Deep Transfer Bug Localization

ABSTRACT

ACM Reference Format:
. 2017. Deep Transfer Bug Localization. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 10 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

An active software project often receives numerous bug reports daily [1]. To resolve each report, developers often need to spend much time and effort [19]. One main task that developers need to do during debugging is to identify code that needs to be fixed to resolve the bug. This task, often referred to as *bug localization*, is a non-trivial one as relevant files need to be identified out of a collection of hundreds or even thousands of files.

To help developers in locating bugs, various automated solutions have been proposed [5, 6, 12, 16, 17]. Many of them analyze the description of a bug report to identify source code files relevant to it [5, 12, 16, 17]. These text-based solutions can be further divided into two families: unsupervised and supervised. Unsupervised solutions, which were historically proposed first, typically employ information retrieval techniques to identify files that contain many of the words that appear in the bug report [12, 16, 17]. More recently, supervised approaches are introduced [5]. These approaches use a collection of bug reports and their relevant source code files as training data. This data is then used to learn a good model that can map new bug reports to their respective relevant source code files. Supervised approaches have been shown to be superior than unsupervised ones.

One limitation of a supervised approach is the need for sufficient good quality training data. Insufficient or low quality data can be detrimental to its effectiveness. This problem is particularly important when a bug localization approach needs to be applied to new projects with limited bug fixing history. Unfortunately, this issue, often referred to as the *cold-start problem*, has not been explored much by past supervised bug localization studies.

To address the above mentioned limitation, in this work we propose *Deep Transfer Bug Localization* (DTBL). DBTL deals with cold-start problem affecting a target software project by adapting data from other projects. DTBL, the first cross-project bug localization solution, combines deep and transfer learning to address the cold-start problem. David says: Please add brief description of the approach.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

There have been a number of transfer learning solutions designed to help with cold-start problem in software engineering context. For example, ... Our approach is unique from previous solutions in the following ways. First, ... David says: Please compare the approach with existing work and highlight its novelty.

We have evaluated our proposed solution on ... The experiment results first highlight the need for DBTL as existing solutions cannot effectively make use of data from other projects to create a model that can accurately locate bug in a target project (given a bug report). ... dlPlease add brief description of the results.

Our contributions are as follows:

(1) We present a new direction of research on cross-project bug localization. We highlight that existing supervised bug localization techniques are not able to perform well when they are are trained using data from other projects.
(2) We propose a deep transfer learning solution that .... David says: Please add very brief description of the approach highlighting its novelty.
(3) We have evaluated our proposed approach on ... The results show that ... David says: Please add very brief description of the results.

The structure of the remainder of this paper is as follows. In Section 2 we summarize the state-of-the-art work on supervised bug localization that our approach builds upon. We elaborate the details of our approach in Section 3. The results of the evaluation of the approach are presented in Section 4. We discuss pertinent points and threats to validity in Section 6. We describe related work in Section 7, before concluding and mentioning future work in Section 8.

2 BACKGROUND

In this section, we give a brief introduction about state-of-the-art supervised bug localization NPCNN (Natural and Programming language Convolutional Neural Network), which was proposed by Huo et al. [5]. Our model is built on NPCNN for cross-project bug localization.

The goal of supervised bug localization is training prediction model using bug reports and source code, and then predicts the localization of buggy code that produces the program behaviors specified in a given bug report. Let $C = \{c_1, c_2, \cdots, c_{n_1}\}$ denotes the set of source code , and $\mathcal{R} = \{r_1, r_2, \cdots, r_{n_2}\}$ denotes the bug reports, where n_1, n_2 denote the number of source files and bug reports from source project and target project, respectively. We formulate cross-project bug localization as a learning task which aims to learn a prediction function $f : \mathcal{R} \times C \mapsto \mathcal{Y}$. $y_{ij} \in \mathcal{Y} = \{+1, -1\}$ indicates whether a source code $c_j \in C$ is relevant to a bug report $r_i \in \mathcal{R}$.

Noticing that the semantics of bug reports in natural language and source code in programming language is different, so the NPCNN model employs different convolutional neural networks to semantic features from bug reports and source code, separately. The general structure of NPCNN is illustrated in fig. 1. The bug reports and source code are firstly encoded as feature vectors by one-hot algorithm to feed into the CNNs. In the intra-language feature extraction layers, two CNNs are employed for semantic feature extraction: CNN for natural language is followed by the standard approach [7], and CNN for programming language is specifically designed.

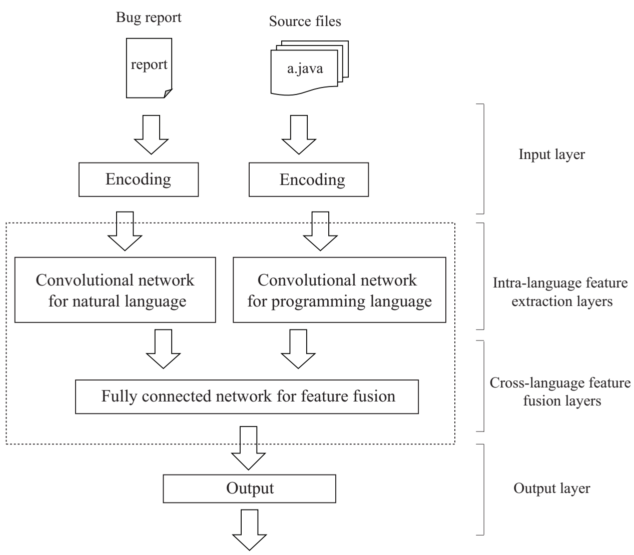

Figure 1: The general structure of Natural and Programming language Convolutional Neural Network.

Huo et al. [5] found that 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 in programming language the basic language component carrying meaningful semantics is statement. Second, natural language organizes words in a "flat" way while programming language organizes its statements in a "structured" way to produce richer semantics. Therefore, the structure of CNN for programming language is specifically designed to solve these two points. The first convolutional and pooling layers extract features within statements while preserving the integrity of statements by sliding convolutional window within statements. The subsequent convolutional and pooling layers extract features between statements reflecting the structural nature by employing different size of convolutional windows. More details can be referred in [5].

After feature extraction, the middle-level features generated from bug reports and source code are fed into the cross-language feature fusion layers. To deal with the imbalance nature of bug localization data, the cross-language feature fusion layers 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 associate 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. Then the weights of the fully connected networks w can be learned by minimizing the following objective function based on SGD (stochastic gradient descent).

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

3 PROPOSED APPROACH

The goal of cross-project bug localization is using data from source project and a few data from target project to locate the potentially buggy source code in the target project that produce the program behaviors specified in a given bug report. Let $C_s = \{c_{s_1}, c_{s_2}, \cdots, c_{s_{nc_1}}\}$ and $C_t = \{c_{t_1}, c_{t_2}, \cdots, c_{t_{nc_2}}\}$ denote the set of source code from source project and target project respectively, $C = C_s \bigcup C_t$. $\mathcal{R}_s = \{r_{s_1}, r_{s_2}, \cdots, r_{s_{nr_1}}\}$ and $\mathcal{R}_t = \{r_{t_1}, r_{t_2}, \cdots, r_{t_{nr_1}}\}$ denotes the bug reports, respectively, $\mathcal{R} = \mathcal{R}_s \bigcup \mathcal{R}_t$, where nc_1, nc_2, nr_1, nr_2 denote the number of source files and bug reports from source project and target project, respectively. We formulate cross-project bug localization as a learning task which aims to learn a prediction function $f : \mathcal{R} \times C \mapsto \mathcal{Y}$. $y_{ij} \in \mathcal{Y} = \{+1, -1\}$ indicates whether a source code $c_j \in C$ is relevant to a bug report $r_i \in \mathcal{R}$.

In this paper, we propose a novel deep transfer neural network named TRANP-CNN (TRAnsfer Natural and Programm Language Convolutional Neural Network) to instantiate the cross-project bug localization problem, which is an extension of NP-CNN proposed by Huo et al. TRANP-CNN takes 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, based on which the prediction can be made with a subsequent output layer. We will introduce the general framework of TRANP-CNN and explain the way to employ deep transfer technique for cross-project bug localization in the following subsections.

3.1 General Structure

The general structure of TRANP-CNN is shown in Figure 2. Specifically, TRANP-CNN consists of four parts: input layer, transferable feature extraction layers, heterogeneous predicting adaptation layers and output layer. The left figure indicates the training process of TRANP-CNN and the right figure indicates the test process. Since the TRANP-CNN model is used to deal with cross-project bug localization tasks, during the training process, pairs of source code and bug reports and ground truth labels from source project are fed into the deep neural network for weight training, as well as very few data from target project, and for testing process, a new bug report from target project 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.

3.2 Transferable Feature Extraction Layers

Before processing in Transferable Feature Extraction Layers, the source code and bug reports should be firstly encoded as feature vectors. Traditional techniques usually employs TFIDF to represent text content, which may lose the word relationships. In our model, to maintain the semantics with structural information of text, we apply word2vec technique to encode bug reports and source code.

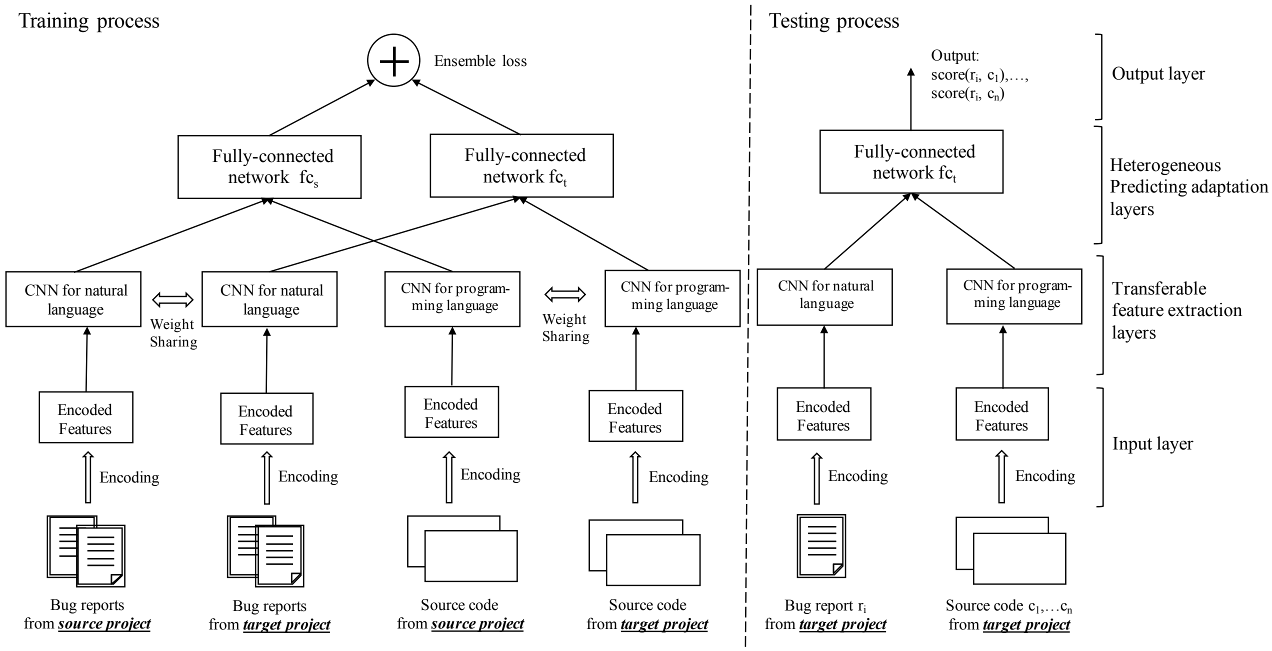

Figure 2: The overall structure of Transfer Natural and Programming language CNN. The left part is the training process of TRANP-CNN based on the bug reports and source code from source projects and a few data from target projects, the weights of which are trained by minimizing the loss of ensemble loss from fully-connected networks fc_s and fc_t. The right part is the testing process, a new bug report and its candidate source code are fed into the model, and TRANP-CNN outputs their relevant scores for bug localization.

To process the bug reports in natural language, we follow the standard convolutional neural network [7] (named as NCNN) to extract semantic features $\mathbf{h}^r$ from bug reports, which has been widely studied. Meanwhile, as mentioned in Huo et al. [5], bug reports and source code should be processed in different ways, because two languages have different structural semantic property. Therefore, we apply programming language specific convolutional neural network (named as PCNN) to extract semantic features from source code. The PCNN is able to preserve the integrity within statements by sliding convolutional windows within statements in the first convolutional and pooling layers, and generates semantic features between statements by applying different sizes of convolutional windows in the subsequent convolutional and pooling layers. The details of PCNN is introduced in the background section and more details can be referred in Huo et al.'s work [5].

Although the data distribution of cross-projects may be different, the semantics within the statements may be similar since the projects are using the same programming language so that the rule of semantic feature extraction is transferable. Therefore, in the training process, the PCNN models in the transferable feature extraction layers for processing source projects data and target projects data share the same parameter weights, which means that the rule to extract semantic features from programming languages are the same. This structure helps generate semantic features from target project using the large number of training source code in the source projects, leading to a better feature representation in the cross-project bug localization.

3.3 Heterogeneous Predicting Adaptation Layers

After processing from transferable feature extraction layers, the high-level semantic features from bug reports and source code are extracted, which would be then fed into a fully-connected neural network for feature fusion. However, in cross-project bug localization, the data distribution of source project and target project is different, which means directly employing the same fully-connected network to fuse features from both source and target projects will have bias, leading to a poor bug localization performance.

One question arises here: can we design a particular network to extract features within project, and fuse feature in separate structure? To address this problem, we design two particular fully-connected networks to combine the middle-level features in Cross-project feature fusion layers: One fully-connected network fc_s is used for source project feature fusion and the other fully-connected work fc_t is used to fuse features in the target project. The structure suggests that the feature extraction of different projects are similar and can be processed in the same Convolutional Neural Network, and the feature fusion and projection process is different so that two separate fully-connected neural network are designed to solve this problem. The objective function in the cross-project feature fusion layers can be rewritten in Eq. (2):

$$\begin{aligned}\arg\min_{\mathbf{W}} \sum_{s_i,s_j} \mathcal{L}(\mathbf{h}_{s_i}, \mathbf{h}_{s_j}, y_{s_{ij}}; W_{fc_s}, W_{conv})) \\ + \sum_{t_i,t_j} \mathcal{L}(\mathbf{h}_{t_i}, \mathbf{h}_{t_j}, y_{t_{ij}}; W_{fc_t}, W_{conv}) + \lambda ||\mathbf{W}||^2\end{aligned} \tag{2}$$

where, $\mathcal{L}$ is the square loss, λ is the trade-off parameter and the weight vectors W contains the weight vectors in convolutional neural networks W_{conv}, in fully-connected network of source domain W_{fc_s} and in fully-connected network of target domain W_{fc_t}. All the weights is learned by minimizing the objective function based on SGD (stochastic gradient descent) in the same time.

4 EXPERIMENTS

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

4.1 Research Questions

Our experiments are designed to address the following research questions:

RQ1*: Is there a need for a specialized technique for cross-project bug localization?*

If a model learned from one project can be used for others project, then there is no need for a specialized technique for cross-project bug localization. Thus, before we consider other research questions, we validate the need for our proposed approach by empirically evaluating the effectiveness of a model learned from one project to localize bugs in other projects.

RQ2*: Does the cross-project feature fusion layer improve the bug localization performance?*

David says: Xuan, please help to motivate this research question. I can't motivate it since I believe the details of the approach is going to change.

In Section 3, we propose to employ heterogeneous predicting layers adaptation that apply two fully-connected networks for prediction from source and target projects separately, which is the key part of TRANP-CNN. In this research question, we evaluate whether the heterogeneous predicting adaptation layers help improve the bug localization performance by comparing the results previous NPCNN and SimpleTrans model.

RQ3*: Can TRANP-CNN outperform other bug localization methods?*

A number of bug localization methods have been proposed in the literature. In this research question, we evaluate whether and to what extent can our proposed approach TRANP-CNN outperforms the state-of-the-art methods designed for bug localization and those that can be adapted for bug localization.

Table 1: Bug Report Dataset

Project	# Reports
HTTPClient	746
Jackrabbit	2,402
Lucene-Java	2,443

4.2 Datasets

We consider three datasets containing a total of 5,591 reports from JIRA issue tracking systems of HTTPClient (H)[1], Jackrabbit (J)[2], and Lucene-Java (L)[3]. HTTPClient is a library for implementing the client side of HTTP standard, while Jackrabbit is a content repository, and Lucene is a text search engine. We only consider projects with JIRA issue tracking systems since links between bug reports and their bug fixing commits stored in them are typically more reliable than those stored in Bugzilla issue tracking systems – c.f., [2]. The details of the reports considered in this study are shown in Table 1 and they have been used before by Kochhar et al. [9].

David says: Xuan, I thought we are also using your previous datasets? If we only use Pavneet's dataset, do we use all of them or only some of them that are not biased? Pavneet showed that some of the dataset is biased ... reviewers may be concerned with this bias ... Bias 2 mentioned in this paper (http://ink.library.smu.edu.sg/cgi/viewcontent.cgi?article=3425&context=sis_research) is particularly important.

Xuan says: Currently we only use Paveneet's datasets (H,J and L), and yes we have considered the bias, and we only use the unbiased data sets. The "fully localized" bug reports are filtered. Our previous data sets are bias, so we are not sure if we use here is suitable. If necessary, we can conduct several comparison experiments ono more data sets.

4.3 Evaluation Metrics

Following prior bug localization studies [5, 17, 18, 24], we consider three evaluation metrics: Top-N, Mean Average Precision (MAP), and Mean Reciprocal Rank. Their brief definitions considering the context of bug localization are given below:

Top-N. Top-N reports the proportion of bug reports for which one of the buggy files appear in the top-N position in the ranked list returned by a bug localization tool. Following prior studies [5, 17, 18, 24], we consider three values of N, i.e., 1, 5, and 10. This is further motivated by the findings of Kochhar et al. [10] which highlight that more than 95% of practitioners would not check beyond the top-10 results of a bug localization tool.

Mean Average Precision (MAP). For each ranked list produced by a bug localization technique for a bug report, we can compute its average precision (AP) as follows:

$$\sum_i \frac{P(i) \times isBuggy(i)}{\#buggyfiles} \tag{3}$$

[1]http://hc.apache.org/httpcomponents-client-ga/index.html
[2]https://jackrabbit.apache.org/jcr/index.html
[3]http://lucene.apache.org/

In the above equation, $P(i)$ is the precision at position i (i.e., proportion of files at position 1 to i that are buggy), while $mathitisBuggy(i)$ is 1 if the file at position i is buggy and 0 otherwise. The denominator of the equation is the number of buggy files in the entire ranked list. MAP is the mean of the APs considering all bug reports.

Mean Reciprocal Rank (MRR). For each ranked list produced by a bug localization technique for a bug report, we can compute the position of the first file that is buggy. The reciprocal of such position is referred to as the reciprocal rank (RR). MRR is the mean of the RRs considering all bug reports.

4.4 Baselines

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

- VSM (Vector Space Model) [16]: a baseline method that firstly uses Vector Space Model to represent the text bug reports and source code, then employs Logistic Regression to predict the related buggy source code.
- Burak (Burak Filter) [15]: a state-of-the-art method for cross-project and cross-company defect prediction problem, which filters training sets using Burak filter that employs k-nearest neighbour to selects instances in the source project similar to the test project.
- TCA-R (Transfer Component Analysis with Logistic Regression): a state-of-the-art transfer learning method in software engineering, which firstly employ TCA to map source and target project into a same feature space and then apply Logistic Regression for bug localization (same settings suggested in their paper).
- TCA-P (Transfer Component Analysis with Multi-layer Perceptron): a state-of-the-art transfer learning method in software engineering, which firstly employ TCA to map source and target project into a same feature space and then apply MLPs for bug localization (same settings with fully-connected layers in TRANP-CNN).
- TCA-D (Transfer Component Analysis with Deep features): a state-of-the-art transfer learning method in software engineering, which firstly employ TCA to map source and target project into a same feature space and then apply Logistic Regression for bug localization (using deep features extracted from CNN instead of TFIDF features).
- NP-CNN (Natural and Programming language Convolutional Neural Network) [5]: a state-of-the-art deep model for bug localization, which use source project data for training and localizing the buggy source code for target project data.
- SimpleTrans (Simple Transfer): a variant of the TRANP-CNN model, which trains the prediction model on the source project data, and use fine tune method for weight adjustment based on the target project.

4.5 Experimental Settings

For parameter settings of baseline methods, we use the same parameter settings suggested in their work [16]. For the TRANP-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 Within-Project feature extraction layers. The number of neurons in fully-connect network in set the same number of CNN. In addition, the drop-out method is also applied 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 0.25 in our experiments.

For data partition, we use data from source projects and 20% target projects as training sets, and locates the 80% buggy code in target projects. This process repeats for 5 times to reduce the influence of randomness, and we report the average results in the next section.

5 EXPERIMENTAL RESULTS

5.1 Experimental Results for Research Questions

Table 2: Performance Comparisons between within-project and cross-project bug localization.

Tasks	*Methods*	*Top 1*	*Top 5*	*Top 10*	*MAP*	*MRR*
J→H	NPCNN	0.317	0.362	0.508	0.276	0.352
	$\text{NPCNN}^{partial}$	0.204	0.258	0.313	0.202	0.292
	NPCNN^{full}	0.533	0.617	0.650	0.472	0.580
L→H	NPCNN	0.142	0.192	0.345	0.161	0.218
	$\text{NPCNN}^{partial}$	0.204	0.258	0.313	0.202	0.292
	NPCNN^{full}	0.533	0.617	0.650	0.472	0.580
H→J	NPCNN	0.167	0.287	0.349	0.247	0.277
	$\text{NPCNN}^{partial}$	0.035	0.211	0.302	0.155	0.189
	NPCNN^{full}	0.508	0.587	0.679	0.462	0.557
L→J	NPCNN	0.152	0.182	0.318	0.176	0.221
	$\text{NPCNN}^{partial}$	0.035	0.211	0.302	0.155	0.189
	NPCNN^{full}	0.508	0.587	0.679	0.462	0.557
H→L	NPCNN	0.173	0.246	0.390	0.196	0.329
	$\text{NPCNN}^{partial}$	0.097	0.219	0.335	0.095	0.109
	NPCNN^{full}	0.289	0.484	0.611	0.287	0.387
J→L	NPCNN	0.110	0.255	0.323	0.141	0.176
	$\text{NPCNN}^{partial}$	0.097	0.219	0.335	0.095	0.109
	NPCNN^{full}	0.289	0.484	0.611	0.287	0.387
Avg.	NPCNN	0.177	0.254	0.372	0.200	0.262
	$\text{NPCNN}^{partial}$	0.112	0.229	0.317	0.151	0.197
	NPCNN^{full}	0.443	0.563	0.647	0.407	0.508

RQ1: *Is there a need for cross-project bug localization?*

To answer this research question, we compare the results of using NP-CNN for bug localization in different settings.

- NPCNN: Employ NPCNN directly for cross-project bug localization, which means directly training the model on the data from source projects and locating the bugs in the target project.
- $\text{NPCNN}^{partial}$: Employ NPCNN using partial data of target projects, which means training based on a few data (20%) in the target projects, and localizes target buggy files without using data from source project.
- NPCNN^{full}: Employ NPCNN using full data of target projects. In this setting, we conduct 5-folds cross-validation for comparison.

The results are detailed in Tab. 2. There are six tasks in the table, in which **H** represents project *httpclient*, **J** represents project *jackrabbit* and **L** represents *lucene-solr*. Meanwhile, the task $\mathbf{H} \rightarrow \mathbf{J}$ represents using *httpclient* as source project and predicts the location of buggy files in target project *jackrabbit*. The results show that the performance of bug localization using full data of target projects is the best, which has a large gap against the performance using partial data. For cross-project bug localization, the performance of NPCNN that directly uses source projects is better than NPCNN^{within}, showing that cross-project data is beneficial to improve the bug localization performance, but directly using within-project bug localization technique will not as well as NPCNN^{full}. The results suggest that there is a need for cross-project bug localization, and directly using within-project bug localization method does not show good performance.

RQ2: *Does the cross-project feature fusion layer improve the bug localization performance?*

To answer this research question, we compare the results of TRANP-CNN with NPCNN and SimpleTrans (Simple Transfer). The difference of the structure between TRANP-CNN and NPCNN is that TRANP-CNN employs two fully-connected networks to combine deep features from source projects and target projects in the cross-project feature fusion layers, respectively, which will counter the influences that cross-project data may have different distribution leading to a bias performance. The results are detailed in Tab. 3.

According to the results, we find that SimpleTrans improves a little performance against NPCNN, showing that transfer technique is effective in improving cross-project bug localization performance. Meanwhile, it can be obviously found that TRANP-CNN performs better than NPCNN and SimpleTrans in terms of all evaluation metrics, which shows that the structure of cross-project feature fusion layer is able to improve cross-project bug localization performance.

The results show that TRANP-CNN performs better than NPCNN and SimpleTrans, which suggests that the cross-projects feature fusion layers can improve the performance of cross-projects bug localization.

RQ3: *Can TRANP-CNN outperform other bug localization methods?*

To answer this research question, we compare the results of TRANP-CNN with state-of-the-art methods: VSM (Vector Space Model), Burak (Burak Filter), TCA-R (Transfer Component Analysis with Logistic Regression), TCA-P (Transfer Component Analysis with Multi-layer Perceptron). Vector Space Model is a baseline technique used in the within-project bug localization and we employ it on cross-project bug localization for comparison. Burak and TCA have been shown good performance on cross-project and cross-company defect prediction, and also we apply it on cross-project bug localization. The parameters are set suggested in their work, i.e., $k = 2$ in Burak method, and for TCA, we implement the algorithm with TCA+. For fair comparison, the classifier in their original paper (Logistic Regression) and multi-layer perception (same as TRANP-CNN) is compared in our experiments. The results are detailed in Tab. 4.

According to the results, we have several findings: 1. Burak and TCA techniques perform better than the baseline VSM model, indicating that using transfer algorithms is able to improve the performance in cross-project bug localization; 2. TRANP-CNN outperforms TCA-P, which shows that the high-level features extracted from CNN are more semantic and informative, leading to a better representation and bug localization performance; 3. TCA-D uses deep features extracted from CNN and the performance is not as well as TRANP-CNN, which further proves that the cross-project feature fusion layers improve bug localization performance; 4. TRANP-CNN obtains the best average values in terms of all evaluation metrics, suggesting that TRANP-CNN outperforms other traditional bug localization methods and transfer techniques on software engineering.

Table 3: Performance Comparisons with previous deep models.

Tasks	*Methods*	*Top 1*	*Top 5*	*Top 10*	*MAP*	*MRR*
J→H	NPCNN	0.317	0.362	0.508	0.276	0.352
	SimpleTrans	0.354	0.396	0.563	0.298	0.395
	TRANP-CNN	0.500	0.583	0.625	0.376	0.543
L→H	NPCNN	0.142	0.192	0.345	0.161	0.218
	SimpleTrans	0.163	0.146	0.354	0.141	0.246
	TRANP-CNN	0.275	0.35	0.488	0.242	0.332
H→J	NPCNN	0.167	0.287	0.349	0.247	0.277
	SimpleTrans	0.133	0.324	0.365	0.273	0.301
	TRANP-CNN	0.396	0.443	0.514	0.371	0.434
L→J	NPCNN	0.152	0.182	0.318	0.176	0.221
	SimpleTrans	0.144	0.204	0.382	0.247	0.249
	TRANP-CNN	0.460	0.462	0.488	0.404	0.478
H→L	NPCNN	0.173	0.246	0.390	0.196	0.329
	SimpleTrans	0.197	0.323	0.426	0.152	0.313
	TRANP-CNN	0.361	0.445	0.535	0.279	0.414
J→L	NPCNN	0.110	0.255	0.323	0.141	0.176
	SimpleTrans	0.140	0.282	0.342	0.163	0.224
	TRANP-CNN	0.301	0.410	0.517	0.247	0.368
Avg.	NPCNN	0.177	0.254	0.372	0.200	0.262
	SimpleTrans	0.189	0.279	0.405	0.212	0.288
	TRANP-CNN	0.382	0.449	0.528	0.320	0.428

6 DISCUSSION

6.1 Why do the heterogeneous predicting adaptation layers work?

Firstly, we explore the reason why the heterogeneous prediction adaptation layers work in the TRANP-CNN model. The key part of TRANP-CNN is the heterogeneous predicting adaptation layers, and in this section, we discuss why the heterogeneous predicting adaptation layers work.

The only difference between NP-CNN and TRANP-CNN is that the TRANP-CNN applies heterogeneous predicting adaptation layers, which is particularly designed for cross-project bug localization to deal with the situation that the distribution of data from source project and target project may be different, leading to a bias results if the prediction structure is the same. The TRANP-CNN model employs two fully-connected network for bug localization prediction, one is for source project and the other one is used for target project. This structure overcomes the problem that the model training from

Table 4: Performance comparisons with traditional bug localization performance.

Tasks	*Methods*	*Top 1*	*Top 5*	*Top 10*	*MAP*	*MRR*
J→H	VSM	0.098	0.157	0.177	0.087	0.143
	Burak	0.110	0.126	0.138	0.116	0.121
	TCA-R	0.120	0.212	0.144	0.157	0.162
	TCA-P	0.114	0.133	0.154	0.123	0.176
	TCA-D	0.122	0.225	0.271	0.168	0.248
	TRANP-CNN	0.500	0.583	0.625	0.376	0.543
L→H	VSM	0.059	0.098	0.237	0.099	0.112
	Burak	0.113	0.203	0.242	0.143	0.143
	TCA-R	0.120	0.188	0.244	0.151	0.158
	TCA-P	0.128	0.200	0.252	0.161	0.167
	TCA-D	0.102	0.237	0.367	0.161	0.202
	TRANP-CNN	0.275	0.350	0.488	0.242	0.332
H→J	VSM	0.035	0.211	0.232	0.165	0.129
	Burak	0.130	0.150	0.206	0.225	0.195
	TCA-R	0.115	0.162	0.209	0.239	0.244
	TCA-P	0.114	0.154	0.203	0.237	0.241
	TCA-D	0.111	0.135	0.157	0.168	0.185
	TRANP-CNN	0.396	0.443	0.514	0.371	0.434
L→J	VSM	0.197	0.212	0.293	0.167	0.216
	Burak	0.161	0.132	0.368	0.170	0.187
	TCA-R	0.136	0.183	0.370	0.170	0.179
	TCA-P	0.114	0.116	0.397	0.138	0.191
	TCA-D	0.178	0.236	0.469	0.227	0.256
	TRANP-CNN	0.460	0.462	0.488	0.404	0.478
H→L	VSM	0.083	0.278	0.393	0.154	0.136
	Burak	0.105	0.226	0.272	0.123	0.222
	TCA-R	0.136	0.208	0.383	0.170	0.279
	TCA-P	0.143	0.226	0.394	0.171	0.288
	TCA-D	0.162	0.207	0.345	0.229	0.292
	TRANP-CNN	0.361	0.445	0.535	0.279	0.414
J→L	VSM	0.038	0.077	0.154	0.124	0.204
	Burak	0.138	0.161	0.176	0.168	0.226
	TCA-R	0.135	0.111	0.172	0.169	0.222
	TCA-P	0.136	0.132	0.192	0.173	0.237
	TCA-D	0.142	0.297	0.308	0.238	0.293
	TRANP-CNN	0.301	0.410	0.517	0.247	0.368
Avg.	VSM	0.085	0.172	0.248	0.133	0.157
	Burak	0.126	0.166	0.234	0.157	0.182
	TCA-R	0.127	0.178	0.254	0.176	0.207
	TCA-P	0.125	0.160	0.265	0.167	0.217
	TCA-D	0.136	0.223	0.319	0.199	0.246
	TRANP-CNN	0.382	0.449	0.528	0.320	0.428

two projects affected from each other. During training process, the data of source project uses the CNN model in the transferable feature extraction layers for feature extraction and employs the fully-connected network fc_s for prediction training, and the target project data is trained using the same CNN but predicted with fully-connected network fc_t. This process helps improve the bug localization performance from target project by enjoying the advantage in sharing the same network of transferable feature extraction process from source project, and meanwhile adapting prediction network using training data from target project.

From the experiments results, we still find that SimpleTrans outperforms NP-CNN in terms of most evaluation metrics. SimpleTrans has the same structure with NP-CNN, but fine tune the parameters of fully-connected network using target project data, which further shows that using target project data to adjust the weight in the fully-connected network for prediction is effective for bug localization. TRANP-CNN outperforms NPCNN and SimpleTrans shows that the heterogeneous predicting adaptation layers are able to improve the performance of bug localization.

6.2 Why does TRANP-CNN improve the bug localization performance?

The reason why TRANP-CNN improve the bug localization performance can be summarized as 4 folds:

- The transferable feature extraction layers are able to generate a more semantic representation of source code. The transferable feature extraction layers are able to extract the semantic features reflecting the structural and sequential nature, leading to a high-level representation of source code. In addition, the results in Table 4 that TCA-D (TCA with deep features) outperforms TCA-P (TCA with traditional features), which show that the deep features generated by transferable feature extraction layers are able to improve the performance of cross-project bug localization performance.
- The transferable feature extraction layers can improve the performance in cross-project bug localization. As aforementioned, since source project and target project use the same programming language, the semantic feature extraction rule of cross-project is similar, which means that the semantic feature extraction sub-structure from source project could be transferable to the target model. The comparison results between TRANP-CNN and NPCNN have supported this reason.
- The heterogeneous predicting adaptation layers are effective for cross-project bug localization. The reason has been explained in the last subsection that the heterogeneous predicting adaptation layers help counter the inconsistent distribution problems in cross-project bug localization task by employing two fully-connected network for predicting adaptation.
- TRANP-CNN can fully exploit the advantage in using the labeled data from target project. A few data (20% in our experiments) from the target project has labels. TRANP-CNN is able to make better use of labeled data from target project in training the fully-connected network fc_t for prediction during the training process. However, traditional transfer model TCA+ has not fully used the labeled data in the target project from transfer learning view.

6.3 Threats to Validity

David says: Need to complete when the dataset used is finalized.

There are three kinds of threat that may impact the validity of this study: threats to internal validity, threats to external validity, and threats to construct validity. We acknowledge these threats below.

Threats to internal validity relate to author bias and errors in our code and experiments. We have checked our code for bugs and fixed any that we can identify. There may still be errors that we do not notice though. The dataset that we obtain are taken from prior papers [9, 24] and have been used to evaluate other bug localization techniques, e.g., [5, 17, 24]. The data are bug reports taken from bug tracking systems from real projects (i.e., ...) and thus are realistic. Thus, we believe there are limited threats to the internal validity of the study.

Threats to external validity relate to the generalizability of the study. We have analyzed data that includes ... bug reports taken from ... projects. Admittedly, the projects that we analyze may not represent all the projects out there. Still, our threats to external validity are less than existing bug localization work since the amount of data that we investigate is larger than prior work. For example, Zhou et al. only use ... bug reports from ... projects [24], Saha et al. only use ... bug reports from ... reports [17], and Huo et al. only use ... bug reports from ... projects [5]. In a future work, we plan to reduce the threats to external validity further by investigating more bug reports from additional projects.

Threats to construct validity relate to the suitability of our evaluation metrics. We have used ... as evaluation metrics. These metrics were also used by prior bug localization studies, e.g., [5, 17, 24]. Thus, we believe there are limited threats to construct validity.

7 RELATED WORK

In this section, we first describe existing work on bug localization in Section 7.1. Next, we present existing work that also deal with cold-start problem in software engineering in Section 7.2. Finally, we describe recent effort in software engineering that adapts deep learning to software engineering in Section 7.3.

7.1 Bug Localization

A number of papers have proposed various techniques that take as input a bug report and return a ranked list of source code files that are relevant to it [5, 12, 16–18, 24]. These *text-based* bug localization techniques can be divided into two general families: supervised approaches [5, 24] and unsupervised ones [12, 16–18]. Supervised approaches learn a model from data of bug reports whose relevant buggy source code files have been identified. Unsupervised approaches do not learn such model. We briefly introduce some of the approaches that belong to each family below. Due to space limitation, our survey here is by no means complete.

Unsupervised Approaches. Lukins et al. apply Latent Dirichlet Allocation (LDA) to extract latent topics from source code files and bug reports [12]. Given an input bug report, source code files that are similar in their topic distributions as the bug report are returned. Rao et al. investigate a number of generic and composite text retrieval models, e.g., Unigram Model (UM), Vector Space Model (VSM), Latent Semantic Analysis (LSA), Latent Dirichlet Allocation (LDA), etc., for bug localization [16]. Their empirical study demonstrates that simple text retrieval models, i.e., unigram model and vector space model, are performing the best. Saha et al. apply structured information retrieval to improve the performance of existing solutions further [18]. Their proposed approach, named BLUiR, separates text in the source code files and bug reports into different groups and compute similarities between the different groups separately before combining the similarity scores together. In particular, it separates text in source code files into class names, method names, identifier names and comments, and text in bug reports into summary and description. In a later work, Saha et al. reports an extended evaluation of BLUiR with several thousand more bug reports [17].

Supervised Approaches. Zhou et al. employs a modified Vector Space Model (i.e., rVSM) and makes use similar fixed bug reports to boost bug localization performance [24]. Their proposed approach employs lazy learning; it stores past fixed bug reports and compares incoming bug reports to these historical bug reports. Source code files are then ranked based on how often they are fixed to address prior bug reports. Zhou et al. have shown that their proposed approach named Bug Locator outperforms many unsupervised approaches, e.g., VSM, SUM, LSI and LDA. More recently, Huo et al. [5] extends Zhou et al.'s work by employing a eager learning approach based on Convolutional Neural Network (CNN) [7]. They demonstrate that their proposed approach named NP-CNN outperforms Bug Locator.

In this work, we propose a novel deep transfer learning approach that is built on top of NP-CNN [7]. To the best of our knowledge, we are the first to explore cross-project bug localization. We have also demonstrated that our proposed approach named DTBL outperforms NP-CNN for cross-project setting.

7.2 Cross-Project Learning

The problem of scarcity of labelled data for a target project (aka. cold-start problem) has been explored in several automated software engineering tasks [8, 13, 20, 25]. Closest to our work, is the line of work on cross-project defect prediction [13, 20, 25]. Note that defect prediction does not consider a target bug report, while bug localization takes as input a bug report and return files relevant to it. They are used in different software development phase, i.e., code inspection and testing (defect prediction) vs. debugging (bug localization), and thus are thus complementary with each other. We provide a description of existing work on cross-project defect prediction below. Due to space limitation, our survey here is by no means complete.

Zimmermann et al. are among the first to investigate cross-project defect prediction [25]. They highlight that defect prediction works well if there is a sufficient amount of data from a project to train a model. However, they argue that sufficient data is often unavailable for many projects (especially new ones) and companies. One way to deal with the problem is to build a model from a project with sufficient data and use the model to predict defective code in another project – which is referred to as cross-project defect prediction. To investigate viability of cross-project defect prediction, Zimmermann et al. consider 12 target projects and demonstrate that cross-project defect prediction is "a serious challenge" – it is not possible to achieve good results by simply using models built from other projects.

Zimmermann et al.'s study is a call-to-arms that spur active interest in the area of cross-project defect prediction. A number of solutions have been proposed to boost the effectiveness of cross-project defect prediction. These include the work by Turhan et al. [20] and Nam et al. [13] highlighted below.

Turhan et al. propose a relevancy filtering method to select training data that are closest to test data [20]. In particular, they employ a k-nearest neighbor method to pick k training instances (i.e., files from a project with known defect labels) that are closest to each test data (i.e., files from a target project with unknown defect labels). The resultant training instances are then used to learn a model that is then applied to predict defect labels of files from the target project in the test data. The approach by Turhan et al. potentially omit many training instances, which may reduce the effectiveness of the resultant model. Nam et al. deal with cross-project defect prediction problem by leveraging the recent development in machine learning – i.e., transfer learning [13]. In particular, they take an existing transfer learning method – referred to as Transfer Component Analysis (TCA) [14] – and adapt it for defect prediction.

Following existing cross-project defect prediction studies, we first demonstrate that cross-project bug localization is a serious challenge (see RQ1 in Section 4). We then propose a novel deep transfer learning method to deal with this challenge. We have also compared our solution with several adaptations of Turhan et al.'s relevancy filtering method [20] and TCA [14] for bug localization, and demonstrated that our solution outperforms these baselines.

7.3 Deep Learning in Software Engineering

Recently, deep learning [3], which is a recent breakthrough in machine learning domain, has been applied in many areas. Software engineering is not an exception. Our approach is built upon the state-of-the-art bug localization technique employing deep learning [5]. In this subsection, we briefly review some related studies that also employ deep learning to improve other automated software engineering tasks. In the process, we highlight the difference between our approach and the existing work, and thus stress our novelty. Due to space limitation, our survey here is by no means complete.

Yang et al. applies Deep Belief Network (DBN) to learn higher-level features from a set of basic features extracted from commits (e.g., lines of code added, lines of code deleted, etc.) to predict buggy commits [23]. Wang et al. applies Deep Belief Network (DBN) to tokens extracted from program Abstract Syntax Trees to better predict defective files [21]. Specifically, DBN is used to extract semantic vectors that are then used as input to a classifier to learn a model to differentiate defective from non-defective files. Guo et al. uses word embedding and one/two layers Recurrent Neural Network (RNN) to link software subsystem requirements (SSRS) to their corresponding software subsystem design descriptions (SSDD) [4]. They have evaluated their solution on 1,651 SSRS and 466 SSDD from an industrial software system. Xu et al. applies word embedding and convolutional neural network (CNN) to predict semantic links between knowledge units in Stack Overflow (i.e., questions and answers) to help developers better navigate and search the popular knowledge base [22]. Lee et al. applies word embedding and CNN to identify developers that should be assigned to fix a bug report [11].

While existing works mostly take an off-the-shelf deep learning algorithm (e.g., DBN, CNN, etc.) and apply it to solve their problem, in this work, we design a customized deep learning algorithm and demonstrates that it works better than off-the-shelf solutions. David says: Xuan and Ming, if you have stronger points to highlight the novelty of our approach compared to the above papers, please kindly help to add it here :-)

8 CONCLUSION AND FUTURE WORK

REFERENCES

[1] John Anvik, Lyndon Hiew, and Gail C. Murphy. 2005. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse Technology eXchange, ETX 2005, San Diego, California, USA, October 16-17, 2005.* 35–39.

[2] Tegawendé F. Bissyandé, Ferdian Thung, Shaowei Wang, David Lo, Lingxiao Jiang, and Laurent Réveillère. 2013. Empirical Evaluation of Bug Linking. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013.* 89–98.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning.* MIT Press. http://www.deeplearningbook.org.

[4] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. 2017. Semantically enhanced software traceability using deep learning techniques. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017.* 3–14.

[5] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2016. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence.* New York, NY, USA, 1606–1612.

[6] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA.* 273–282.

[7] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of Conference on Empirical Methods in Natural Language Processing.* Doha, Qatar, 1746–1751.

[8] Barbara A. Kitchenham, Emilia Mendes, and Guilherme Horta Travassos. 2007. Cross versus Within-Company Cost Estimation Studies: A Systematic Review. *IEEE Trans. Software Eng.* 33, 5 (2007), 316–329.

[9] Pavneet Singh Kochhar, Yuan Tian, and David Lo. 2014. Potential biases in bug localization: do they matter?. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014.* 803–814.

[10] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016.* 165–176.

[11] Sunro Lee, Min-Jae Heo, Chan-Gun Lee, Milhan Kim, and Gaeul Jeong. 2017. Applying deep learning based automatic bug triager to industrial projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017.* 926–931.

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

[13] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer defect learning. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013.* 382–391.

[14] Sinno Jialin Pan, Ivor W. Tsang, James T. Kwok, and Qiang Yang. 2011. Domain Adaptation via Transfer Component Analysis. *IEEE Trans. Neural Networks* 22, 2 (2011), 199–210.

[15] Fayola Peters, Tim Menzies, and Andrian Marcus. 2013. Better cross company defect prediction. In *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories.* IEEE, 409–418.

[16] Shivani Rao and Avinash Kak. 2011. 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.* ACM, 43–52.

[17] Ripon K. Saha, Julia Lawall, Sarfraz Khurshid, and Dewayne E. Perry. 2014. On the Effectiveness of Information Retrieval Based Bug Localization for C Programs. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014.* 161–170.

[18] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013.* 345–355.

[19] G. Tassey. 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing. In *National Institute of Standards and Technology, RTI Project, vol. 7007, no. 011, 2002.*

[20] Burak Turhan, Tim Menzies, Ayse Basar Bener, and Justin S. Di Stefano. 2009. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering* 14, 5 (2009), 540–578.

[21] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference*

on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016. 297–308.

[22] Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. 2016. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 51–62.

[23] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep Learning for Just-in-Time Defect Prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5, 2015*. 17–26.

[24] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 14–24.

[25] Thomas Zimmermann, Nachiappan Nagappan, Harald C. Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*. 91–100.