modeling source code, so that LS-CNN captures the information within statements by CNN can exploit more local semantics, leading to a better feature representation and bug localization performance.

Table 2: Performance Comparisons (Top 10 Rank) of all methods. The best performance of each data sets is boldfaced. The value that significantly worth than LS-CNN is marked with ∘.

Method	*AspectJ*	*PDE*	*Platform*	*JDT*	*Avg.*
BugLocator	0.622 ∘	0.675 ∘	0.727 ∘	0.736 ∘	0.690
BLUiR	0.659 ∘	0.679 ∘	0.754 ∘	0.745 ∘	0.709
AmaLgam	0.693 ∘	0.691 ∘	0.773 ∘	0.757 ∘	0.729
CNN	0.773 ∘	0.752 ∘	0.821 ∘	0.859 ∘	0.801
NP-CNN	0.836 ∘	0.783	0.872 ∘	0.882 ∘	0.843
LSTM	0.792 ∘	0.761 ∘	0.869 ∘	0.868 ∘	0.823
LSTM+	0.855	0.780	0.870 ∘	0.883 ∘	0.847
LS-CNN	**0.869**	**0.793**	**0.895**	**0.917**	**0.869**

Table 3: Performance Comparisons (MAP) of all methods. The best performance of each data sets is boldfaced. The value significantly worth than LS-CNN is marked with ∘.

Method	*AspectJ*	*PDE*	*Platform*	*JDT*	*Avg.*
BugLocator	0.416 ∘	0.367 ∘	0.422 ∘	0.441 ∘	0.412
BLUiR	0.431 ∘	0.391 ∘	0.441 ∘	0.451 ∘	0.429
AmaLgam	0.428 ∘	0.435 ∘	0.449 ∘	0.458 ∘	0.443
CNN	0.512 ∘	0.463 ∘	0.493 ∘	0.517 ∘	0.496
NP-CNN	0.545 ∘	0.497	0.541 ∘	0.533 ∘	0.529
LSTM	0.519 ∘	0.464 ∘	0.521 ∘	0.524 ∘	0.507
LSTM+	0.548 ∘	0.502	0.542 ∘	0.546 ∘	0.535
LS-CNN	**0.568**	**0.503**	**0.568**	**0.583**	**0.556**

The performance comparisons in terms of MAP are similar to Top 10 Rank, as shown in Table 3. MAP evaluates the average performance across all data sets and has been widely used in information retrieval based bug localization problems [Zhou *et al.*, 2012; Saha *et al.*, 2013]. For baseline methods, AmaLgam achieves the best performance among baseline models and the reason may because AmaLgam considers more information such as structure and version history for bug localization. It can be clearly observed that LS-CNN receives the best MAP value (0.556 in average) across all data sets and shows significant improvement in most data sets, indicating the superiority of LS-CNN over the other compared methods is statistically significant.

We also evaluate the performance comparisons between LS-CNN and state-of-the-art bug localization models in terms of AUC, illustrating in Figure 3. AUC measures the comprehensive performance of different predictors and LS-CNN still performs the best on all data sets comparing to state-of-the-art bug localization models.

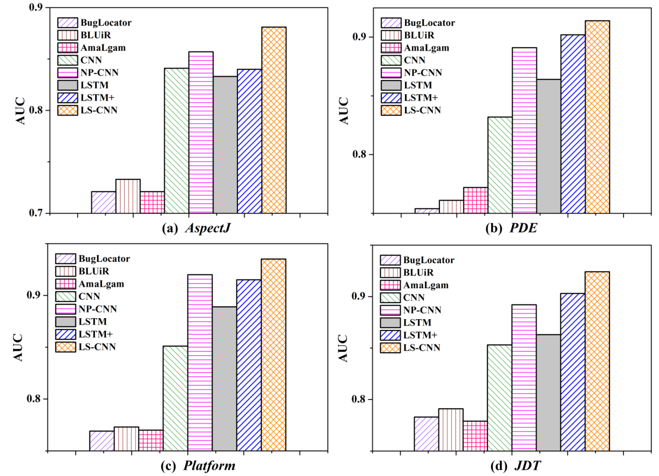

Figure 3: Performance Comparisons (AUC) of all methods on all data sets.

In summary, the sequential nature from source code in programming language is able to provide a better semantic representation, and the specific model LS-CNN that combines CNN and LSTM can exploit the sequential nature as well as the structural information of source code and further improve the bug localization performance.

5 Conclusion

In this paper, we propose a novel deep model called LS-CNN to enhance the unified features for bug localization by exploiting the sequential nature of source code, where a particular network structure is designed to combine CNN and LSTM that captures sequential semantics as well as structural information from source code. Experimental results on widely-used software projects indicate that enhancing the unified feature by exploiting the program sequential information from source code is beneficial and effective, where LS-CNN significantly outperforms the state-of-the-art methods for locating buggy source code.

In the future, incorporating additional data to enrich the network structure of LS-CNN will be carefully investigated. Furthermore, improving bug localization by combining richer program structural and sequential information derived from program analysis tools for feature extraction will be another interesting future work.

References

[Donahue *et al.*, 2015] Jeff Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Trevor Darrell, and Kate Saenko. Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 2625–2634, Boston, MA, USA, 2015.

[Fischer *et al.*, 2003] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the 19th International Conference on Software Maintenance*, pages 23–32, Amsterdam, The Netherlands, 2003.

[Gay *et al.*, 2009] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. On the use of relevance feedback in ir-based concept location. In *Proceedings of the 25th International Conference on Software Maintenance*, pages 351–360, Edmonton, Canada, 2009.

[Graves, 2012] Alex Graves. Supervised sequence labelling with recurrent neural networks. *Studies in Computational Intelligence*, 385:1–131, 2012.

[Hinton *et al.*, 2012] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580*, 2012.

[Huo *et al.*, 2016] Xuan Huo, Ming Li, and Zhi-Hua Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pages 1606–1612, New York, NY, USA, 2016.

[Johnson and Zhang, 2015] Rie Johnson and Tong Zhang. Effective use of word order for text categorization with convolutional neural networks. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 103–112, Denver, CO, USA, 2015.

[Kim, 2014] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of Conference on Empirical Methods in Natural Language Processing*, pages 1746–1751, Doha, Qatar, 2014.

[Kochhar *et al.*, 2016] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 165–176, 2016.

[Lam *et al.*, 2015] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports. In *Proceedings of the 28th International Conference on Automated software Engineering*, pages 476–481, Lincoln, NE, USA, 2015.

[Lukins *et al.*, 2008] Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn. Source code retrieval for bug localization using latent dirichlet allocation. In *Proceedings of 15th Working Conference on Reverse Engineering*, pages 155–164, Antwerp, Belgium, 2008.

[Mikolov *et al.*, 2010] Tomas Mikolov, Martin Karafit, Luks Burget, Jan Cernocky, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association*, pages 1045–1048, Makuhari, Japan, 2010.

[Mou *et al.*, 2016] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 13th AAAI Conference on Artificial Intelligence*, pages 1287–1293, Phoenix, AZ, USA, 2016.

[Poshyvanyk *et al.*, 2007] Denys Poshyvanyk, Yann-Gaël Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Vá Clav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[Saha *et al.*, 2013] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. Improving bug localization using structured information retrieval. In *Proceedings of the 28th International Conference on Automated Software Engineering*, pages 345–355, Silicon Valley, CA, USA, 2013.

[Sepp and Juergen, 1977] Hochreiter Sepp and Schmidhuber Juergen. Long short-term memory. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pages 1606–1612, Cambridge, MA, USA, 1977.

[Sutskever *et al.*, 2014] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing*, pages 3104–3112, Montreal, Canada, 2014.

[Wang and Lo, 2014] Shaowei Wang and David Lo. Version history, similar report, and structure: putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 53–63, Hyderabad, India, 2014.

[White *et al.*, 2015] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th IEEE Working Conference on Mining Software Repositories*, Florence, Italy, 2015.

[Ye *et al.*, 2014] Xin Ye, Razvan C. Bunescu, and Chang 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*, pages 689–699, Hong Kong, China, 2014.

[Zaremba and Sutskever, 2014] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *arXiv/1410.4615*, 2014.

[Zhou *et al.*, 2012] Jian Zhou, Hongyu Zhang, and David 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*, pages 14–24, Zurich, Switzerland, 2012.