Deep Transfer Bug Localization

ABSTRACT

Many projects often receive more bug reports than what they can handle. To help debug and close bug reports, a number of bug localization techniques have been proposed. These techniques analyze a bug report and return a ranked list of potentially buggy source code files. Recent development on bug localization has resulted in the construction of effective supervised approaches that use historical data of manually localized bugs to boost performance. Unfortunately, as highlighted by Zimmermann et al., sufficient bug data is often unavailable for many projects and companies. This raises the need for *cross-project bug localization* – the use of data from a project to help locate bugs in another project. To fill this need, we propose a deep transfer learning approach for cross-project bug localization. Our proposed approach named TRANP-CNN extracts transferable semantic features from source project and fully exploits labeled data from target project for effective cross-project bug localization. We have evaluated TRANP-CNN on curated bug datasets from Herzig et al. and Kochhar et al. The experiments show that averaging across the datasets, it can locate buggy files correctly at top-1, top-5, and top-10 positions for 38.22%, 44.88%, 52.78% of the bugs respectively. It can outperform a state-of-the-art bug localization solution based on deep learning and several other advanced alternative solutions by 61.13% to 349.61% considering various standard evaluation metrics.

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 [27]. 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 [9, 10, 17, 23, 24]. Many of them analyze the description of a bug report to identify source code files relevant to it [9, 17, 23, 24]. 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 [17, 23, 24]. More recently, supervised approaches are introduced [9]. 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). DTBL 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. The DTBL model firstly employs transferable feature extraction layers to extract semantic features of bug reports and source code from both source and target projects, and then heterogeneous predicting adaptation layers are applied to learn prediction function for source project data and target data separately.

There have been a number of transfer learning solutions designed to help with cold-start problem in software engineering context. For example, Turhan et al. [28] proposed Burak Filter to select k nearest instances from source domain similar to the target domain to build training models. Another transfer approach TCA was proposed by Nam et al. [19], which maps source and target domain data into a same feature space to overcome the different data distribution problem in cross-project tasks. Our approach is unique from previous solutions in the following ways. First, TRANP-CNN employs two convolutional neural network to extract semantic features for bug localization; Second, TRANP-CNN employs transferable feature extraction layers to improve bug localization performance; Third, TRANP-CNN can fully exploit the advantage in using the labeled data from target project, while TCA only uses the distribution of target domain for transfer task. In addition, NP-CNN proposed by Huo et al. [9] has shown good performance in bug localization by introducing particular framework to learn unified features from bug reports and source code. TRANP-CNN is unique from NPCNN in the way that TRANP-CNN employs heterogeneous predicting adaptation layers to apply two fully-connected network to train predictors separately from source project and target projects, which enjoy the advantage in extracting the high-level semantic features with the same deep model and training the prediction networks using different models to overcome the inconsistency data distribution. David says: Please compare the approach with existing work and highlight its novelty.

David says: Please add brief description of the results. We have evaluated our proposed solution on 6 tasks from several open source projects. 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 reports). The experiments results also show that our proposed DTBL model TRANP-CNN outperforms previous state-of-the-art bug localization methods on all 6 tasks in terms of all evaluation metrics. In addition, the experiments have highlight the effectiveness of heterogeneous predicting adaptation layers, which is the key part of TRANP-CNN .

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) David says: Please add very brief description of the approach highlighting its novelty. We propose a deep transfer learning model named TRANP-CNN that employs programming language specific convolutional neural network to extract transferable semantic features, and a novel heterogeneous predicting adaptation layers are designed to improve cross-project bug localization performance.
(3) David says: Please add very brief description of the results. We have evaluated our proposed approach on several open source projects. The results show that the TRANP-CNN model significantly improve the performance on all 6 tasks in terms of Top-k rank, MAP and MRR metrics.

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. [9]. 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 [11], and CNN for programming language is specifically designed.

Huo et al. [9] 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 [9].

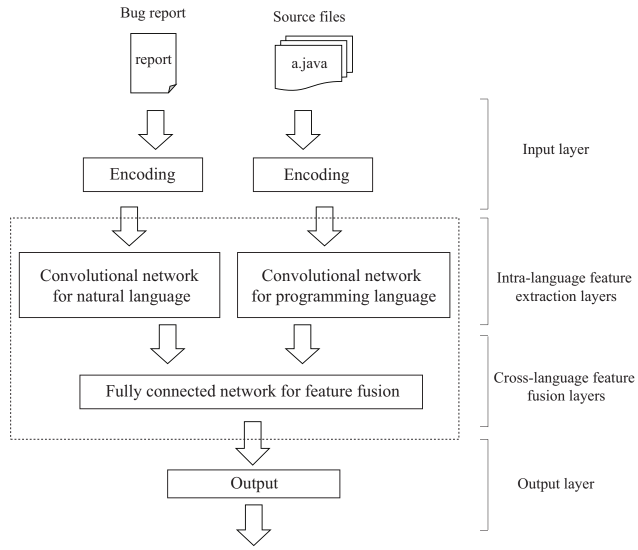

Figure 1: The general structure of Natural and Programming language Convolutional Neural Network.

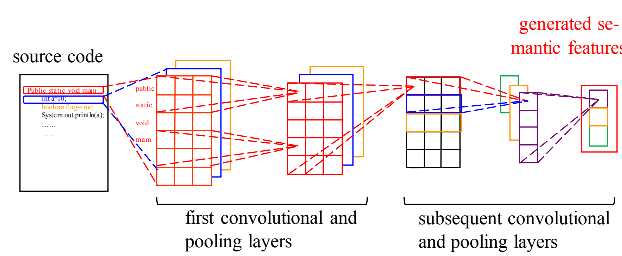

Figure 2: The structure of convolutional neural network for programming language.

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).

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

3 PROPOSED APPROACH

In cross-project bug localization, we are provided with a *source* project with many bug reports carefully localized to the corresponding source code, and a *target* project in which only several bug reports are localized. The goal is to construct a model to fully leverage rich information from the source project to facilitate effective bug localization for the target project.

Let $C^s = \{c_1^s, c_2^s, \cdots, c_{n_1^c}^s\}$ and $C^t = \{c_1^t, c_2^t, \cdots, c_{n_2^c}^t\}$ denote the set of source code files from the source project and the target project, respectively, and $C = C^s \bigcup C^t$. Let $\mathcal{R}^s = \{r_1^s, r_2^s, \cdots, r_{n_1^r}^s\}$ and $\mathcal{R}^t = \{r_1^t, r_2^t, \cdots, r_{n_2^r}^t\}$ denote the set of bug reports from the source project and target project, respectively, and $\mathcal{R} = \mathcal{R}^s \bigcup \mathcal{R}^t$. In the above notations, $n_1^c, n_2^c, n_1^r, n_2^r$ 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 prediction functions $\mathbf{f} = (f^s, f^t)$, where $f^\alpha : \mathcal{R}^\alpha \times C^\alpha \mapsto \mathcal{Y}^\alpha$. $y_{ij}^\alpha \in \mathcal{Y}^\alpha = \{+1, -1\}$ indicates whether a source code $c_j^\alpha \in C^\alpha$ is relevant to a bug report $r_i^\alpha \in \mathcal{R}^\alpha$, where $\alpha \in (s, t)$.

Source and target projects may differ from each other in the way how a bug report is localized to the corresponding source code files. To effectively learn the prediction function for a target project, one problem should be addressed carefully: how to identify information from a source project that is potentially useful for learning the prediction function for the target project? Intuitively, if information from the source project can be used for the target project, both must share latent commonalities. Differences observed in the two projects are due to the way these latent commonalities are manifested. Therefore, we argue that a model that facilitates an effective cross-project bug localization can be constructed by 1) learning a shared latent feature representation from both source and target project, and 2) biasing the learner towards specific project considering the shared latent feature representation.

We instantiate the aforementioned idea by proposing a novel deep transfer neural network named TRANP-CNN (TRAnsfer Natural and Program Language Convolutional Neural Network). Firstly, TRANP-CNN takes bug reports and source code files as inputs and learns a common transferable feature representation shared by both source and target projects. Next, TRANP-CNN simultaneously creates a pair of prediction functions that are biased towards the source and target project, respectively, based on the shared feature representation, in order to generate project-specific predictions for the target domain. David says: Why do we need to learn a prediction function for the source project? This is not clear to me ...

3.1 Model Structure

The model structure of TRANP-CNN is depicted in Figure 3, where the subfigure to the left depicts the training process of TRANP-CNN , and the one to the right depicts the corresponding test process.

TRANP-CNN consists of four layers: input layer, transferable feature extraction layer, project-specific correlation fitting layer and output layer. The input layer takes bug reports as well as the source code files in their original formats and generate their corresponding encodings such that they can be further processed by the subsequent layers of TRANP-CNN . The transferable feature extraction layer aims to learn an intermediate latent feature representation from the bug reports and source code files shared by source and target projects. The project-specific prediction layer is responsible for biasing learning, based on the transferable feature representation learned in the previous layer, to identify project-specific correlation patterns between bug reports and source code files in source and target projects. The output layer generates the final correlation scores for report-file pairs (i.e., pairs of bug reports and source code files). It is obvious that the the transferable feature extraction layers and project-specific correlation fitting layers are the key parts of the proposed model, which would be explained in details in the following subsections.

To train TRANP-CNN model, pairs of bug reports and source code files from the source and target projects along with their ground truth labels (i.e., buggy or not) are fed into the proposed deep model in order to learn the transferable feature representation shared by both source and target projects as well as the project-specific prediction functions for the source and target project, respectively. After the model is fully trained, prediction function f^t would be used for determine the correlation of each report-file pair (r_i^t, c_j^t) from the target project.

3.2 Transferable Feature Extraction Layer

When provided with bug reports and source code files from source and target projects in their original format, the encodings for the bug reports and the source codes are first generated by the input layer. Traditional TF-IDF (term frequency - inverse document frequency) representation [5] fails to capture correlation between terms. Thus, we employ word2vec [18] encoding to represent both bug reports and source code files with the purpose of enriching the initial representation, based on which, the transferable features are further extracted.

The transferable features for bug reports and source codes files should satisfy the following properties. First, the transferable features should be able to represent the functional semantics in both bug reports and source code files such that the semantics can be further utilized to identify the correlation patterns between the reports and the files. Second, the extracted semantics should be able to capture some common knowledge between the source and target project such that knowledge learned from the source project can eventually be transferred to facilitate learning for the target project.

To allow for effective knowledge transfer between source and target projects, a key challenge here is how to identify what information is useful and transferable and what is useful but may not be transferable. To address this challenge, we employ a special

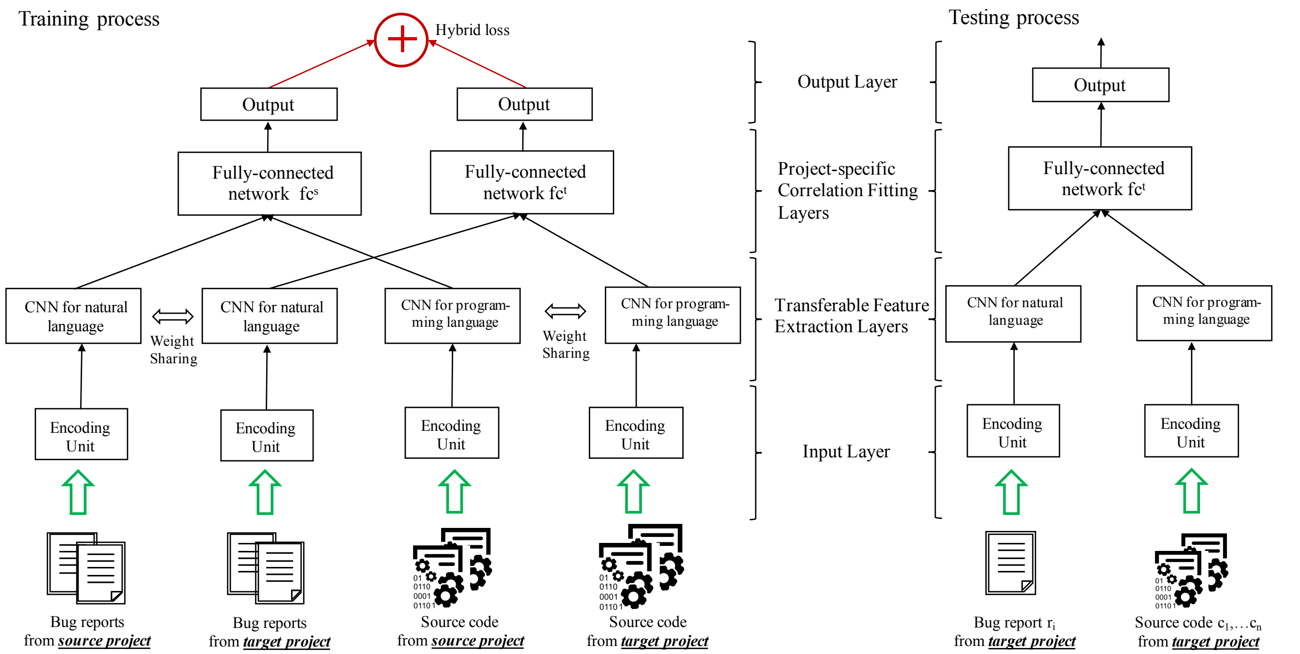

Figure 3: The overall structure of Transfer Natural and Programming language CNN. The left part is the training process of TRANP-CNN based on the bug reports and source code from source projects and a few data from target projects, the weights of which are trained by minimizing the loss of ensemble loss 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.

strategy we refer to as *weight sharing* during the learning process, which imposes a hard constraint that the learned weights in the networks (both N-CNN and P-CNN) for the target project should be exactly the same as the learned weights in the networks for the source project. By weight sharing, the learning procedure that optimizes for good performance on both source project and target project is forced to focus on common features shared by both source and target projects rather than extracting project-specific features for each project. Thus, the resulting features allow for *transfer* of common knowledge that is useful to determine correlation between bug reports and source code files from the source project to the target project.

3.3 Project-Specific Correlation Fitting Layers

After being processed by the transferable feature extraction layers, bug reports (as well as the source code files) from both source and target project are represented using the *same* set of transferable features embedded with common knowledge shared by both source and target projects. However, since the source project and target projects may differ in how bug reports and source code files are correlated, the subsequent project-specific correlation fitting layer is employed to fit a model to capture project-specific correlation patterns considering the same set of transferable features.

David says: Why do we want simultaneously learn two prediction functions? Why not just learn for target project? I'm not fully sure here ...

The key challenge here is how to simultaneously learn two prediction functions with different goals (i.e., to perform well in the source and target projects, respectively) based on the same set of features within the same model structure. To address this challenge, we construct two fully connected neural network substructures, namely fc^s and fc^t, for the subsequent project-specific correlation fitting layers, where fc^s and fc^t share the same input unit and have different output units. Such a special design facilitates the model to take the same set of transferable features from both bug reports and source code files and fit to correlation pattern of the report-code pair in the source project and target project, respectively, with different substructures fc^s and fc^t.

David says: The following phrase is unclear to me: fit to correlation pattern of the report-code pair in the source project and target project, respectively, with different substructures fc^s and fc^t. David says: I'm not sure how to fit network substructures to correlation patterns? Do we mean we use the substructures to *learn* different correlation patterns?

In order to fit the two network substructures fc^s and fc^t to different correlation patterns within the same model structure, we propose the following hybrid loss function to unify the loss from different goals into one function:

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

In the above equation, $\mathcal{L}$ 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}.

David says: The intuition for the above function may need to be elaborated

By solving this optimization problem using the stochastic gradient descent (SGD), the proposed TRANP-CNN can be effectively learned.

David says: Need to cite SGD. David says: I'm not sure if we learn the whole TRANPCNN using SGD or only some parts of it. If it is some parts of it, which parts are not mentioned in the description above.

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*: Do the heterogeneous predicting adaptation layers improve the bug localization performance?*

In Section 3, we propose to employ heterogeneous predicting adaptation layers 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.

4.2 Datasets

We consider datasets that were previously studied and manually vetted by Herzig et al. [8] and Kochhar et al. [13]. Herzig et al. highlighted that many reports in issue tracking systems are wrongly labelled as bugs when they are actually feature requests (and vice versa). Kochhar et al. highlighted that many bug reports are already *fully localized*, i.e., developers have already identified all buggy source code files in the bug reports. For such bug reports, bug localization tool is no longer needed.

To deal with these biases, Herzig et al. have manually classified 5,591 issue reports from JIRA issue tracking systems of three projects (HTTPClient, Jackrabbit, and Lucene-Java) and 1,810 issue reports from Bugzilla issue tracking systems of two projects (Rhino and Tomcat 5) and released the dataset publicly[1]. Kochhar et al. have analyzed 1,191 reports from JIRA issue tracking systems that were confirmed by Herzig et al. as bug reports. They focused on reports from JIRA issue tracking systems since a number of studies have shown that reports in Bugzilla are often poorly linked with commits that fix them [2, 3], while bug reports in JIRA are often more well-linked as JIRA provides a utility to better connect bug reports to their corresponding commits [4]. Kochhar et al. have identified a set of 398 bug reports that are already *fully localized* and another set of 793 bug reports that are either *partially localized* (i.e., reports where some of the files containing the bugs are explicitly mentioned in the report) or *not localized* (i.e., reports which do not explicitly specify any of the buggy files.). They also have released this dataset publicly[2].

In this work, we consider the 793 bug reports from three software systems: 63 from HTTPClient (H), 534 from Jackrabbit (J), and 196 from Lucene-Java (L), which are provided by Kochhar et al. HTTPClient[3] is a library for implementing the client side of HTTP standard, while Jackrabbit[4] is a content repository, and Lucene is a text search engine[5]. The details of the reports considered in this study are shown in Table **??**. Although the number of reports considered is fewer than those considered in several prior work, these reports are of high-quality. They has been manually vetted before and are absent from well-known biases identified by Herzig et al. and Kochhar et al. Most past studies have ignored these well-known biases and thus introduce a threat to the validity of their findings.

4.3 Evaluation Metrics

Following prior bug localization studies [9, 24, 25, 33], 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 [9, 24, 25, 33], we consider three values of N, i.e., 1, 5, and 10. This is further motivated by the findings of Kochhar et al. [14] which highlight

[1] https://www.st.cs.uni-saarland.de/softevo/bugclassify
[2] https://github.com/smusis/buglocalizationbiases
[3] http://hc.apache.org/httpcomponents-client-ga/index.html
[4] https://jackrabbit.apache.org/jcr/index.html
[5] http://lucene.apache.org/

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:

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

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 $isBuggy(i)$ 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) [23]: 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) [22]: 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+ (Transfer Component Analysis) [20]: a state-of-the-art transfer learning method in software engineering, which firstly normalizes the data and employs 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 normalizes the data and employs 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 normalizes the data and employs 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 VSM features).
- NP-CNN (Natural and Programming language Convolutional Neural Network) [9]: a state-of-the-art deep model for bug localization, which uses source project data for training a deep convolutional neural network and localizes the buggy source code for target project data.

4.5 Experimental Settings

First, the parameter settings of baseline methods (VSM, burak, TCA+, TCA+$^{(P)}$, TCA+$^{(D)}$), we use the same parameters settings suggested in their work [23][20]. The parameters are set the same in [9].

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 for comparison.

5 EXPERIMENTAL RESULTS

5.1 Experimental Results for Research Questions

Table 1: 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
	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
	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
	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
	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
	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
	NPCNN^{partial}	0.097	0.219	0.335	0.095	0.109
	NPCNN^{full}	**0.289**	**0.484**	**0.611**	**0.287**	**0.387**

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.

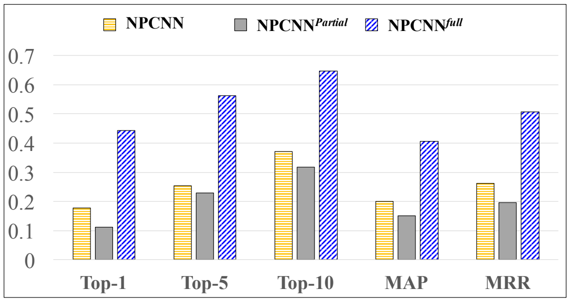

Figure 4: Performance comparisons between within-project and cross-project bug localization.

- 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 Table. 1 and Figuire. 4. There are six tasks in the table, in which **H** represents project *HTTPClient*, **J** represents project *Jackrabbit* and **L** represents *Lucene-Java*. Meanwhile, the task **H** → **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^{partial}, 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: *Do the project-specific correlation fitting layers improve the bug localization performance?*

To answer this research question, we compare the results of TRANP-CNN with the original NPCNN. 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 project-specific correlation fitting 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. 2.

The results show that TRANP-CNN performs better than NPCNN, which suggests that the project-specific correlation fitting layers can improve the performance of cross-projects bug localization. It is because the project-specific correlation fitting layers employ two separate fully-connected network to learn the prediction function for each project separately to encounter the bias of data distribution and the results have provided the evidences.

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

Table 2: 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
	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
	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
	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
	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
	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
	TRANP-CNN	**0.301**	**0.410**	**0.517**	**0.247**	**0.368**

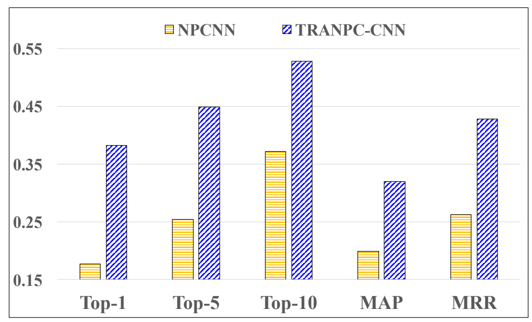

Figure 5: Performance comparisons with deep models.

To answer this research question, we compare the results of TRANP-CNN with state-of-the-art methods: VSM, Burak, TCA+, TCA+$^{(}P$). VSM 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 of VSM and Burak Filter are set suggested in their work. For fair comparison with TCA+, we use same normalization strategy and the classifier in their original paper (Logistic Regression) and multi-layer perception (same as TRANP-CNN) are compared in our experiments. The results are detailed in Tab. 3.

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. Comparing to the best baseline TCA+$^{(D)}$, TRANP-CNN improves the results by 24.6% in terms of Top-1, by 22.6% in terms of Top-5, by 20.9% in terms of Top-10, by 21.9% in terms of MAP and by 17.2% in terms of MRR. According to Mann-Whitney U-test, we find TRANP-CNN significant better in terms of all metrics. The results suggest that TRANP-CNN outperforms other traditional bug localization methods and transfer techniques on software engineering.

Table 3: Performance comparisons with traditional bug localization models.

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**

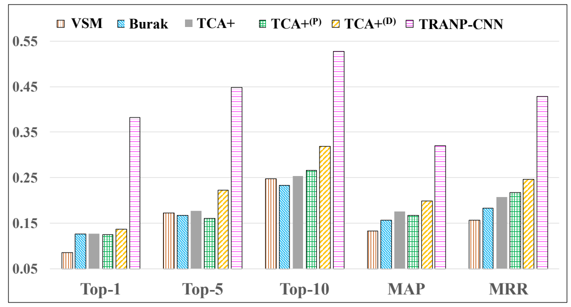

Figure 6: Performance comparisons with traditional bug localization models.

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 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.

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 3 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

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 [13, 33] and have been used to evaluate other bug localization techniques, e.g., [9, 24, 33]. The data are bug reports taken from bug tracking systems from real projects (i.e., Lucene-Java) 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 793 bug reports taken from three projects. Admittedly, the projects that we analyze may not represent all the projects out there. However, to the best of our knowledge, there are no other bug localization dataset that is free from biases presented by Kochhar et al. [13]. In the future, we plan to reduce the threats to external validity by investigating more bug reports that are free from these biases.

Threats to construct validity relate to the suitability of our evaluation metrics. We have used Top-N, MAP, and MRR as evaluation metrics. These metrics were also used by prior bug localization studies, e.g., [9, 24, 33]. 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

Most bug localization techniques are either spectrum-based or text-based. Spectrum-based techniques analyze program spectra produced by running a set of test cases to identify buggy program elements [10, 15, 26, 32]. On the other hand, text-based techniques analyze textual contents in bug reports to identify bug locations [9, 17, 23–25, 33]. These two techniques have different usage scenarios and thus are complementary to each other; one can help developers to fix bugs that manifest as regression test failures, while another can help developers resolve incoming bug reports. In this work, we focus on advancing research on text-based bug localization.

Existing text-based bug localization techniques can be divided into two general families: supervised approaches [9, 33] and unsupervised ones [17, 23–25]. 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 [17]. 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 [23]. 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 [25]. 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 [24].

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 [33]. 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. [9] extends Zhou et al.'s work by employing a eager learning approach based on Convolutional Neural Network (CNN) [11]. 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 [11]. 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 [12, 20, 28, 34]. Closest to our work, is the line of work on cross-project defect prediction [20, 28, 34]. 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 [34]. 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. [28] and Nam et al. [20] highlighted below.

Turhan et al. propose a relevancy filtering method to select training data that are closest to test data [28]. 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 [20]. In particular, they take an existing transfer learning method – referred to as Transfer Component Analysis (TCA) [21] – 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 [28] and TCA [21] for bug localization, and demonstrated that our solution outperforms these baselines.

7.3 Deep Learning in Software Engineering

Recently, deep learning [6], 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 [9]. 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 [31]. Wang et al. applies Deep Belief Network (DBN) to tokens extracted from program Abstract Syntax Trees to better predict defective files [29]. 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) [7]. 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 [30]. Lee et al. applies word embedding and CNN to identify developers that should be assigned to fix a bug report [16].

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

Recent supervised bug localization techniques make use of a training dataset of manually localized bugs to boost performance. Unfortunately, Zimmermann et al. have highlighted that sufficient defect data is often unavailable for many projects and companies [34]. Much defect data is also not clean and suffer from a variety of biases – c.f., [2, 3, 8, 13]. To deal with this limitations, there is a need for a cross-project bug localization solution that can take data from a project to train a model for another project for which only limited clean bug data is available. To address this need, we propose a deep transfer learning approach specialized for bug localization named TRANP-CNN . David says: Please add a sentence that remind readers about the novelty of the approach. Experiments on manually curated datasets by Herzig et al. [8] and Kochhar et al. [13] demonstrated that the proposed approach outperform the state-of-the-art bug localization solution based on deep learning recently proposed by Huo et al. [9] and several other advanced baselines. TRANP-CNN can outperform the best performing baseline by 61.13% to 180.66% considering various standard evaluation metrics.

As a future work, we plan to extend the evaluation of TRANP-CNN by including more bug reports from additional projects (after a manual curation process similar to the ones performed by Herzig et al. and Kochhar et al.). We also plan to develop our solution into a tool that is integrated with an IDE followed by its evaluation by one of our industry partners. We also plan to further improve the performance of TRANP-CNN by considering data beyond text in bug reports and source code files. David says: Xuan, please kindly help to complete the conclusion section too.

Acknowledgement and Replication Package. Acknowledgement and link to replication package are omitted for double blind submission.

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] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar T. Devanbu, and Abraham Bernstein. 2010. The missing links: bugs and bug-fix commits. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. 97–106.

[3] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar T. Devanbu. 2009. Fair and balanced?: bias in bug-fix datasets. 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*. 121–130.

[4] 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.

[5] D Manning Christopher, Raghavan Prabhakar, and Schütze Hinrich. 2008. Introduction to information retrieval. *An Introduction To Information Retrieval* 151 (2008), 177.

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

[7] 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.

[8] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 392–401.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. 177–188.

[16] 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.

[17] 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.

[18] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013).

[19] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer defect learning. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE, 382–391.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*. 273–283.

[27] 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.*

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. 2017. Boosting spectrum-based fault localization using PageRank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*. 261–272.

[33] 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.

[34] 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.