CooBa: Cross-project Bug Localization via Adversarial Transfer Learning

Ziye Zhu[1], **Yun Li**[1*], **Hanghang Tong**[2] and **Yu Wang**[1]
[1]Department of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, China
[2]Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL
{1015041217, liyun, 2017070114}@njupt.edu.cn, htong@illinois.edu

Abstract

Bug localization plays an important role in software quality control. Many supervised machine learning models have been developed based on historical bug-fix information. Despite being successful, these methods often require sufficient historical data (i.e., labels), which is not always available especially for newly developed software projects. In response, cross-project bug localization techniques have recently emerged whose key idea is to transferring knowledge from label-rich source project to locate bugs in the target project. However, a major limitation of these existing techniques lies in that they fail to capture the specificity of each individual project, and are thus prone to negative transfer. To address this issue, we propose an adversarial transfer learning bug localization approach, focusing on *only* transferring the common characteristics (i.e., public information) across projects. Specifically, our approach (CooBa) learns the indicative public information from cross-project bug reports through a shared encoder, and extracts the private information from code files by an individual feature extractor for each project. CooBa further incorporates adversarial learning to ensure that public information shared between multiple projects could be effectively extracted. Extensive experiments on four large-scale real-world data sets demonstrate that the proposed CooBa significantly outperforms the state of the art techniques.

1 Introduction

Bug localization, which aims to locate the corresponding buggy source code files for a given bug report, has been attracting more and more attention in software quality control [Zhou *et al.*, 2012]. Software projects routinely receive a large number of bug reports describing the errors or unexpected results during program operation [Li *et al.*, 2018]. Once a bug report received, programmers can use it to locate the related buggy source code files (referred to 'code files' for short). Many studies have documented the tremendous success of applying supervised machine learning models for bug localization, which are capable of automatically pointing out the related code files with respect to a new bug report [Zhou *et al.*, 2012; Ye *et al.*, 2014; Huo and Li, 2017]. However, much of the work on bug localization only copes with the projects with sufficient historical bug-fix information (i.e., labels) [Zimmermann *et al.*, 2009]. The performance of these techniques could be dramatically affected by the quantity and quality of bug reports and their related code files [Rahman and Roy, 2018]. Unfortunately, in practice, it might be difficult to obtain sufficient bug-fix data. This is especially true for new projects in the first release. Such newly developed projects urgently need an automated bug localization technique to ease the burden on programmers.

One way to deal with the shortage of historical label data is to leverage the knowledge from label-rich projects to locate buggy code files in the current project. Recently, a few works have explored how to transfer available knowledge from label-rich projects [He *et al.*, 2012; Ma *et al.*, 2012; Nam *et al.*, 2017]. For example, Huo et al. [2019] proposed the first cross-project bug localization (**CPBL**) model TRANP-CNN. It first extracted transferable features from bug reports and source code files of source and target projects, then generated project-specific predictions for new bugs by the extracted features. However, a straight-forward application of transfer learning for CPBL is prone to bring the noise into the model, known as the negative transfer effect. For example, the multiple projects in a CPBL task may be produced by different companies (e.g., Microsoft or Mozilla Foundation), developed for different domains (e.g., web browser or mobile application), or designed by different development concept (e.g., large scale development or agile development), etc [Zimmermann *et al.*, 2009]. Therefore, directly transferring knowledge from a label-rich project inevitably brings information that pertains to that specific project (referred to as 'private information' in this paper). Transferring such private characteristic information is likely to hurt, instead of help, the prediction of the buggy code file in another project.

Based on the above observation, we present a new model named CooBa for cross-project bug localization. To be specific, from bug reports, the proposed CooBa learns indicative information about the bug through an encoder that is shared across multiple projects. From code files, the pro-

*Corresponding Author

posed CooBa simultaneously extracts both the public and private information. CooBa leverages an individual Graph Convolutional Networks (GCN)-based feature extractor for each project to extract the private information and a shared Convolutional Neural Network (CNN)-based feature extractor between projects to extract the public information. It further introduces adversarial learning to ensure the public information shared between projects could be effectively extracted. Extensive experiments on large-scale real-world data sets reveal that our model CooBa significantly outperforms state-of-the-art on all evaluation measures.

The main contributions of our work are as follows,

- We present an end-to-end bug localization model called CooBa, to locate the code files that need to be fixed for projects with insufficient historical bug-fix data.
- Our model leverages adversarial transfer learning to transfer public information shared between multiple projects, while preventing the private information to avoid negative transfer.
- We explore the rich structural information of the code file, and extract the private information by a novel multi-layer GCN.

2 Related Work

2.1 Within-project Bug Localization

Most of the existing bug localization models have been evaluated on within-project bug localization (WPBL) settings. As shown in Figure 1(a), each instance representing a pair (b, c), where b is a bug report and c is a code file, with a label (buggy or clean) to indicate whether the code file c is related to the bug report b. In this setting, a supervised model is trained using the fixed bug reports history in Project p. This model is then used to locate the buggy code files that caused the inappropriate bugs described in the upcoming bug reports for the same Project p. Information Retrieval (IR) and machine learning methods are widely used in WPBL. For example, Zhou et al. [2012] proposed the BugLocator based on the *revised Vector Space Model* (rVSM). Kim et al. [2013] treated the bug localization as a classification task and proposed a two-phase prediction model using Naive Bayes. With the advent of deep learning, many researchers presented deep bug localization techniques. For instance, Lam et al. [2017] proposed a bug localization model combining rVSM [Zhou *et al.*, 2012] with Deep Neural Network (DNN). NP-CNN [Huo *et al.*, 2016] and LS-CNN [Huo and Li, 2017] learned a unified feature from natural language and programming language to locate the buggy code files.

2.2 Cross-project Bug Localization

More often than not, software engineers need bug localization beyond the same project. Cross-project bug localization (CPBL) techniques predict buggy code files even for new projects lacking in fixed bug reports history by 'borrowing' information from other projects. As shown in Figure 1(b), a supervised model is trained by the help of labeled instances in Project s (*source* project) and predicts buggy source files for the upcoming bug reports in Project t (*target* project).

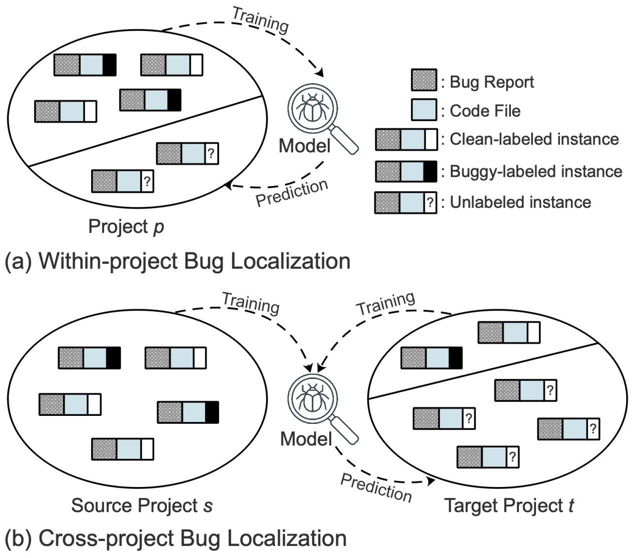

Figure 1: Within-project and Cross-project Bug Localization

Recently, several works have explored how to transfer available knowledge from data-rich projects [He *et al.*, 2012; Ma *et al.*, 2012; Nam *et al.*, 2013]. For example, Turhan et al. [2009] proposed Nearest Neighbor (NN) Filter by selecting similar instances from source project to construct a new training set that is homogeneous with the target project. Nam et al. [2013] proposed another cross-project defect prediction approach, called TCA+, by extending Transfer Component Analysis (TCA) [Pan *et al.*, 2010], which utilized the latent feature space provided by TCA for data of source and target projects. Huo et al. [2019] proposed a cross-project bug localization model TRANP-CNN that combined deep learning and transfer learning. It first extracted transferable potential features from bug reports and source code files for source and target projects, then generated project-specific predictions for new bugs by extracted features.

3 The Proposed Model

In this paper, we focus on cross-project bug localization task. For source project s, we denote $B^s = \{b_1^s, b_2^s, \dots b_{m^s}^s\}$ as the set of bug reports, and $C^s = \{c_1^s, c_2^s, \dots, c_{n^s}^s\}$ as the collection of code files, where the m^s, n^s is the number of the bug reports and code files, respectively. For target project t, we denote $B^t = \{b_1^t, b_2^t, \dots b_{m^t}^t\}$ as the set of bug reports, and $C^t = \{c_1^t, c_2^t, \dots, c_{n^t}^t\}$ as the collection of code files, where the m^t, n^t is the number of the bug reports and code files, respectively. It should be noted that m^t is far less than m^s. Besides, indicator matrices $W^\alpha \in \mathbb{R}^{m^\alpha \times n^\alpha}$ are used to indicate a code file is buggy or clean with respect to a bug report, and $\alpha \in \{s, t\}$. As an example, $W_{i,j}^s = 1$ indicates that code file c_j is a buggy file of the bug report b_i in the source project, and $W_{i,j}^s = 0$ indicates not.

We instantiate the cross-project bug localization as a classification learning task. In the training process, the model aims to learn prediction functions $f^\alpha : B^\alpha \times C^\alpha \to W^\alpha$ by the input pairs (b^α, c^α) from the source and target projects with their labels, $\alpha \in \{s, t\}$. After the model is fully trained,

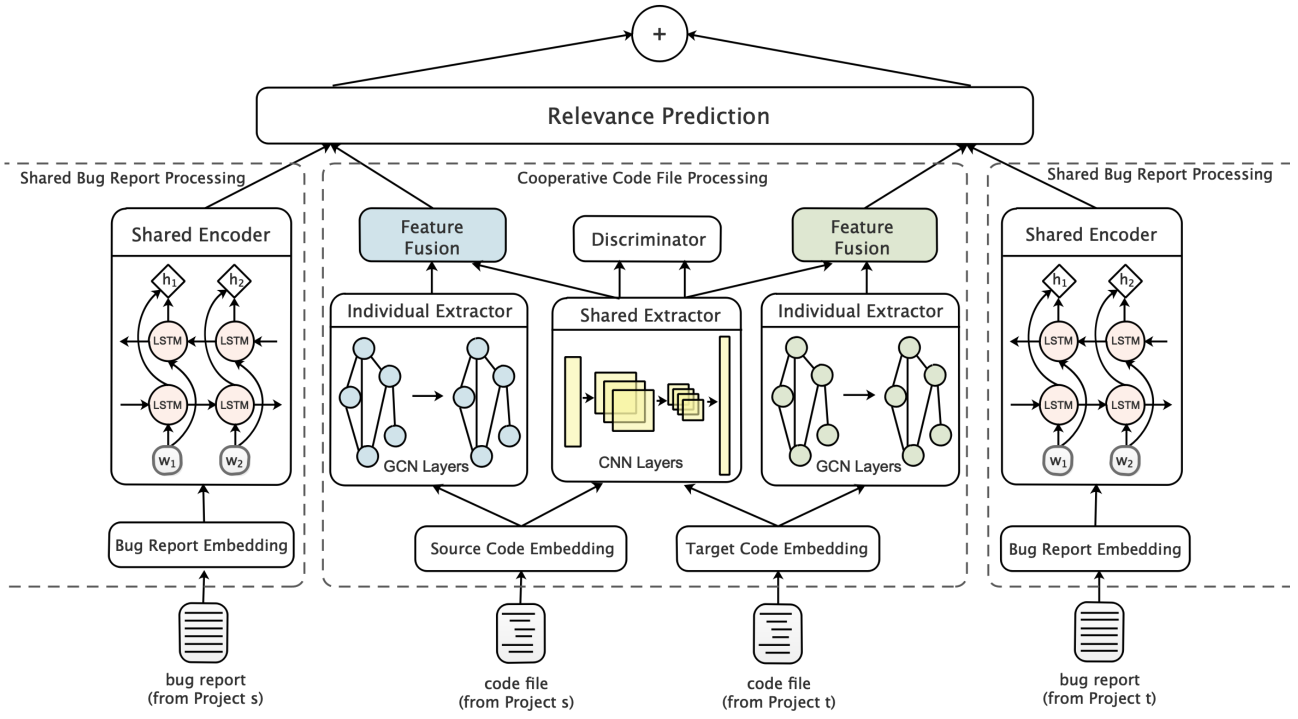

Figure 2: Overall framework of our proposed COOBA, including shared bug report processing, cooperative code file processing, and relevance prediction. Note that the shared bug report processing module located to the left and right of COOBA share parameters.

prediction function f^t is used for determine the relationship of each pair (b^t, c^t) from target project t during testing. The framework of our COOBA is illustrated in Figure 2, which consists of three integral parts: shared bug report processing, cooperative code file processing, and relevance prediction. We will elaborate on each part in the following subsections.

3.1 Shared Bug Report Processing Module

In our task, bug reports submitted to different project groups are written in the same natural language (English) and share the same topic (incorrect or unexpected results during program operating). We take the attitude that sharing bug report processing module between source and target projects in training is beneficial for learning natural language knowledge by one project and subsequently utilized by another one. The shared bug report processing contains an embedding layer and a bug report encoder to extract high-level indicative information about the bugs described in a bug report.

Embedding Layer

Given a bug report, we can extract all sentences from the summary and description items, which are the main part of a bug report, then integrate all sentences into one sequence. We exploit the pre-trained GloVe [Pennington *et al.*, 2014] to map each word of the sequence into a k-dimensional embedding. The input embedding layer represents the bug report b as a embedding sequence $\boldsymbol{w} = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n\}$, n is the number of words in the sequence. It is worth mentioning that the bug report embedding layer is shared for both source and target projects.

Bug Report Encoder

After representing the bug report as an embedding sequence $\boldsymbol{w}$, we exploit a Bidirectional Long Short-Term Memory (Bi-LSTMs) [Schuster and Paliwal, 1997] to encode the input sequence $\boldsymbol{w}$ by considering information from both forward and backward directions. We concatenate the last forward hidden state $\overrightarrow{\boldsymbol{h}}_n$ and the first backward hidden state $\overleftarrow{\boldsymbol{h}}_1$ into a new vector $\boldsymbol{b}$ as follows,

$$\overrightarrow{\boldsymbol{h}}_t = \mathrm{LSTM}_f(\boldsymbol{w}_t, \overrightarrow{\boldsymbol{h}}_{t-1}), \tag{1}$$

$$\overleftarrow{\boldsymbol{h}}_t = \mathrm{LSTM}_b(\boldsymbol{w}_t, \overleftarrow{\boldsymbol{h}}_{t+1}), \tag{2}$$

$$\boldsymbol{b} = \overrightarrow{\boldsymbol{h}}_n \oplus \overleftarrow{\boldsymbol{h}}_1, \tag{3}$$

where the $\boldsymbol{w}_t$ is the input of the Bi-LSTMs at the time step t, and the $\oplus$ is the concatenate operation. After the embedding sequence $\boldsymbol{w}$ passes through the encoder, we obtain the $\boldsymbol{b}$ that is the indicative representation of the bug report. For bug reports from project s or project t, we also share the encoder layer and denote the generated indicative representation as $\boldsymbol{b}^s$ or $\boldsymbol{b}^t$, respectively.

3.2 Cooperative Code File Processing Module

To appropriately transfer knowledge from a label-rich project to a target project, we propose a cooperative code file processing module for extracting shared information of code files belonging to different projects and simultaneously extracting their private information. It is worth mentioning the cooperative nature of our approach. This is crucial because transferring knowledge directly from a source project will likely bring its private information that is not applicable to another (target) project. There are mainly four components in this module: embedding layer, private feature extraction, public feature extraction, and project-specific feature fusion.

Embedding Layer

We exploit graph structure to represent code files, thereby extending the text representation from a sequential (or grid) point of view to a graphical view. For each code file, we obtain a graph $\mathcal{G}$ and a node embedding matrix X from the embedding layer. Specifically, the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is generated by the Abstract Syntax Tree (AST) corresponding to the code file, which is widely used in code analysis and clone detection [Neamtiu *et al.*, 2005]. $\mathcal{V}$ is the node set containing the code tokens in the AST (each node is a code token) and a synthetic root node 0 (represents the start of the program). $\mathcal{E}$ is the edge set containing the links between tokens in the AST. The node embedding matrix $X \in \mathbb{R}^{|\mathcal{V}| \times k}$ contains code token embeddings initialized by GloVe. Each row of X is the embedding of a node (i.e., code token), and k is the dimension of the vector. For code files from source and target project, we set up two code embedding layers with the same structure but no parameters shared between them. Therefore, we obtain the graph $\mathcal{G}^\alpha$ and the node embeddings X^α then feed into the individual and shared feature extractors subsequently, where $\alpha \in \{s, t\}$.

Private Feature Extraction

For private feature extraction from each project, we employ the multi-layer GCN [Kipf and Welling, 2016] with the same structure but no shared parameters as the individual feature extractor. Previous work of GCN has demonstrated the ability to operate directly on a graph and induce the embedding vectors of nodes based on the properties of their neighborhoods [Linmei *et al.*, 2019]. For a code graph $\mathcal{G}$ of code file c, we introduce the $\tilde{A} = A + I$, which is the adjacency matrix of the undirected graph $\mathcal{G}$ with added self-connections, where I is the identity matrix. And the GCN follows the layer-wise propagation rule,

$$H^{l+1} = \sigma(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^l W^l), \tag{4}$$

where $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$, $\sigma(.)$ denotes an activation function. H^{l+1} and W^l are the node hidden representations and the trainable weight matrix in the l-th layer; $H^0 = X$. For a code file from project α, its corresponding individual encoder E^α_{in} generates the private feature $\boldsymbol{g}^\alpha_{pri}$ as follows,

$$\boldsymbol{g}^s_{pri} = E^s_{in}(\mathcal{G}^s, X^s); \boldsymbol{g}^t_{pri} = E^t_{in}(\mathcal{G}^t, X^t), \tag{5}$$

where $\mathcal{G}^\alpha$ and X^α are obtained from code file embedding layer.

Public Feature Extraction

Besides the private features of each project, public feature extraction aims to learn the common feature among different projects. Specifically, the adversarial training is utilized to ensure effective learning public features [Wang *et al.*, 2018a]. The adversarial training strategy [Ganin *et al.*, 2016] normally consists of a generator and a discriminator. In our task, we use a shared feature extractor as the generator, and the discriminator aims to estimate which project the code file comes from.

Shared Feature Extractor. To extract the fundamental information shared in multiple projects, we adopt CNN, which is widely used for text processing and analysis tasks [Kim, 2014], in the shared feature extractor. Many experimental results [Kalchbrenner *et al.*, 2014] have shown that the convolution and pooling operations enable CNN to induce lexical and semantic features over the input sequence. Accordingly, we prefer to choose convolution kernels with small sizes (e.g., 3, 5), especially including convolution kernels of size 1 to extract the lexical and semantic features in code files. For simplicity, we use E_{sh} denote the CNN structure, the input of it is the embedding matrix X^s of source project or the embedding matrix X^t of target project. The extractor generates the public feature of code files as the following equations,

$$\boldsymbol{g}^s_{pub} = E_{sh}(X^s); \boldsymbol{g}^t_{pub} = E_{sh}(X^t). \tag{6}$$

Project Discriminator. We incorporate adversarial training into shared space to guarantee that private features of projects do not exist in shared space. We employ a multi-layer perceptron (MLP) network as a project discriminator to estimate which project the code file comes from. Formally, the project discriminator can be expressed as follow,

$$D(\boldsymbol{g}^\alpha_{pub}) = \text{softmax}(\text{MLP}(\boldsymbol{g}^\alpha_{pub})), \alpha \in \{s, t\}. \tag{7}$$

There is a min-max optimization that the shared encoder generates a representation to mislead the project discriminator and the discriminator tries its best to correctly determine the type of project (source project or target project). The adversarial training process can be formalized as follow,

$$\mathcal{L}^{adv} = \min_{\theta_{sh}} (\max_{\theta_d} \sum_{\alpha} \sum_{t=1}^{T^\alpha} \log D(E_{sh}(X^\alpha_{(i)}))), \tag{8}$$

where θ_{sh} and θ_d denote the trainable parameters of shared feature extractor and project discriminator, respectively. T^α is the number of training instance of project α. $X^\alpha_{(i)}$ is the i-th instance of project α.

Project-specific Feature Fusion

We obtain the private and public representations for each code file by the feature extractors. However, due to the unequal amount of data in the source and target projects, different projects may differ in how their private and public information are correlated. Thus, we design the project-specific feature fusion layer to learn the project-specific correlation patterns considering the private and public features. For a code file from project s or project t, we get a fused representation appropriately merged by a two-layer MLP network as

$$\boldsymbol{c}^s = \text{MLP}^s(\boldsymbol{g}^s_{pri} \oplus \boldsymbol{g}^s_{pub}); \boldsymbol{c}^t = \text{MLP}^t(\boldsymbol{g}^t_{pri} \oplus \boldsymbol{g}^t_{pub}), \tag{9}$$

where the $\oplus$ is the concatenate operation.

3.3 Relevance Prediction Module

The relevance prediction module aims to learn the correlation patterns of bug reports and their related code files by the information obtained from bug report and code file processing module. As the lack of labeled instances, the relevance prediction module cannot perform well on the target project. We take advantage of the fact that bug reports and related code files are correlated in a similar way in both source and target projects, which is all bug reports describe how the product was damaged. Therefore, we construct the shared relevance prediction module across the source and target projects. Specifically, to measure the relevance degree of a (b,c) pair, the relevance metric in the prediction module is defined as

$$F(b^\alpha, c^\alpha) = \|\boldsymbol{b}^\alpha - \boldsymbol{c}^\alpha\|_2^2, \alpha \in \{s, t\}. \tag{10}$$

We encourage the representation of the relevant buggy code file to be as close as possible to the representation of the given bug report. For a bug report b, we use C_+ to denote its related code file set and treat unrelated code file set as C_-. Considering the related pairs (b, c_+) should have lower distance than the unrelated ones (b, c_-), where $c_+ \in C_+$ and $c_- \in C_-$. Thus, the task loss function for each project is

$$\mathcal{L}^\alpha = \sum_{b^\alpha, c_-^\alpha, c_+^\alpha} \max(0, \tau - (F(b^\alpha, c_-^\alpha) - F(b^\alpha, c_+^\alpha))), \tag{11}$$

where $F(.,.)$ is computed by Eq. 10; τ is a margin, forcing $F(b, c_+)$ to be greater than $F(b, c_-)$; $\alpha \in \{s, t\}$.

3.4 Training

Giving the labeled bug-fix data of project s and project t, the final training objective is defined as follow,

$$\mathcal{L} = I(b, c)\mathcal{L}^s + (1 - I(b, c))\mathcal{L}^t + \lambda\mathcal{L}_{adv}, \tag{12}$$

where λ is the hyper-parameter, $\mathcal{L}^s$ and $\mathcal{L}^t$ are computed via Eq. 11, and $\mathcal{L}_{adv}$ is computed via Eq. 8. $I(b, c)$ is an indicator function to identify which project the input pair (b, c) comes from. It is defined as follow,

$$I(b, c) = \begin{cases} 1, & \text{if } (b, c) \in P^s; \\ 0, & \text{if } (b, c) \in P^t, \end{cases} \tag{13}$$

where P^s and P^t are source project corpora and target project corpora, respectively. At each iteration in the training process, we alternately sample a batch of training instances from P^s or P^t to update the parameters. In this way, we adapt Adam [Kingma and Ba, 2014] to directly minimize the final loss function $\mathcal{L}$ in our model.

4 Experimental Results and Analysis

4.1 Experimental Setup

Data sets. We evaluate our method on the data sets provided by Ye et al. [2014]. The data sets contain the bug reports, source code links, buggy files, API documentation, and the oracle of bug-to-file mappings, which are all publicly available[1]. Four open-source projects are collected in these data sets as follows,

- AspectJ: an aspect-oriented programming extension for Java programming language.
- SWT: an open source widget toolkit for Java.
- JDT: a suite of Java development tools for Eclipse.
- Eclipse Platform UI: a user interface of a development platform for Eclipse.

[1]http://dx.doi.org/10.6084/m9.figshare.951967

Comparison methods. We compare CooBa with the following methods, including the conventional models for within-project bug localization and the recent models for cross-project bug localization:

- BugLocator [Zhou *et al.*, 2012]: a classic IR-based bug localization method that consider similar bugs information that have been fixed before.
- DNNLOC [Lam *et al.*, 2017]: a model combining rVSM [Zhou *et al.*, 2012] with DNN while considering the metadata of the bug-fixing history and API elements.
- NP-CNN [Huo *et al.*, 2016]: a deep learning method based on CNN, which leverages both lexical and program structure information from natural language and programming language.
- NN Filter [Turhan *et al.*, 2009]: a cross-project and cross-company defect prediction method by selecting similar instances from source project to construct a training set that is homogeneous with target project.
- TCA+ [Nam *et al.*, 2013]: a cross-project defect prediction approach, which is an extension of TCA by normalization before applying it.
- TRANP-CNN [Huo *et al.*, 2019]: a transfer learning approach for cross-project bug localization by extracting transferable semantic features from the source project.

Among these methods, BugLocator, DNNLOC, and NP-CNN are designed for within-project context. Burak, TCA+, and TRANP-CNN are recent methods for cross-project software mining problems. Followed the work of Huo et al. [2019], we select the cross-project defect prediction models Burak and TCA+ as our comparison methods. In our experiments, we use the same settings suggested in their original works. For bug localization in cross-project context, we suppose the training data includes all the fixed bug reports of source project and 20% fixed bug reports of target project. The remaining 80% bug reports of target project are used for testing. We repeat this experiment for 10 times, and 10 cross-validation is used in the experiment. We evaluate the bug localization performance with two criteria, i.e., Top-10 Rank and Mean Average Precision (MAP), which are widely adopted in bug localization based on historical bug-fix information [Zhou *et al.*, 2012; Wang *et al.*, 2018b; Huo *et al.*, 2019].

4.2 Effectiveness Results

The experimental results of accuracy comparison for different methods are shown in Figure 3 and the best performance of each task is marked with •. The top half of the figure depicted performance w.r.t. Top-10 Rank, and the bottom half w.r.t. MAP. The horizontal coordinate indicates different tasks, for example, "AspectJ $\rightarrow$ JDT" represents using AspectJ as the source project and locates the bugs in JDT. From Figure 3, we can observe: **1) Our proposed CooBa can effectively resolve the cross-project bug localization problem, and outperform the state-of-the-art in all tasks on both metrics.** As we can see, CooBa outperforms all the compared methods. For example, CooBa defeats its best competitor (i.e.,

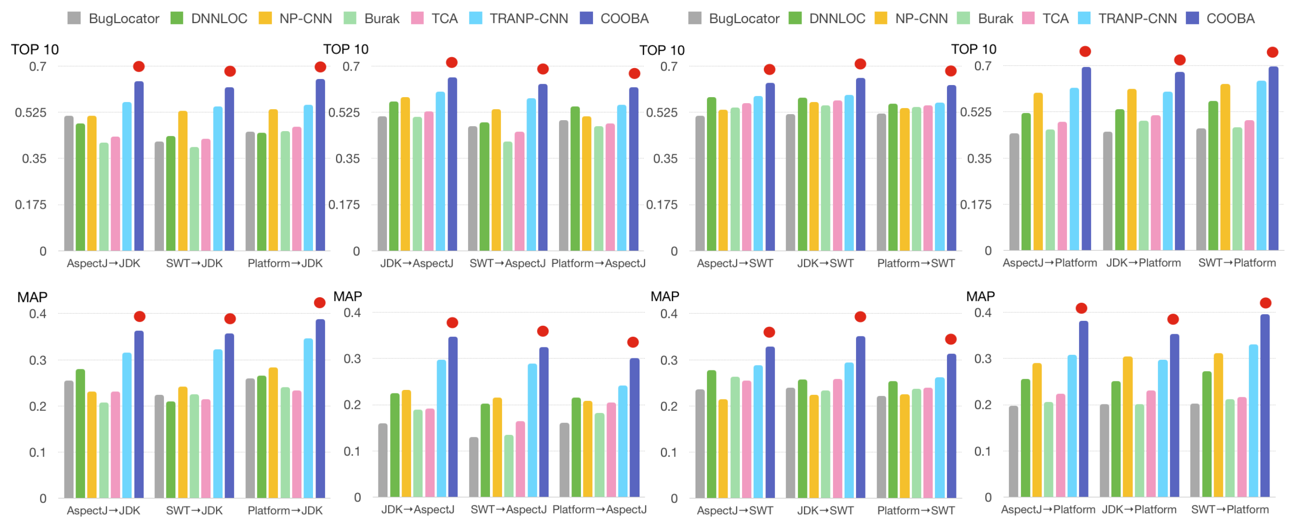

Figure 3: Performance comparisons with bug localization models in terms of Top-10 Rank and MAP.

TRANP-CNN) by 6.5% w.r.t. MAP on project Eclipse Platform UI. It also achieves 4.1%, 4.8%, and 5.2% improvement over its best competitor w.r.t. MAP on JDK, AspectJ, and SWT respectively. In summary, CooBa performs best considering cases comprising of all possible pairs of metrics and tasks. Compared with the average results, CooBa achieves the average top-10 of 0.652 and MAP of 0.351. **2) The conventional within-project bug localization models trained in source data cannot be directly used effectively in target data.** The experimental results show that the performance of all transfer learning bug localization models (i.e.,CooBa and TRANP-CNN) significantly outperforms the within-project bug localization models. For example, CooBa and TRANP-CNN improve NP-CNN by 11.7% and 7.6% w.r.t. MAP on project JDT. They also achieve 10.3% and 4.5% improvement over DNNLOC w.r.t. Top-10 on AspectJ. These results indicate that there is a demand to build a specialized technique for cross-project bug localization, since directly using within-project methods does not produce satisfactory performance.

4.3 Effectiveness of Adversarial Transfer Learning

To investigate the impact of adversarial learning on transferring knowledge from the source project to the target project, we change the model and denote it as CooBa* in this group of experiments. The main differences between CooBa and CooBa* are (1) individual feature extractors and (2) a project discriminator. The CooBa* only use a shared feature extractor to directly transfer knowledge across projects. We compare the results of CooBa with CooBa* on all tasks in Table 1 and find, in terms of MAP, CooBa performs significantly better in all tasks. The results indicate that a straightforward application of transfer learning is prone to negative transfer effect. And the adversarial transfer learning with a project discriminator significantly improves the performance of cross-project bug localization. These results demonstrate that the adversarial transfer learning indeed avoid bringing in the noise that does not pertain to the target project, thereby more effectively and reasonably transferring knowledge from the source project.

Task	A→J	S→J	P→J	J→A	S→A	P→A
CooBa	0.363	0.357	0.388	0.347	0.325	0.301
CooBa*	0.311	0.324	0.349	0.295	0.268	0.255
Task	**A→S**	**J→S**	**P→S**	**A→P**	**J→P**	**S→P**
CooBa	0.329	0.351	0.313	0.382	0.353	0.396
CooBa*	0.277	0.304	0.264	0.331	0.304	0.328

Table 1: MAP of the CooBa and CooBa* on all tasks. In this table, A denotes Aspect, J denotes JDT, S denotes SWT, and P denotes Eclipse Platform UI.

5 Conclusions

Cross-project bug localization is crucial for projects with limited bug-fix information. It can significantly extend the application horizon of bug localization techniques, e.g., locating system bugs in the first release of a new project. To produce a successful cross-project bug localization, we need to transfer knowledge from source project carefully, while avoiding bringing in the noise that does not pertain to the target project. In this paper, we propose a novel model named CooBa leveraging adversarial transfer learning. Experimental results demonstrate the effectiveness of the proposed CooBa. In future work, we plan to explore cross-project bug localization in the context of projects written in different programming languages.

Acknowledgments

This work was partially supported by Natural Science Foundation of China (No.61772284), Postgraduate Research&Practice Innovation Program of Jiangsu Province (SJKY19_0763). And the Hanghang Tong author is partially supported by NSF (1947135, 1715385, and 1939725).

References

[Ganin *et al.*, 2016] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-adversarial training of neural networks. *The Journal of Machine Learning Research*, 17(1):2096–2030, 2016.

[He *et al.*, 2012] Zhimin He, Fengdi Shu, Ye Yang, Mingshu Li, and Qing Wang. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19(2):167–199, 2012.

[Huo and Li, 2017] Xuan Huo and Ming Li. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1909–1915, 2017.

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

[Huo *et al.*, 2019] Xuan Huo, Ferdian Thung, Ming Li, David Lo, and Shu-Ting Shi. Deep transfer bug localization. *IEEE Transactions on Software Engineering*, 2019.

[Kalchbrenner *et al.*, 2014] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.

[Kim *et al.*, 2013] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering*, 39(11):1597–1610, 2013.

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

[Kingma and Ba, 2014] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[Kipf and Welling, 2016] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[Lam *et al.*, 2017] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Bug localization with combination of deep learning and information retrieval. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC)*, pages 218–229, 2017.

[Li *et al.*, 2018] Xiaochen Li, He Jiang, Dong Liu, Zhilei Ren, and Ge Li. Unsupervised deep bug report summarization. In *Proceedings of the 26th Conference on Program Comprehension*, pages 144–155, 2018.

[Linmei *et al.*, 2019] Hu Linmei, Tianchi Yang, Chuan Shi, Houye Ji, and Xiaoli Li. Heterogeneous graph attention networks for semi-supervised short text classification. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4823–4832, 2019.

[Ma *et al.*, 2012] Ying Ma, Guangchun Luo, Xue Zeng, and Aiguo Chen. Transfer learning for cross-company software defect prediction. *Information and Software Technology*, 54(3):248–256, 2012.

[Nam *et al.*, 2013] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. Transfer defect learning. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 382–391. IEEE, 2013.

[Nam *et al.*, 2017] Jaechang Nam, Wei Fu, Sunghun Kim, Tim Menzies, and Lin Tan. Heterogeneous defect prediction. *IEEE Transactions on Software Engineering*, 44(9):874–896, 2017.

[Neamtiu *et al.*, 2005] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, 2005.

[Pan *et al.*, 2010] Sinno Jialin Pan, Ivor W Tsang, James T Kwok, and Qiang Yang. Domain adaptation via transfer component analysis. *IEEE Transactions on Neural Networks*, 22(2):199–210, 2010.

[Pennington *et al.*, 2014] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[Rahman and Roy, 2018] Mohammad Masudur Rahman and Chanchal K Roy. Improving ir-based bug localization with context-aware query reformulation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 621–632, 2018.

[Schuster and Paliwal, 1997] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

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

[Wang *et al.*, 2018a] Xiaozhi Wang, Xu Han, Yankai Lin, Zhiyuan Liu, and Maosong Sun. Adversarial multi-lingual neural relation extraction. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 1156–1166, 2018.

[Wang *et al.*, 2018b] Yaojing Wang, Yuan Yao, Hanghang Tong, Xuan Huo, Min Li, Feng Xu, and Jian Lu. Bug localization via supervised topic modeling. In *Proceedings of the 2018 IEEE International Conference on Data Mining (ICDM)*, pages 607–616. IEEE, 2018.

[Ye *et al.*, 2014] Xin Ye, Razvan Bunescu, and Chang Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 689–699, 2014.

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

[Zimmermann *et al.*, 2009] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 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 symposium on The foundations of software engineering*, pages 91–100, 2009.