Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code*

Xuan Huo and **Ming Li** and **Zhi-Hua Zhou**
National Key Laboratory for Novel Software Technology, Nanjing University
Collaborative Innovation Center of Novel Software Technology and Industrialization
Nanjing 210023, China
{huox, lim, zhouzh}@lamda.nju.edu.com

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 code according to a bug report remains a great challenge in software maintenance. Many previous studies treated the source code as natural language by representing both the bug report and source code based on bag-of-words feature representations, and correlate the bug report and source code by measuring similarity in the same lexical feature space. However, these approaches fail to consider the structure information of source code which carries additional semantics beyond the lexical terms. Such information is important in modeling program functionality. In this paper, we propose a novel convolutional neural network NP-CNN, which leverages both lexical and program structure information to learn unified features from natural language and source code in programming language for automatically locating the potential buggy source code according to bug report. Experimental results on widely-used software projects indicate that NP-CNN significantly outperforms the state-of-the-art methods in locating the buggy source 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 (or bug, informally) before its formal release due to the limited software testing resources and tight development schedule. Thus, software systems are often shipped with bugs.

To facilitate fast and efficient identification and fixing of the bugs in a released software system, *bug reports*, 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 potential buggy source files in the source code, and assign appropriate developer to fix the bug accordingly. However, 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 potential buggy source files based on bug reports.

Bug localization, which aims to alleviate the burden of software maintenance team by automatically locating potentially buggy files in source code bases for a given bug report, has drawn significant attention in software engineering community. To accomplish the goal, the key for bug localization is to correlate the abnormal program behaviors written in natural languages and the source code written in programming languages that implement the corresponding functionality. Most of the state-of-the-art 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 the bug reports and source files by measuring similarity in the same feature space. For example, Lukins *et al.* [2008] apply a generative probabilistic Latent Dirichlet Allocation (LDA) to represent software code and bug reports and locate the buggy files according to their similarities. Gay *et al.* [2009] represent both source files and bug reports using 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, and their experiment results suggest that the VSM model may perform better than the LDA model. Zhou *et al.* [2012] propose 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 simply measuring the similarity between bug reports and source files. Recently, Lam *et al.* [2015] employ autoencoder to learn features that correlate the frequently occurred terms in bug reports and source files in order to enhance the bag-of-words features.

While enjoying the convenience of correlated the heterogeneous data in the same lexical feature space, these methods also suffer from the loss of information when tailoring programming language to natural language by ignoring the program structure. The program structure specifies how

*This research was supported by NSFC (61333014, 61422304, 61272217, 61321491), JiangsuSF(BK20131278) and NCET-13-0275.

different statements interact with each other to accomplishing certain functionality, which provides additional semantics to the program functionality besides the lexical terms. For example, assume a private string type variable `path` initialized with a default value `DEFAULT_PATH`, the following two pieces of code, i.e., "`path = getNewPath(); File f = File.open(path);`" and "`File f = File.open(path); path = getNewPath();`", may result in different program behaviors. Thus, to represent the program functionality better, a richer feature representation which captures both lexical semantics of terms and the program structure needs to be extracted from source code. However, the enrichment of feature for source code lays the bug reports and source files into two different feature spaces, and consequently, increases the difficulties in measuring the correlation between reports and source codes.

One question arises here: can we learn a unified feature representation from both natural and programming language, where the semantics in lexicon and program structure are captured and the correlations between bug reports and source files for bug localization are carefully embedded. In this paper, we propose a novel convolutional neural network called NP-CNN (Natural language and Programming language Convolutional Neural Network) to learn unified feature from bug report in natural language and source code in programming language. This model mainly consists of two consecutive parts. The first part is the intra-language feature extraction layer, which extracts features based on multiple layers of convolutional neurons using bug reports and source files, respectively, where the convolution operation for source code is particularly designed to reflect the program structure. The second part is the cross-language feature fusion layer, which combines the extracted features from bug reports and source files into a unified representation in the purpose of correctly identifying the related source code given a bug report. Experimental results on widely-used software projects indicate that learning unified feature by respecting the program structure is beneficial and the proposed NP-CNN significantly outperforms the state-of-the-art bug localization methods.

The contributions of our work are in two folds:

- We propose a CNN-based deep neural network to learn unified features from natural language and programming language for locating buggy files.
- We design particular convolution operations with respect to the program structure, which is able to capture semantics of the program from both lexical and program-structural perspectives.

The rest of this paper is organized as follows. In Section 2, we discuss several related work. In Section 3, we present the proposed NP-CNN model. In Section 4, we report the experimental results, and finally in Section 5, we conclude the paper and issue some future works.

2 Related Work

Bug localization, which locates source files potentially responsible for the bugs reported in bug reports, is an important but costly activity in software maintenance. Most existing approaches treated the source files as documents and formalized the bug localization problem as a document retrieval problem. Various models has been constructed to compute the similarity or relevancy between the bug reports and the source files. Many information retrieval based bug localization methods have been proposed. Poshyvanyk *et al.* [2007] proposed a feature location model to mine buggy files based on a Latent Semantic Indexing (LSI) model, which can identify the relationship between reports and terms based on Singular Value Decomposition (SVD). Lukins *et al.* [2008] treated bug reports as a mixture of various topics that spit out words with certain probabilities, so they applied a generative probabilistic Latent Dirichlet Allocation (LDA) model for locating buggy files. Gay *et al.* [2009] employed Vector Space Model (VSM) based on concept localization to represent bug reports and source code files as feature vectors, which is used to measure the similarity between bug reports and source files. Zhou *et al.* [2012] proposed BugLocator approach using revised Vector Space Model (rVSM), which is based on document length and similar bugs that have been resolved before as new features. However, all these models ignore the structural information of software code, which may disclose important semantics of the source files beyond the textual representation of source files.

In natural language processing (NLP), deep learning is applied to learn word vector representation. Collobert *et al.* [2011] presented a multi-layer neural network architecture that can handle a number of NLP tasks, which was determined to avoid task-specific engineering. Kim [2014] conducted a serious of experiments with CNN trained on top of pre-trained word vectors, showing that a simple CNN with little hyper parameter tuning achieves excellent results for sentence classification tasks. Zhang *et al.* [2015] applied temporal ConvNets to various large-scale text understanding tasks, in which ConvNets do not require knowledge of words or knowledge of syntax. Johnson and Zhang [2015] studied CNN on text categorization to exploit the word order of text data for text categorization, which showed that CNN provides an alternative mechanism for effective use of word order.

Recently, deep learning is applied to tackle some software engineering problems. White *et al.* [2015] tried deep learning to induce high-quality model for code suggestion. Mou *et al.* [2016], applied CNN on abstract syntax tree to detect code snippets of certain patterns. Lam *et al.* [2015] combined auto encoder with information retrieval based methods to locate buggy files.

3 Convolutional Neural Networks for Natural and Programming Languages

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 $\mathcal{C} = \{c_1, c_2, \cdots, c_{N_1}\}$ denotes the set of source code 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 are the number of source files and bug reports, respectively. The bug reports and source files can be collected from bug tracking systems (e.g., Bugzilla, Jira, etc.) and history control systems (e.g., CVS, Git, etc.).

Unlike many existing methods [Gay *et al.*, 2009; Zhou *et al.*, 2012] which represented bug reports and source codes in the same lexical feature space and computed similarity to identify their correlation, we formalize the bug localization as a learning task, which attempts 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 file $c_j \in \mathcal{C}$ is related to a bug report $r_i \in \mathcal{R}$, which can be obtained by investigating software commit logs and bug report descriptions [Fischer *et al.*, 2003]. The prediction function f can be learned by minimizing the following objective function

$$\min_f \sum_{i,j} \mathcal{L}(f(r_i, c_j), y_{ij}) + \lambda\Omega(f), \tag{1}$$

where $\mathcal{L}(\cdot,\cdot)$ is the empirical loss and $\Omega(f)$ is a regularization term imposing on the prediction function. The trade-off between $\mathcal{L}(\cdot,\cdot)$ and $\Omega(f)$ is balanced by λ.

We instantiate the learning task by proposing a novel convolutional neural network NP-CNN which takes the raw data of bug reports and source codes as input and learns a unified feature mapping $\varphi(\cdot,\cdot)$ for a given r_i and c_j, based on which the prediction can be made with a subsequent linear output layer.

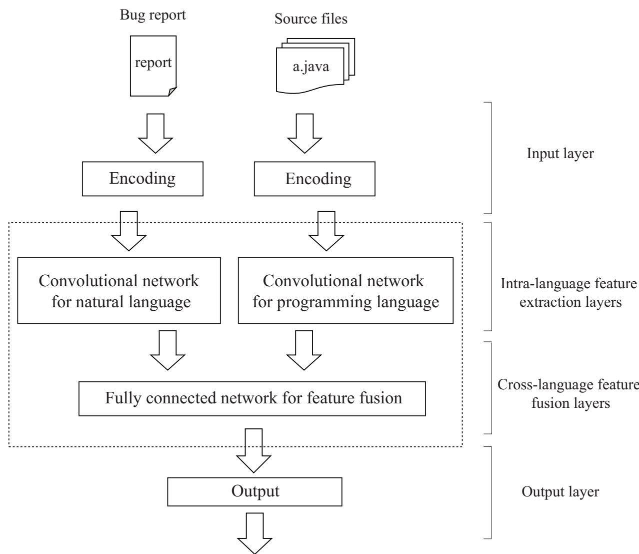

Figure 1: The general framework of NP-CNN.

The framework of NP-CNN is shown in Figure 1. The NP-CNN model contains four parts: input layer, intra-language feature extraction layers, cross-language feature fusion layers, and output layer. In order to feed the raw textual data to convolutional layers for feature learning, bug reports and source codes are firstly encoded in the input layer. To preserve internal and local information, each text term is represented as a k-dimensional one-hot vector, where k is the size of the vocabulary V. For example, suppose the vocabulary V ={"`bug`", "`defective`", "`file`", "`is`", "`this`"}, and the sentence $D =$ "`This file is defective`" can be represented as:

$$\mathbf{x} = [\underbrace{0\,0\,0\,0\,1}_{\text{this}} \mid \underbrace{0\,0\,1\,0\,0}_{\text{file}} \mid \underbrace{0\,0\,0\,1\,0}_{\text{is}} \mid \underbrace{0\,1\,0\,0\,0}_{\text{defective}}]^{\mathrm{T}}$$

Based on the one-hot encoding, a bug report or a source file with n regions of sentences can be represented by $X \in \mathbb{R}^{n\times k}$, which is then fed into subsequent convolutional layers. Such encoding directly transforms textual data to the raw binary representation with no requirement on domain knowledge for data representation. Such representation is shown to be effective in processing textual data [Kim, 2014].

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 file c_j are then passed to intra-language feature extraction layers. In these layers, bug reports and source codes are processed separately by different convolutional networks to extract middle-level intra-language features, where the convolution operations are designed with respect to different characteristics of natural language and programming language. 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 feature to $\mathcal{Y}$ which indicates whether c_j is related to r_i.

The key of the NP-CNN model lies in the intra-language feature extraction layers and cross-language feature fusion layer, which are discussed in detail in the following subsections.

3.1 Intra-language Feature Extraction Layers

Intra-language feature extraction layers employ separate convolutional neural networks to extract intra-language features form natural language and programming language. Since extracting features from natural language using CNN has been widely studied [Johnson and Zhang, 2015], we follow the standard approach to extract features from bug reports. Thus, we focus on building convolutional networks for source code in programming language.

Programming language, although in textual format, differs from natural language mainly in two aspects. First, the basic language component carrying meaningful semantics in natural language is word or term, and the semantics of the natural language can be inferred from a bag of words. By contrast, in programming language the basic language component carrying meaningful semantics is statement, and the semantics of the programming language can be inferred from the semantics on multiple statements plus the way how these statements interact with each other along the execution path. Thus, to extract features from programming language, the convolution operations should explicitly respect to the atomicity of statements in semantics. Second, natural language organizes words in a "flat" way while programming language organizes its statements in a "structured" way to produce richer semantics. For example, a branching structure "`if-then-else`" defines two parallel groups of statements. Each group interacts with the statements before and after the branching block while there is no interaction between the two groups. Thus, to extract features from programming language, the convolution operations should obey the program structure defined by the

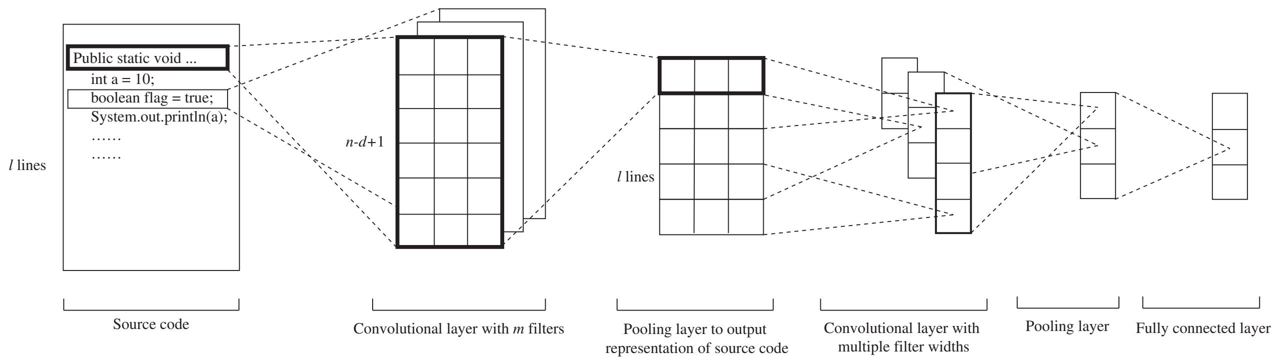

Figure 2: The overall structure of convolutional neural network for programming language. The first convolutional and pooling layer aims to represent the semantics of a statement based on the tokens within the statement, and the subsequent convolution and pooling layer aims to model the semantics conveyed by the interactions between statements with respect to the program structure while preserving the integrity of statements. The fully connected networks are connected to cross-language feature fusion layers.

programming languages.

Based on the aforementioned considerations, we propose the substructure of NP-CNN responsible for extracting features from source code based on convolutional neural network. The network structure is specified in Figure 2. The first convolutional and pooling layer aims to represent the semantics of a statement based on the tokens within the statement, and the subsequent convolution and pooling layers aim to model the semantics conveyed by the interactions between statements with respect to the program structure while preserving the integrity of statements.

Suppose $\mathbf{t}_p \in \mathbb{R}^k$ is a k-dimensional vector corresponding to the p-th token and a window of d tokens is represented as $\mathbf{s}_q \in \mathbb{R}^{dk}$. The first convolutional layer employs a filter $\mathbf{w} \in \mathbb{R}^{dk}$ and a non-linear activation function σ to convert a statement of n words into a new vector $\mathbf{z} \in \mathbb{R}^{n-d+1}$. Since the length of each sentence is different, the extracted features cannot be fed directly to a fixed-size neural layer. Therefore, we fix the number of pooling units and dynamically determine the pooling region size on each data point, which has been shown to be efficient in previous works [Zhang *et al.*, 2015; Johnson and Zhang, 2015]. It is noteworthy that each row of the feature map represents one line of code after the first convolutional and pooling layer, and consequently the integrity of the statements is well-preserved.

The subsequent convolutional and pooling layer aims to model high order interactions between statements in different granularity by varying the size of convolution windows. For example, the first filter operates on the window of features with $d = 2$, which can be viewed that the information between two consecutive statements along the execution path is extracted and represented. The second filter on the window with $d = 3$ is viewed that the filter extracts features from three consecutive statements along the execution path, and so on. To avoid the poor performance caused by using a large window size [Collobert and Weston, 2008; Kim, 2014], we slice the program into different building blocks [Binkley *et al.*, 2014] and set the maximal window size as the average length of program blocks. Moreover, we pad the window locating on the boundary of branches and loops to ensure the interactions between statements do not violate the execution path.

3.2 Cross-language Feature Fusion Layers

In the cross-language feature fusion layers, we employ a fully connected neural network to fuse middle-level features extracted from bug reports and source files to generate a unified feature representation, where the network is learned in order to facilitate the determination on whether the given source code file is related to the given bug report based on the unified feature.

In most cases of bug localization, a reported bug may be only related to one or only a few source code files, while a large number of source code files are irrelevant to the given bug report. Such an imbalance nature increases the difficulty in learning a well-performing prediction function based on the unified feature.

To address this problem, we propose to learn the unified feature that may counteract the negative influence of the imbalanced data in the subsequent learning of prediction function. Inspired by [Zhou and Liu, 2006], we introduce an unequal misclassification cost according to the imbalance ratio and train the fully connected network in a cost-sensitive manner.

Let $cost_n$ denote the cost of incorrectly associating an irrelevant source code file to a bug report and $cost_p$ denote the cost of missing a buggy source code file that is responsible for the reported bugs. The weight of the fully connected networks $\mathbf{w}$ can be learned by minimizing the following objective function based on SGD (stochastic gradient descent).

$$\min_{\mathbf{w}} \sum_{i,j} [cost_n L(\mathbf{z}_i^r, \mathbf{z}_j^c, y_{ij}; \mathbf{w})(1 - y_{ij}) \\ + cost_p L(\mathbf{z}_i^r, \mathbf{z}_i^c, y_{ij}; \mathbf{w})(1 + y_{ij})] + \lambda ||\mathbf{w}||^2 \tag{2}$$

where L is the loss function and λ is the trade-off parameter.

4 Experiments

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

4.1 Experiment Settings

The data sets used in the experiments are extracted from four well-known open source software projects and the statistics are shown in Table 1.

The data set *JDT* (Java Development Tools) [1] is an Eclipse project used for plug-ins support and development of any Java applications. The project *PF* (Eclipse Platform) [2] contains set of frameworks and common services that make up Eclipse infrastructures. Another project *PDE* (Plug-in Development Environment) [3] is a tool to create and deploy features and plug-ins of Eclipse. We also investigate the *AspectJ* [4] project to evaluate the performance, which is an aspect-oriented extension to the Java programming language. All the projects and labels of software code and bug reports can be extracted from bug tracking system and the CSV/Git version control system, which have been widely used in previous studies [Zhou *et al.*, 2012; Lam *et al.*, 2015].

Table 1: Statistics of our data sets.

Data sets	# fixed bug reports	# source files	# avg. buggy files per report
JDT	12,826	2,272	4.39
PF	14,893	1,012	6.79
PDE	4,034	2,970	8.34
AspectJ	1,734	1,136	1.73

As indicated by Table 1, the number of candidate source files is large, but the data sets are highly imbalanced in that only a few source files are related to a given bug report. Therefore, we use AUC, which has been widely applied to evaluate the learning performance in imbalanced learning problem. Besides, we also evaluate the performance using MAP (Mean Average Precision) and Top *k* Rank, which are widely used for evaluating the cost-effectiveness of bug localization performance [Zhou *et al.*, 2012; Ye *et al.*, 2014; Lam *et al.*, 2015].

We compare the proposed model NP-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 identify potential buggy files related to a given bug report.
- Two-phase [Kim *et al.*, 2013]: a state-of-the-art bug localization model that firstly uses Naive Bayes to filter uninformative bug reports and then use vector space model to predict buggy files.
- HyLoc [Lam *et al.*, 2015]: a recently proposed bug localization model which employs auto-encoder and vector space model to identify potential buggy files related to a given bug report.
- CNN: a straightforward CNN-based approach which merges textual bug reports and textual source code together and feeds them directly to a CNN.
- US-CNN (CNN with Under Sampling): a variant of NP-CNN, which addresses the imbalanced problem by applying under-sampling to training set in advance to reduce the number of negative pairs of bug reports to irrelevant source files.
- N-CNN (Natural language CNN): a variant of NP-CNN where no program structure is considered in "intra-language feature extraction layers" for source code and source code is processed in the same way as the textual bug reports.

For Buglocator and Two-Phase, we use the same parameter settings suggested in [Zhou *et al.*, 2012; Kim *et al.*, 2013], respectively. For all data sets, we fix the activation function to $\sigma(x) = \max(x, 0)$. We use windows (d) of 2,3,4,5 with 100 feature maps each, and roughly 5,000 words that appear most frequently in the bug reports and software code are used in the experiment. In addition, we use two techniques to improve prediction performance: response normalization [Krizhevsky *et al.*, 2012] and dropout [Hinton *et al.*, 2012]. Response normalization scales the output of the pooling layer $\mathbf{z}$ by multiplying $(1+|\mathbf{z}^2|^{-1/2})$. Another method dropout is used to prevent co-adaptation of hidden units by randomly dropping out values. In our experiment, we set dropout probability p=0.5 in fusion layer.

4.2 Experiment Results

For each data set, 10-fold cross validation is repeated 10 times and the average performance of all the compared methods with respect to AUC and MAP are tabulated in Table 2 and Table 3, respectively, where the best performance on each data set is boldfaced, and the performance with respect to Top *k* Rank is depicted in Figure 3. We conduct Mann-Whitney test at 95% confidence level. If NP-CNN significantly outperforms a compared method, the inferior performance of the compared method would be marked with "∘", and the value that significant better than NP-CNN is marked with "•". The highest value of each data set is marked in bold.

It can be observed from the tables that the proposed NP-CNN achieves the best average performance (0.891) in terms of AUC, which improves BugLocator (0.747) by 19.2%, Two-phase (0.738) by 20.7%, HyLoc (0.807) by 10.4%, and NP-CNN achieves best performance (0.557) with respect to MAP on almost all data sets except for *JDT*. The superiority of the NP-CNN is statistically significant.

Figure 3 also indicates the superiority of NP-CNN over the other compared methods with respect to Top *k* Rank. NP-CNN can achieve an average Top *k* Rank at 0.881, which improves the average value of BugLocator (0.691) by 27.4% , Two-phase (0.600) by 46.8% and HyLoc (0.752) by 17.2%.

The superior performance of NP-CNN over the state-of-the-art bug localization methods indicates that NP-CNN is

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

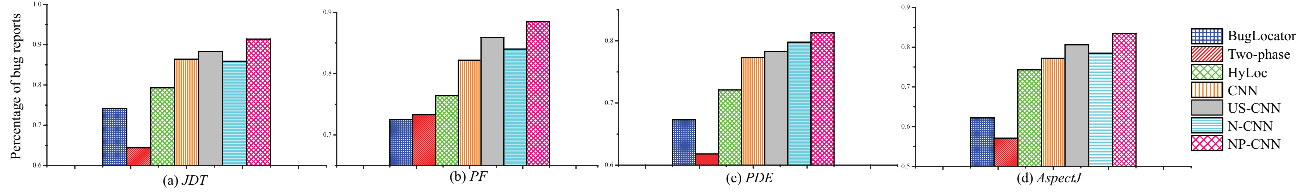

Figure 3: Top 10 Rank of the compared methods on all data sets. The higher the metric value, the better the prediction performance. It can be easily observed that NP-CNN performs the best on all data sets.

Table 2: MAP of the compared methods on all data sets. The highest value of each column is marked in bold. The value that significant worse than NP-CNN is marked with ○. The value that significant better than NP-CNN is marked with •.

Method	*JDT*	*PF*	*PDE*	*AspectJ*	Avg.
BugLocator	.441 ○	.350 ○	.422 ○	.418 ○	.408
Two-phase	.372 ○	.398 ○	.381 ○	.397 ○	.387
HyLoc	.455 ○	.410 ○	.552 ○	.433 ○	.463
CNN	.517 ○	.483 ○	.582 ○	.512 ○	.524
US-CNN	**.527**	.496 ○	.574 ○	.532	.532
N-CNN	.508 ○	.493 ○	.584 ○	.523 ○	.527
NP-CNN	.522	**.537**	**.624**	**.545**	**.557**

Table 3: AUC of the compared methods on all data sets. The highest value of each column is marked in bold. The value that significant worse than NP-CNN is marked with ○. The value that significant better than NP-CNN is marked with •.

Method	*JDT*	*PF*	*PDE*	*AspectJ*	Avg.
BugLocator	.781 ○	.772 ○	.753 ○	.683 ○	.747
Two-phase	.724 ○	.803 ○	.763 ○	.664 ○	.738
HyLoc	.813 ○	.830 ○	.824 ○	.761 ○	.807
CNN	.853	.851 ○	.863 ○	.849	.854
US-CNN	.872 ○	.865 ○	.857 ○	.842 ○	.859
N-CNN	.866 ○	.847 ○	.862 ○	.848	.856
NP-CNN	**.881**	**.913**	**.914**	**.857**	**.891**

effective in learning unified features from natural language and programming language for bug localization. Comparing to previous bug localization models, BugLocator and Two-phase model use vector space model to represent bug reports and source files, respectively. They use cosine similarity between source files and bug reports to identify bugs, but NP-CNN learns unified features of natural and programming language leading to a better representation. The HyLoc model uses auto-encoder to search for the links between text terms and source code terms, which can only learn their shallow and partial relationship. NP-CNN uses deep convolutional network structure, which can extracts more complete and semantic features.

We further evaluate the effectiveness of intra-language feature extraction layers and cross-language feature fusion layers, which are the key parts of the NP-CNN model. It can be observed from Table 2 and Table 3 that NP-CNN outperforms N-CNN 3.7% in terms of AUC and 5.6% in terms of MAP, which suggests our convolutional neural network for programming language can extract better inner features from source code than natural language network.

In addition, to evaluate the effectiveness of cost-sensitive cross-language fusion layer, we use US-CNN for comparison, a variant implementation of NP-CNN which first uses undersampling operation on training data sets to discard negative pairs until the number equal to the positive ones. The sampling procedure repeats for 10 times and the results are ensembled at last. It can be clearly observed from Table 2 and Table 3 that NP-CNN performs better than US-CNN on all data sets in terms of MAP and AUC.

In summary, the experimental results suggest that NP-CNN can learn unified feature representation from natural and programming languages to facilitate better bug localization.

5 Conclusion

In this paper, we propose a novel convolutional neural network called NP-CNN to learn unified features from natural language and source code in programming language for bug localization problems, where particular convolution operations that reflect the program structure are carefully designed to generate features that capture semantics from both lexicon and program structure. Experimental results on widely-used software projects indicate that learning unified feature by respecting to the program structure is beneficial and the proposed NP-CNN significantly outperforms the state-of-the-art bug localization methods.

NP-CNN exploits the program structure by explicitly modeling the high-order interactions between statements. Combining richer program structure information derived from program analysis tools for extracting features from programming languages will be investigated in future. Moreover, incorporating additional data to enrich the structure of NP-CNN is also another interesting future work.

References

[Binkley *et al.*, 2014] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. Orbs: Language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120, Hong Kong, China, 2014.

[Collobert and Weston, 2008] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167, Helsinki, Finland, 2008.

[Collobert *et al.*, 2011] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 12:2493–2537, 2011.

[Fischer *et al.*, 2003] M. Fischer, M. Pinzger, and H. 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] G. Gay, S. Haiduc, A. Marcus, and T. 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.

[Hinton *et al.*, 2012] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580*, 2012.

[Johnson and Zhang, 2015] R. Johnson and T. 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 *et al.*, 2013] D. Kim, A. Zeller, Y. Tao, and S. Kim. Where should we fix this bug? a two-phase recommendation model. *IEEE Transactions on Software Engineering*, 99(1):1, 2013.

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

[Krizhevsky *et al.*, 2012] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, Lake Tahoe, NV, USA, 2012.

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

[Lukins *et al.*, 2008] S. K. Lukins, N. Kraft, L. H. Etzkorn, et al. 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.

[Mou *et al.*, 2016] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 13th AAAI Conference on Artificial Intelligence*, Phoenix, AZ, USA, 2016.

[Poshyvanyk *et al.*, 2007] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. C. 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.

[White *et al.*, 2015] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th IEEE Working Conference on Mining Software Repositories*, pages 334–345, Florence, Italy, 2015.

[Ye *et al.*, 2014] 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*, pages 689–699, Hong Kong, China, 2014.

[Zhang *et al.*, 2015] X. Zhang, J. Zhao, and Y. LeCun. Character-level convolutional networks for text classification. In *Advances in Neural Information Processing Systems, Montreal, Canada*, pages 649–657, Montreal, Canada, 2015.

[Zhou and Liu, 2006] Z.-H. Zhou and X.-Y. Liu. Training cost-sensitive neural networks with methods addressing the class imbalance problem. *IEEE Transactions on Knowledge and Data Engineering*, 18(1):63–77, 2006.

[Zhou *et al.*, 2012] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, pages 14–24, Zurich, Switzerland, 2012.