Software Visualization and Deep Transfer Learning for Effective Software Defect Prediction

Jinyin Chen¹, Keke Hu¹, Yue Yu², Zhuangzhi Chen¹, Qi Xuan³, Yi Liu⁴,⁵, Vladimir Filkov⁵

¹College of Information Engineering, Zhejiang University of Technology, Hangzhou 310023, China
²College of Computer, National University of Defense Technology, Hefei 230000, China
³Institute of Cyberspace Security, Zhejiang University of Technology, Hangzhou 310023, China
⁴Institute of Process Equipment and Control Engineering, Zhejiang University of Technology, Hangzhou 310023, China
⁵Department of Computer Science, University of California, Davis, CA 95616, USA

ABSTRACT

Software defect prediction aims to automatically locate defective code modules to better focus testing resources and human effort. Typically, software defect prediction pipelines are comprised of two parts: the first extracts program features, like abstract syntax trees, by using external tools, and the second applies machine learning-based classification models to those features in order to predict defective modules. Since such approaches depend on specific feature extraction tools, machine learning classifiers have to be custom-tailored to effectively build most accurate models.

To bridge the gap between deep learning and defect prediction, we propose an end-to-end framework which can directly get prediction results for programs without utilizing feature-extraction tools. To that end, we first visualize programs as images, apply the self-attention mechanism to extract image features, use transfer learning to reduce the difference in sample distributions between projects, and finally feed the image files into a pre-trained, deep learning model for defect prediction. Experiments with 10 open source projects from the PROMISE dataset show that our method can improve cross-project and within-project defect prediction. Our code and data pointers are available at https://zenodo.org/record/3373409#.XV0Oy5Mza35.

KEYWORDS

Cross-project defect prediction, within-project defect prediction, deep transfer learning, self-attention, software visualization

1 INTRODUCTION

Software defect prediction techniques can help software developers locate defective code modules automatically, to save human effort and material resources. Most prediction methods build prediction models based on modules in the source code and historical development data at different levels of modeling, e.g., commit changes, methods, and files [1]. In practice, based on whether the historical training data comes from the same project or not, we distinguish between within-project defect prediction (WPDP) and cross-project defect prediction (CPDP) [2].

A large number of manually designed features, including static code features and process features [3], have been adopted to predict whether a module is defective or not. To improve on those, current software defect prediction studies mainly focus on two main directions: new feature extraction methods and classification methods learned from large-scale datasets. However, defect prediction techniques which feed manually designed features into machine learning algorithms for classification, have some limitations [4]. The required feature engineering is time consuming and requires that special tools be used upstream, such as code complexity analysis, submission log mining, and code structure analysis tools. Consequently, many features can be difficult to capture in some projects. For example, semantic code information, such as the features hidden in abstract syntax trees (ASTs), may not be effectively represented by existing traditional features. In addition to the inconvenience of feature engineering for traditional features, as described by Wan et al. in a recent review of defect prediction [5], semantic information can be more valuable than syntax information to distinguish one code region from another. Thus, while AST-conveyed features can be useful for defect prediction, such approaches are indirect, requiring additional tools in order to build and mine the ASTs. Moreover, in such approaches, the source code is most frequently not used once the ASTs are extracted.

A number of machine learning methods, e.g., support vector machines (SVMs) [6], naïve Bayes (NB) [7], decision trees (DTs) [8], and neural networks (NNs) [9], have been applied to defective module prediction. In particular, recent research has been conclusive that deep learning networks are highly effective in image classification, feature extraction, and knowledge representation in many areas [10–15]. In defect prediction specifically, to better generate semantic features, a state-of-the-art method [16] leveraged deep belief network (DBN) for learning features from token vectors extracted from programs’ ASTs. On this basis, Li et al. [17] and Dam...
As one of the primary areas of interest in software engineering research, defect prediction (DP) has been receiving significant attention [22–25]. A number of studies have focused on manually designing features or producing new combinations of existing features from labeled historical defects. These features are typically fed into a machine learning-based classifier to determine if a file is defective. Commonly used features can be divided into static, e.g., code size and code complexity (e.g., Halstead features [26], McCabe features [27], CK features [28]), and process features, like the behavioral differences of developers in the software development process. Many studies have demonstrated that process features can predict software quality [29]. Moser et al. [24] used authors, past fixes, the number of revisions and ages of files as features to predict defects. Nagappan et al. [30] indicated that code churn was effective for DP. Hassan et al. [31] used entropy of changes to predict defects. Other process features are helpful too, including individual developer characteristics [22, 32] and their collaborations [33–35].

Based on these features, many machine learning models, including SVM [6], NB [7], DT [8], NN [9], etc., have been built for the two different DP tasks: within-project, WPDP, and cross-project, CPDP. For WPDP, the training set and test set come from the same project, while for CPDP, they come from different projects. While WPDP can give better results, it is of limited use in practice as it is often difficult to obtain enough training data for a new project. Some studies have instead used related projects to build prediction models with sufficient historical data, and then used them to predict defects in the target project [36–39]. Panichella et al. [40] proposed an approach named CODEP, which uses a classification model to combine the results of 6 classification algorithms (including logistic regression, radial basis function network, and multi-layer perceptrons) for CPDP. Turhan et al. [41] and Peters et al. [42] used different strategies to select appropriate instances in a source project, based on...
Software Visualization and Deep Transfer Learning for Effective Software Defect Prediction

ICSE 2020, May 23-29, 2020, Seoul, South Korea

nearest neighbor methods, to train the prediction models. Additionally, the application of transfer learning is receiving more attention. Xia et al. [43] proposed an approach named HYDRA to build classifiers using genetic algorithm and ensemble learning. Ma et al. [44] proposed a transfer naive Bayes (TNB) method to assign weights to training instances, and then used them to construct a prediction model. Nam et al. [45] proposed TCA+, which uses TCA [46] and optimizes the normalization to improve CPDP.

Recently, some studies have used deep learning to directly get features from source code for DP. Wang et al. [16] deployed deep belief networks (DBNs) to learn semantic features from token vectors extracted from ASTs automatically, and leveraged the learned semantic features to build machine learning models for DP. Li et al. [17] used convolutional neural networks (CNNs) to generate features from source code and combined CNN learned features with traditional features to further improve upon the prediction. Similarly, Dam et al. [18] proposed a prediction model that takes ASTs representing the source file as input. It used an LSTM architecture to capture long-term dependencies which often exist between code elements.

Our proposed approach differs from the above in that we do not use feature engineering followed by machine learning-based classifiers. We also do not need ASTs to bridge the gap between defect prediction and deep learning. Instead, our approach is based on feeding software visualizations into an automatic, image-based, feature discovery pipeline.

Software visualization (SV) for DP SV has long history in software engineering research and practice [47, 48]. It has been used for visualizing code structure and features [49], code execution [50], and evolution [51], of large codebases in particular [52]. SV has also been applied to bug repositories, where it has been helpful in correlating bugs with code structure [53]. In fact, the way code was visualized in a recent study on malware code visualization [54] has, in part, inspired us in this paper. However, to the best of our knowledge, there is a dearth of applications of SV to software defect prediction in the research literature, perhaps because prior to DL approaches no connection was seen between the two areas.

Deep transfer learning (DTL) Transfer learning (TL) is an important tool that can help when there is insufficient training data in machine learning. It works by transferring information from a source domain to a target domain. The key is to relax the assumption that the training and test data must be independent and identically distributed [55]. This is helpful in areas where it is difficult to get enough training data. In the field of DL, studies have shown that CPDP can achieve better performance with transfer learning [56].

Most TL studies are based on traditional machine learning methods. Recently, deep learning-based transfer learning studies have emerged, called deep transfer learning, DTL. Based on the techniques used in them they can be divided into four categories: network-based, instance-based, mapping-based, and adversarial-based [57].

Instance-based DTL is implemented through a specific weight adjustment strategy. The instances selected from the source domain are assigned weights that complement the training data in the target domain [58–61]. Network-based DTL refers to the use of a partially trained network in the source domain for the deep neural network of the target domain, including its network structure and connecting parameters [62, 63]. Adversarial-based DTL refers to introducing adversarial technology inspired by generative adversarial nets (GAN) to find a transferable representation that is applicable to the source and target domains [64, 65]. Mapping-based DTL refers to mapping instances of the source and target domains to a new data space, where the distributions of the two domains are similar. TCA [46] and TCA-based methods have been widely used in applications of traditional transfer learning [66]. Tzeng et al. [67] used maximum mean discrepancy (MMD) to measure the sample distribution after deep neural network processing and learned domain invariant representations by introducing an adaptation layer and additional domain confusion loss. Long et al. [68] improved previous work by replacing MMD with multiple kernel variant MMD (MK-MMD), originally by [69], and proposed a method called deep adaptation networks (DAN). Long et al. [70] proposed joint maximum mean discrepancy to promote the transfer learning ability of neural networks to adapt to the data distribution of different domains. The Wasserstein’s distance, proposed by Arjovsky et al. [71], has been used as a new distance measure to find a better mapping between domains.

To the best of our knowledge, DTL methods have not yet been used in defect prediction. In this paper, we propose a novel, mapping-based DTL method using a self-attention mechanism for defect prediction.

3 APPROACH

The overall framework of our approach deep transfer learning for defect prediction, DTL-DP, is shown in Figure 2. It is comprised of two stages, (1) source code visualization and (2) modeling with DTL. In the first stage we use a visualization method to convert program files into images. In the second stage, we build a DTL model based on the AlexNet network structure with transfer learning and a self-attention mechanism to construct an end-to-end defect prediction framework. We use that framework to predict if a new instance file is defective or not.

In a nutshell, our approach takes the raw program files of a training and test sets directly as input and generates images from them, which are then used to build the evaluation model for defect prediction. Specifically, since the input data of the network model based on the CNN structure should be in the form of images, we build a mapping to convert the files into images. Then we use the first 5 layers of AlexNet as Feature-Net to generate features from the images. The shallow CNN layers correlate the presence and absence of defects to the overall code structure, gradually deepening the granularity from function/loop bodies to function names, identifiers, etc. These features are then fed into the Attention Layer where they are used in assigning weights, and highlighting features, which are more helpful in the classification. The re-weighted features of the training and test sets are used to calculate MK-MMD, which is used to measure the difference between their distributions, as the MMD loss. After that, the re-weighted features of the training set are entered into the fully connected layers to calculate the cross entropy as the classification loss. The weighted sum of the MMD loss and classification loss is fed back to train the whole network, including Feature-Net, Attention Layer and the fully connected layers. Finally, based on the source code visualization method and the DTL model,
we build, train, and evaluate a defect prediction model. We give the
details in the following.

3.1 Source Code Visualization
Converting code to images retains information and facilitates available
deep neural network (DNN) -based methods to improve defect
prediction performance. Most existing methods ignore associations
between extracted features and the classification task. Our proposed
method forms a unified end-to-end framework of feature extraction,
modeling, and prediction.

Figure 3 shows how we convert each program file into 6 different
images. We call this process source code visualization and the im-
ages produced code images. First, each source file is converted into a
vector of 8-bit unsigned integers corresponding to the ASCII decimal
values of the characters in the source code, e.g., ‘a’ is converted to
97, etc. We then generate an image from that vector, by arranging
its values in rows and columns and interpreting it as a rectangular
image. A straightforward way to visualize the 8-bit values as color
intensities would be as levels of gray, e.g., 0=black, and 255=white.
However, the relatively small size of our original training set means
that if we did that we would end up with a deep model that cannot be
sufficiently well trained. Fortunately, an additional benefit to using
images for training is that we can augment the original data set to
produce a larger data set with the same semantics. Common data
augmentation methods for image data sets include flipping (both
vertically and horizontally), rotating, cropping, translating (moving
along the x or y axis), adding Gaussian noise (distortion of high
frequency features), zooming and scaling. Since the sizes of our pro-
grams are small, and we want to retain the semantic and structural
features in the images, the above image data augmentation methods
could result in data loss.

Instead, we designed a novel, color based augmentation method
to generate 6 color images from each source code file. Namely,
each pixel in a color image can be represented by three primary
color components, or channels: red (R), green (G), and blue (B).
The letters R, G, B, and thus the color channels, can be ordered
in 6 different ways: RBG, RGB, BGR, BRG, GRB and GBR. By
adopting a different order every time, as shown in Figs. 2 and 3, we
generate six different images for each program file. For example,
‘for’ is converted to [102, 111, 114], and the 3 values are filled into
the R, G, and B channels in 6 different ways to obtain 6 differently
colored pixels.

The above method expands our data set six-fold, and because of
the nature of the downstream analysis, the generated samples are
reasonable since the representation is changed only by the order of
the different channels.

Whereas we generated 6 images for each instance in the training
set, we randomly selected an RGB permutation for testing. We found
that it was not necessary to generate all six images for testing because
our experiments on 10 datasets showed that the performance of using
the different permutations was quite comparable. We randomly chose
one to increase the speed in practice. The data augmentation was
used to improve the efficacy of the model.

After the above, for each of the 6 orderings of R, G, B, we obtain
a vector of pixels of length one third the original code file size
(each source code character is assigned to a color channel, and thus
three characters in a row represent a pixel). We then turned those

A CNN model learns the features by convolution. Since we use ImageNet’s pre-trained
AlexNet model, the initial parameters of the convolution kernel are different for each
channel, which means even though byte sequences are the same, the final set of features
obtained by sequentially inputting into the model the different channels is also different.
Table 1: Image Width for Various File Sizes

<table>
<thead>
<tr>
<th>File Size Range</th>
<th>Image Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;10 kB</td>
<td>32</td>
</tr>
<tr>
<td>10 kB - 30 kB</td>
<td>64</td>
</tr>
<tr>
<td>30 kB - 60 kB</td>
<td>128</td>
</tr>
<tr>
<td>60 kB - 100 kB</td>
<td>256</td>
</tr>
<tr>
<td>100 kB - 200 kB</td>
<td>384</td>
</tr>
<tr>
<td>200 kB - 500 kB</td>
<td>512</td>
</tr>
<tr>
<td>500 kB - 1000 kB</td>
<td>768</td>
</tr>
<tr>
<td>&gt;1000kB</td>
<td>1024</td>
</tr>
</tbody>
</table>

Table 1 gives some recommended image widths for different file sizes, based on previous work [72, 73]. We adopt those in this work.

Figure 4: The training and testing process of our approach, DTL-DP, inspired by the original DAN paper [68]

3.2 Modeling with Deep Transfer Learning

Next we describe our approach, DTL-DP. The goal is to learn transferable features across projects, and build a classifier \( y = \phi(x) \) of file defectiveness, with source code supervision, where \( x \) is the representation of the sample and \( y \) is the predicted label. In the defect prediction problem, we are given a source project \( P_s = \{ (x_i^s, y_i^s) \}_{i=1}^{n_s} \) with \( n_s \) labeled instances, and a target project \( P_t = \{ (x_i^t, y_i^t) \}_{i=1}^{n_t} \) with \( n_t \) unlabeled instances. In WPDP, the source project is the previous version of the target project, while in CPDP, the source project is a related project of the target project. The sample distributions \( p \) and \( q \) in the source and target projects are often similar in WPDP, but different in CPDP.

For our deep network architecture we adopt a similar architecture as that of deep adaptation networks (DANs) [68] to capture the semantic and structural information of the source code. To the original DAN model we add an attention layer to further enhance the expressive ability of important features. The overall architecture is illustrated in Figure 4. In particular, our DTL-DP consists of an input layer, five convolution layers \((\text{conv}1-\text{conv}5)\), from AlexNet, an Attention-Layer, and finally four fully-connected hidden layers \((\text{fc}6-\text{fc}9)\), working as a classifier. The structure and parameters of \((\text{conv}1-\text{conv}5)\) and \((\text{fc}6-\text{fc}8)\) are consistent with a DAN. But since defect prediction is a binary classification problem, a fully-connected layer \((\text{fc}9)\) is added to obtain a binary result at the end.

We adopt the defaults for AlexNet, so the input to our DTL-DP must be a 224*224 size image cropped from a three-channel (RGB) image of size 256*256. The code image is placed in the approximate center of the 256*256 image (the determination of the size of each code image is described later, in Sect. 3.3). We note that the code image can be smaller than 224*224 pixels due to the variance in the size of the code files. If this happens the image is padded around with blank (zero valued) pixels. Padding should not negatively effect the qualitative performance of feature detection in the images; in fact for deeper DL architectures like ours, padding has been shown to provide extra contrast to the embedded image for each of the layers as well as buffering against data loss by each layer [74].

Training deep models requires a significant amount of labeled data, which is not available for a new project, so we first pre-trained an AlexNet model on ImageNet 2012. Unlike a DAN, which freezes \( \text{conv}1-\text{conv}3 \) and fine-tunes \( \text{conv}4-\text{conv}5 \), we fine tune all convolution layers \( \text{conv}1-\text{conv}5 \) by taking the parameters of the pre-trained models as initial parameters that we then optimize during the training phase. We do that to minimize the differences between our code images and the actual object images in ImageNet 2012.

In order for DTL-DP to focus on the key features in different defect images, and thus further improve prediction, we employ a self-attention mechanism into our model inspired by the good performance of self-attention in GANs [21]. As shown in the Attention Layer in Figure 4, the attention mechanism makes the feature map generated by layer \( \text{conv}5 \) be the self-attention feature map input to the next layer, \( \text{fc}6 \). Specifically, at first, it linearly maps the input features \( x \) (it is a 1*1 convolution, to compress the number of channels, i.e., \( \text{out}_\text{channels} = \text{in}_\text{channels}/8 \)), and produces \( f(x), g(x) \), and \( h(x) \), where \( f(x) = w_f x, g(x) = w_g x, h(x) = w_h x \). The difference between the three is that the size of \( h(x) \) is still the same as \( x \), but the other two are not. Thus, if the width of \( x \) is \( W \), the height \( H \) and the number of channels \( C \), the size of \( x \) is \([C, N]\), where \( N = W \times H \), the size of \( f(x) \) and \( g(x) \) is \([C/8, N]\), but the size of \( h(x) \) is \([C, N]\). The transposed \( f(x) \) and \( g(x) \) are matrix-multiplied to obtain the autocorrelation in the features, i.e., the relationship of each pixel to all other pixels, where \( S_{ij} = f(x)_i^T g(x)_j \). Then, softmax is applied to the autocorrelation features, \( S \), to get the attention map, comprised of weights with values between 0 and 1:

\[
\alpha_{j,i} = \frac{\exp(S_{ij})}{\sum_{i=1}^{N} \exp(S_{ij})} .
\]
After that, the output of the AttentionLayer is the self-attention feature map \( \sigma = (\sigma_1, \sigma_2, ..., \sigma_j, ..., \sigma_N) \), where

\[
\sigma_j = \sum_{i=1}^{N} \alpha_{ji} h(x_i).
\]

Then, each fully connected layer learns a nonlinear mapping \( h_i = f^i(w_i h_{i-1} + b_i) \), where \( h_i \) is the \( i \)th layer hidden representation of feature \( x_i \), \( w_i \) and \( b_i \) are the weights and biases of the \( i \)th layer, and \( f^i \) is the activation function, using ReLU (\( f^i(o) = \max(0, o) \)) for \( f \in \{6 \} \) and softmax for \( f \in \{9 \} \). If we let \( \Theta \) denote the set of all DTL-DP parameters, the empirical risk of DTL-DP, then, is

\[
\min_{\Theta} \frac{1}{n} \sum_{i=1}^{n} F(\sigma(o_i), y_i)
\]

where \( F \) is the cross-entropy loss function, \( n \) is the number of instances and \( \sigma(o_i) \) is the conditional probability that the DTL-DP assigns \( o_i \) to label \( y_i \). It is used to calculate the final loss in Equation 7.

To make the distribution of the source and target projects similar, the same multi-layer adaptation and multi-kernel MMD strategy as in a DAN [68] are used in our model. The feature mapping function \( \sigma \) is defined as the combination of \( m \) positive semi-definite kernels \( k_u \),

\[
<\sigma(o^s), \sigma(o^t)> = k(o^s, o^t)
\]

\[
k = \sum_{u=1}^{m} \beta_u k_u
\]

where \( \beta \geq 0 \) are the weights of the kernels.

The MK-MMD \( d_k(p, q) \) of the probability distributions \( p \) and \( q \) is defined as the reproducing kernel Hilbert space distance between the mean embedding of \( p \) and \( q \). The square formula of MK-MMD is defined as

\[
d_k^2(p, q) = \| E_p[\sigma(x)] - E_q[\sigma(x)] \|_2^2
\]

We see then that the smaller \( d_k(p, q) \) is, the more similar \( p \) and \( q \) are. If \( d_k \) is 0, the distribution of the target project is the same as that of the source project. So the final loss function for train DPL-DP is

\[
\frac{1}{n} \sum_{i=1}^{n} F(\sigma(o_i), y_i) + \lambda \sum_{l=1}^{l_2} d_k^2(D_{s,l}, D_{t,l})
\]

where \( D_{s,l} \) is the \( l \)th layer hidden representation for the source and target, \( d_k^2(D_{s,l}, D_{t,l}) \) is the MK-MMD between the source and target evaluated on the \( l \)th layer representation, \( \lambda \) is a penalty parameter, and \( l_1 \) and \( l_2 \) are \( f_c \) layers to calculate the MK-MMD between source and target. In our implementation of DTL-DP, we set \( \lambda = 1 \), \( l_1 = 6 \) and \( l_2 = 9 \), as per the original DAN work [68].

### 3.3 Model Sensitivity to Parameter Choices

We have made a number of choices to make our modeling platform work effectively. Here we justify those choices by presenting sensitivity studies. For this analysis we used all the data.

Hyperparameter \( \lambda \) The DTL-DP model has a hyper-parameter \( \lambda \) for the MMD penalty, that adjusts the final loss. We conducted \( \lambda \) parameter sensitivity experiments in both the WPDP and CPDP setting. We fix the other parameters and range \( \lambda \) in \( \{0.01, 0.05, 0.1, 0.2, 0.5, 1, 2, 5 \} \). The results are shown in Figure 5(a). While fairly constant across the range, small variations exist. In CPDP, the F-measure first increases and then decreases along the hyper-parameter \( \lambda \). In WPDP, \( \lambda \) affects the performance of DTL-DP, but in the opposite direction. We chose \( \lambda = 1 \) here.

**Different Color Orders** The selection of the R,G,B permutations in the target project images may also potentially affect the outcome. We performed experiments with all six different orderings. When testing them, we set \( \lambda \) to 1 and changed the image type of the target project to each one in \( \{RGB, RBG, BRG, BGR, GBR, GRB\} \). The results are shown in Figure 5(b). The target images in different color orders have somewhat different performance. But overall the values are close to DTL-DP’s average performance.

**Different Image Widths** To explore the impact of different choices for image widths on the results, we performed additional experiments.

For our code images, 3 bytes of source code are needed to form 1 pixel, each byte for one of the R, G, and B channels. The sizes of the source files are between 0 and 100 kb, corresponding to images with pixel count between 0 and 34, 133 (100*1024/3). The size of the image is \( (\text{width}, \text{height}) \), where \( \text{width} \) is obtained from Table 1 according to the size of the code file \( \text{(size)} \).

\[
\text{height} = \frac{\text{size} \times 1024}{3 \times \text{width}}
\]

For example, the size of image converted from a 20 kb code file is \( (64,107) \).

We experimented with image widths of 1/8, 1/4, 1/2, 2, 4, and 8 times of the recommended widths in Table 1. The larger the multiplier, the wider the image. For images of different widths, we
We conducted experiments to assess the performance of DTL-DP and when training the DTL-DP, code images are generated from the labeled instances in the source project and the unlabeled instances in the target project, and then, are simultaneously input into the model. They share the convolution layers conv1-conv5 and AttentionLayer to extract their respective features, and calculate the MK-MMD between source and target projects in the fully connected layers fc6-fc9. It should be noted that we only calculate cross entropy for the source project, because the target project’s labels are not provided. We use a mini-batch stochastic gradient descent along the loss function in order to train the parameters of the entire model. We train for 500 epochs for each pair of source and target projects, then pick the epoch with the best F-measure (described in section 4.3) from which to read out the parameters of the final model. Finally, the files from the target project that need to be predicted are converted into code images and then input into the trained model for classification.

Overfitting is always a possibility with such pipelines. We moderate it here with our choices of (1) augmenting the dataset by using six color channel permutations, described in section 3.1, (2) selecting a simple AlexNet model structure, and (3) using the ImageNet 2012 pre-trained model to reduce fluctuations.

4 EXPERIMENTAL SETUP
We conducted experiments to assess the performance of DTL-DP and to compare it with existing deep learning-based defect prediction approaches, for both within-project, WPDP, and cross-project, CPDP, defect prediction. We ran experiments on a modern-day Linux server with 3 Titan XP GPUs. Unless otherwise stated, each experiment was run 10 times and the average results are reported.

4.1 Dataset Description
In order to directly compare our work with prior research, we used publicly available data from the PROMISE\(^3\) data repository, which has been widely used in defect prediction work [16–18, 75]. We selected all open source Java projects from this repository, and collected their version numbers, class names, and the buggy label for each file. Based on the version number and class name, we obtained the source code for each file from GitHub\(^4\) and fed it to our end-to-end framework. In total, data for 10 Java projects were collected. Table 2 shows the details of these projects, including project description, versions, the total and average number of files, and the defect rate. It should be noted that the average number of files over all projects ranges between 150 and 1046, and the defect rates of the projects have a minimum value of 13.4% and a maximum value of 49.7%. The number of files in some projects is not sufficient to train deep models, and the classes are imbalanced, thus augmentation is needed, as described above.

\(^3\)http://openscience.us/repo/defect/
\(^4\)https://github.com/apache

4.2 Baseline Comparison Methods
To evaluate the performance of our end-to-end framework DTL-DP for defect prediction, we compare it with the following baseline methods in the WPDP setting:

- **Semantic** [16, 75]: the state-of-the-art method which employs deep belief networks, DBN, on source code to extract semantic features for defect prediction.
- **PROMISE-DP** [16]: a traditional method which builds an alternating decision tree, ADTree, classifier based on the original 20 features of the PROMISE dataset.
- **DP-LSTM** [18]: a long short-term memory, LSTM, based deep neural network model which uses ASTs to represent source files and predict whether the file is defective or not.
- **DP-CNN** [17]: a convolutional neural network, CNN, based model which is seeded by AST-derived numerical vectors to automatically learn semantic and structural features of programs. The CNN-learned features are used to train the final classifier in combination with traditional features.

For the cross-project settings, CPDP, Semantic and PROMISE-DP could not be used directly. Instead, we used the following:

- **DBN-CP** [16, 75]: a variant of Semantic which trains a DBN by using the source project and generates semantic features for both the source and target projects.
- **TCA+ [45]**: the state-of-the-art technique for CPDP.

To obtain the training and test data, we followed the processes established in [16]. For WPDP, we use two consecutive versions of each project listed in Table 2. The older version is used to generate the training data, and the more recent version is used as test data. For CPDP, we pick versions randomly from each project, for 11 target projects. And for each target project, we select 2 source projects that are different from the target projects. We use the same 22 test pairs as in [16]. When implementing the baseline methods, we use the same network architecture and parameter settings as described in the papers that introduced them.

4.3 Performance measures
To evaluate the prediction performance we use the F-measure, a widely adopted metric in the literature [16–18, 43, 45]. The F-measure captures a predictor’s accuracy and combines both precision and recall, for a comprehensive evaluation of predictive performance.

Specifically, a prediction that a file is defective is called a true positive (TP) if the file is in fact defective, and false positive (FP) otherwise. Similarly, a prediction that a file is not defective is a true negative (TN) if the file is in fact not defective, and false negative (FN) otherwise. Then, the precision (P), recall (R), and F-measure are defined as:

\[
P = \frac{TP}{TP + FP} \quad (9)
\]

\[
R = \frac{TP}{TP + FN} \quad (10)
\]

\[
F = \frac{(2 \times P \times R)}{(P + R)} \quad (11)
\]

5 RESULTS
This section discusses our results of comparing DTL-DP to baseline tools for defect prediction.
We compare DTL-DP to 4 baseline approaches, representing two deep learning-type methods, based on extracting features from AST. The highest F-measure values of the 5 methods are shown in bold. Since the methods based on deep learning include some randomness, we run DTL-DP, Semantic, DP-LSTM and DP-CNN 10 times for each experiment. On average, the F-measure of our approach is 0.642, and the PROMISE-DP, Semantic, DP-LSTM and DP-CNN achieve 0.499, 0.641, 0.515 and 0.570, respectively. The results demonstrate that our approach is competitive, and may improve on defect prediction compared to PROMISE-DP, DP-LSTM and DP-CNN. The results of Semantic and our approach are similar.

5.1 RQ1: How does DTL-DP compare to feature-based machine learning methods and AST-based deep learning methods, in WPDP?

We compare DTL-DP to 4 baseline approaches, representing two different kinds of defect prediction methods. PROMISE-DP is the baseline representative of traditional feature-based machine learning methods. Semantic, DP-LSTM and DP-CNN are the baselines for deep learning-type methods, based on extracting features from AST. Guided by prior work, we conducted 16 sets of WPDP experiments, each using two versions of the same project. The older version is used to train the prediction model, and the newer version is used to evaluate the trained model.

Table 3 shows the F-measure values for the within-project, WPDP, defect prediction experiments. The highest F-measure values of the 5 methods are shown in bold. Since the methods based on deep learning include some randomness, we run DTL-DP, Semantic, DP-LSTM and DP-CNN 10 times for each experiment. On average, the F-measure of our approach is 0.642, and the PROMISE-DP, Semantic, DP-LSTM and DP-CNN achieve 0.499, 0.641, 0.515 and 0.570, respectively. The results demonstrate that our approach is competitive, and may improve on defect prediction compared to PROMISE-DP, DP-LSTM and DP-CNN. The results of Semantic and our approach are similar.

5.1.1 Case Study: WPDP Discrimination of DTL-DP. t-SNE is a non-linear dimensional reduction algorithm that is effective in visualizing similarities and helping identify clusters in complex data sets [76]. To give insight in the performance of DTL-DP, we demonstrate feature transferability by showing t-SNE embeddings in Figure 6. The blue points are non-defective files, and the red are defective ones. We observe the following: (1) The target instances are not discriminated very well, using either the traditional manual features or the TCA+ improved manual features or the semantic features extracted from ASTs, while with our approach, the points are discriminated much better. (2) With the other three approaches the categories between source and target projects are not well aligned, while with our approach, the categories between the projects are more consistent. These conclusions are derived from the intra- and inter-class distances of the two categories in Figures 6 and 7. They are visually apparent.

Our method did not perform as well as the comparison methods on some projects, such as ant, which is likely caused by large variance in file sizes. While the sizes of ant-1.5 and ant-1.6 are close, there is a marked difference between ant-1.6 and ant-1.7, the former being much smaller than the latter. From Table 3, the performance of our method on ant1.5->ant1.6 is better than that on ant1.6->ant1.7. On the contrary, baseline methods other than DP-LSTM perform better for the task ant1.6->ant1.7, notably, Semantic is dominant, indicating that the semantic feature-based method is more robust to file size variability. Moreover, the ant dataset has two shortcomings, the first is that the amount of data is small, cf. Table 2, where the average amount is only 488 files in one project. The second is that

\[ \text{Average} = \frac{1}{1} \times (64.1 + 49.9 + 51.5 + 57.0) = 64.2 \]
the classes are unbalanced, and the proportion of defective files is 13.4%, which is the least of all our projects. This makes training of the deep model more difficult, which leads to the poorer performance of our method on some projects.

Table 4: F-measure of DTL-DP, DBN-CP (DBN), TCA+, DP-LSTM (LSTM) and DP-CNN (CNN) in CPDP

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
<th>DBN</th>
<th>TCA+</th>
<th>LSTM</th>
<th>CNN</th>
<th>DTL-DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>ant1.6</td>
<td>camel1.4</td>
<td>31.6</td>
<td>29.2</td>
<td>32.1</td>
<td>32.3</td>
<td>39.5</td>
</tr>
<tr>
<td>jEdit4.1</td>
<td>camel1.4</td>
<td>69.3</td>
<td>33.0</td>
<td>31.8</td>
<td>65.1</td>
<td>40.7</td>
</tr>
<tr>
<td>camel1.4</td>
<td>ant1.6</td>
<td>97.9</td>
<td>61.6</td>
<td>44.8</td>
<td>60.7</td>
<td>59.1</td>
</tr>
<tr>
<td>poi3.0</td>
<td>ant1.6</td>
<td>47.8</td>
<td>59.8</td>
<td>38.6</td>
<td>53.2</td>
<td>69.3</td>
</tr>
<tr>
<td>camel1.4</td>
<td>jEdit4.1</td>
<td>61.5</td>
<td>53.7</td>
<td>39.4</td>
<td>54.7</td>
<td>53.1</td>
</tr>
<tr>
<td>log4j1.1</td>
<td>jEdit4.1</td>
<td>50.3</td>
<td>41.9</td>
<td>38.9</td>
<td>42.3</td>
<td>63.9</td>
</tr>
<tr>
<td>lucene2.2</td>
<td>log4j1.1</td>
<td>64.5</td>
<td>57.4</td>
<td>57.4</td>
<td>65.6</td>
<td>78.3</td>
</tr>
<tr>
<td>lucene2.2</td>
<td>xalan2.5</td>
<td>61.8</td>
<td>57.1</td>
<td>57.8</td>
<td>63.2</td>
<td>79.4</td>
</tr>
<tr>
<td>xalan2.5</td>
<td>lucene2.2</td>
<td>59.4</td>
<td>56.1</td>
<td>75.0</td>
<td>62.1</td>
<td>78.3</td>
</tr>
<tr>
<td>log4j1.1</td>
<td>lucene2.2</td>
<td>69.2</td>
<td>52.4</td>
<td>75.0</td>
<td>66.3</td>
<td>76.9</td>
</tr>
<tr>
<td>xalan2.5</td>
<td>xerces1.3</td>
<td>38.6</td>
<td>39.4</td>
<td>34.0</td>
<td>39.1</td>
<td>40.0</td>
</tr>
<tr>
<td>xerces1.3</td>
<td>ivy2.0</td>
<td>42.6</td>
<td>39.8</td>
<td>26.1</td>
<td>42.1</td>
<td>42.0</td>
</tr>
<tr>
<td>synapse1.2</td>
<td>ivy2.0</td>
<td>82.4</td>
<td>83.8</td>
<td>26.1</td>
<td>37.1</td>
<td>49.4</td>
</tr>
<tr>
<td>ivy1.4</td>
<td>synapse1.1</td>
<td>48.9</td>
<td>34.8</td>
<td>45.1</td>
<td>49.1</td>
<td>54.5</td>
</tr>
<tr>
<td>poi2.5</td>
<td>synapse1.1</td>
<td>42.5</td>
<td>37.6</td>
<td>43.5</td>
<td>43.6</td>
<td>59.7</td>
</tr>
<tr>
<td>ivy2.0</td>
<td>synapse1.2</td>
<td>43.3</td>
<td>57.0</td>
<td>53.0</td>
<td>45.6</td>
<td>62.0</td>
</tr>
<tr>
<td>poi3.0</td>
<td>synapse1.2</td>
<td>51.4</td>
<td>54.2</td>
<td>50.3</td>
<td>53.2</td>
<td>62.3</td>
</tr>
<tr>
<td>synapse1.2</td>
<td>poi3.0</td>
<td>66.1</td>
<td>65.1</td>
<td>78.5</td>
<td>67.1</td>
<td>82.7</td>
</tr>
<tr>
<td>ant1.6</td>
<td>poi3.0</td>
<td>61.9</td>
<td>34.3</td>
<td>78.5</td>
<td>62.7</td>
<td>82.7</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>56.8</td>
<td>47.9</td>
<td>49.5</td>
<td>52.8</td>
<td>61.8</td>
</tr>
</tbody>
</table>

5.2 RQ2: How does DTL-DP compare to feature-based Machine Learning and AST-based deep learning methods, in CPDP?

Here we compare to TCA+ and DBN-CP, instead of PROMISE and Semantic, as the baseline approaches, as explained in Sect. 4.2. Again, guided by prior work, we conducted 22 sets of CPDP experiments. In each we randomly select two project versions from two different projects, one as a training set and the other as a test set.

Table 4 presents the F-measure results of DTL-CP and the 4 baseline approaches. The highest F-measure values are in bold. On average, the F-measure of our approach in CPDP is 0.618, and the DBN-CP, TCA+, DP-LSTM and DP-CNN achieve 0.568, 0.479, 0.495 and 0.528. Thus, DTL-CP outperforms them by 8.8%, 29.0%, 24.8% and 15.5%, respectively. In addition, we found that for projects log4j1.1 and poi3.0 our method does better than the corresponding WPDP best performing method.

5.2.1 Case Study: CPDP Discrimination of DTL-DP. Our method is more obviously dominant in CPDP than in WPDP. A possible reason for that is that deep transfer learning makes the distributions of the training and test samples more similar in feature space. Another reason might be the superior ability of deep models to represent features, enabling the model to obtain more transferable features from the images. To gain more insight, we choose the task poi3.0 \(\rightarrow\) ant1.6, and show the t-SNE embeddings in Figure 7. We make similar observations as in the RQ1 case study, that (1) the target instances are more easily discriminated with our approach, and (2) the target instances can be better discriminated with the source classifier. This implies that our approach can learn more transferable features for more effective defect prediction.

5.3 RQ3: How much does each of the three mechanisms, i.e., data augmentation, transfer learning and self-attention mechanism, contribute to DTL-DP's performance?

To find out the specific contributions of the three parts to defect prediction, we conducted further experiments. We built the original AlexNet model for binary classification and use it as the Base. +TL, +Attention and +DataAug are three new baselines built by adding to the base AlexNet one of three mechanisms (transfer learning, self attention and data augmentation), respectively. It should be noted...
that, in addition to +DataAug, we use images generated by one sequence of R, G, and B as training and test sets in each experiment. Therefore, the experimental results of Base, +TL and +Attention in Tables 5 and 6 are average results obtained over the 6 different RGB permutations, as described above.

Table 5: Contributions of the three mechanisms to WPDP

<table>
<thead>
<tr>
<th>Project</th>
<th>Version</th>
<th>Base</th>
<th>+TL</th>
<th>+Atten</th>
<th>+Aug</th>
<th>DTL-DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>ant</td>
<td>1.5→1.6</td>
<td>66.0</td>
<td>67.6</td>
<td>66.4</td>
<td>67.7</td>
<td>70.0</td>
</tr>
<tr>
<td>camel</td>
<td>1.2→1.4</td>
<td>41.5</td>
<td>38.1</td>
<td>41.6</td>
<td>40.3</td>
<td>40.6</td>
</tr>
<tr>
<td>jEdit</td>
<td>3.2→4.0</td>
<td>48.4</td>
<td>50.5</td>
<td>49.7</td>
<td>48.5</td>
<td>49.4</td>
</tr>
<tr>
<td>log4j</td>
<td>1.0→1.1</td>
<td>68.8</td>
<td>67.9</td>
<td>69.2</td>
<td>65.8</td>
<td>68.8</td>
</tr>
<tr>
<td>lucene</td>
<td>2.0→2.2</td>
<td>67.7</td>
<td>73.3</td>
<td>77.8</td>
<td>76.1</td>
<td>78.3</td>
</tr>
<tr>
<td>xalan</td>
<td>2.4→2.5</td>
<td>75.7</td>
<td>78.5</td>
<td>77.5</td>
<td>77.6</td>
<td>79.4</td>
</tr>
<tr>
<td>xerces</td>
<td>1.2→1.3</td>
<td>80.9</td>
<td>82.2</td>
<td>80.0</td>
<td>78.8</td>
<td>82.0</td>
</tr>
<tr>
<td>ivy</td>
<td>1.4→2.0</td>
<td>82.0</td>
<td>82.4</td>
<td>82.7</td>
<td>81.2</td>
<td>82.9</td>
</tr>
<tr>
<td>synapse</td>
<td>1.0→1.1</td>
<td>52.5</td>
<td>48.4</td>
<td>51.1</td>
<td>53.0</td>
<td>54.7</td>
</tr>
<tr>
<td>poi</td>
<td>1.5→2.5</td>
<td>66.3</td>
<td>59.3</td>
<td>65.4</td>
<td>61.4</td>
<td>68.5</td>
</tr>
<tr>
<td>Average</td>
<td>64.1</td>
<td>61.6</td>
<td>62.4</td>
<td>62.3</td>
<td>64.2</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Contributions of the three mechanisms to CPDP

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
<th>Base</th>
<th>+TL</th>
<th>+Atten</th>
<th>+Aug</th>
<th>DTL-DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>ant1.6</td>
<td>camel1.4</td>
<td>38.3</td>
<td>37.9</td>
<td>36.5</td>
<td>37.7</td>
<td>39.5</td>
</tr>
<tr>
<td>jEdit4.1</td>
<td>ant1.6</td>
<td>36.8</td>
<td>39.4</td>
<td>39.5</td>
<td>37.1</td>
<td>40.7</td>
</tr>
<tr>
<td>camel1.4</td>
<td>ant1.6</td>
<td>67.0</td>
<td>66.7</td>
<td>65.3</td>
<td>59.6</td>
<td>59.1</td>
</tr>
<tr>
<td>poi3.0</td>
<td>ant1.6</td>
<td>60.6</td>
<td>62.7</td>
<td>63.4</td>
<td>63.8</td>
<td>69.3</td>
</tr>
<tr>
<td>camel1.4</td>
<td>jEdit4.1</td>
<td>66.3</td>
<td>65.7</td>
<td>65.2</td>
<td>64.7</td>
<td>63.1</td>
</tr>
<tr>
<td>log4j1.1</td>
<td>jEdit4.1</td>
<td>59.9</td>
<td>59.9</td>
<td>59.2</td>
<td>59.4</td>
<td>63.9</td>
</tr>
<tr>
<td>jEdit4.1</td>
<td>log4j1.1</td>
<td>75.4</td>
<td>80.0</td>
<td>77.4</td>
<td>74.6</td>
<td>78.3</td>
</tr>
<tr>
<td>lucene2.2</td>
<td>log4j1.1</td>
<td>57.9</td>
<td>59.0</td>
<td>59.2</td>
<td>59.4</td>
<td>63.9</td>
</tr>
<tr>
<td>xerces2.2</td>
<td>xalan2.5</td>
<td>66.1</td>
<td>66.4</td>
<td>65.0</td>
<td>64.7</td>
<td>68.9</td>
</tr>
<tr>
<td>xerces1.3</td>
<td>xalan2.5</td>
<td>63.8</td>
<td>62.6</td>
<td>66.2</td>
<td>68.6</td>
<td></td>
</tr>
<tr>
<td>xalan2.5</td>
<td>lucene2.2</td>
<td>73.0</td>
<td>75.1</td>
<td>71.0</td>
<td>76.9</td>
<td>78.3</td>
</tr>
<tr>
<td>log4j1.1</td>
<td>lucene2.2</td>
<td>71.3</td>
<td>74.5</td>
<td>74.0</td>
<td>72.7</td>
<td>76.9</td>
</tr>
<tr>
<td>xalan2.5</td>
<td>xerces1.3</td>
<td>38.7</td>
<td>35.5</td>
<td>39.3</td>
<td>37.3</td>
<td>40.0</td>
</tr>
<tr>
<td>ivy2.0</td>
<td>xerces1.3</td>
<td>41.3</td>
<td>40.8</td>
<td>39.8</td>
<td>37.8</td>
<td>42.0</td>
</tr>
<tr>
<td>xerces1.3</td>
<td>ivy2.0</td>
<td>51.8</td>
<td>50.0</td>
<td>50.4</td>
<td>50.0</td>
<td>47.2</td>
</tr>
<tr>
<td>synapse2.2</td>
<td>ivy2.0</td>
<td>52.1</td>
<td>49.0</td>
<td>50.9</td>
<td>47.6</td>
<td>49.4</td>
</tr>
<tr>
<td>ivy1.4</td>
<td>synapse1.1</td>
<td>55.1</td>
<td>58.1</td>
<td>55.3</td>
<td>53.6</td>
<td>54.5</td>
</tr>
<tr>
<td>poi2.5</td>
<td>synapse1.1</td>
<td>50.8</td>
<td>54.1</td>
<td>51.6</td>
<td>54.5</td>
<td>59.7</td>
</tr>
<tr>
<td>ivy2.0</td>
<td>synapse1.2</td>
<td>60.4</td>
<td>62.2</td>
<td>61.9</td>
<td>61.1</td>
<td>62.0</td>
</tr>
<tr>
<td>poi3.0</td>
<td>synapse1.2</td>
<td>60.0</td>
<td>63.2</td>
<td>61.3</td>
<td>61.2</td>
<td>62.3</td>
</tr>
<tr>
<td>synapse1.2</td>
<td>poi3.0</td>
<td>77.0</td>
<td>80.4</td>
<td>78.1</td>
<td>81.7</td>
<td>82.7</td>
</tr>
<tr>
<td>ant1.6</td>
<td>poi3.0</td>
<td>77.7</td>
<td>66.2</td>
<td>70.0</td>
<td>81.2</td>
<td>82.7</td>
</tr>
<tr>
<td>Average</td>
<td>59.4</td>
<td>60.3</td>
<td>59.7</td>
<td>59.9</td>
<td>61.8</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: Time cost of the three mechanisms: defect visualization (Visualiz.), self-attention (Attent.) and transfer learning (TL)

<table>
<thead>
<tr>
<th>Project</th>
<th>TL</th>
<th>Attent.</th>
<th>Visualiz.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ant</td>
<td>45.04</td>
<td>1.31</td>
<td>4.59</td>
</tr>
<tr>
<td>camel</td>
<td>96.55</td>
<td>2.42</td>
<td>7.85</td>
</tr>
<tr>
<td>jEdit</td>
<td>59.15</td>
<td>2.66</td>
<td>4.97</td>
</tr>
<tr>
<td>log4j</td>
<td>13.71</td>
<td>1.31</td>
<td>1.76</td>
</tr>
<tr>
<td>lucene</td>
<td>37.39</td>
<td>1.86</td>
<td>3.54</td>
</tr>
<tr>
<td>xalan</td>
<td>94.44</td>
<td>2.71</td>
<td>8.23</td>
</tr>
<tr>
<td>xerces</td>
<td>51.17</td>
<td>2.14</td>
<td>2.75</td>
</tr>
<tr>
<td>ivy</td>
<td>29.02</td>
<td>1.42</td>
<td>3.05</td>
</tr>
<tr>
<td>synapse</td>
<td>20.42</td>
<td>1.26</td>
<td>2.09</td>
</tr>
<tr>
<td>poi</td>
<td>38.65</td>
<td>1.57</td>
<td>2.21</td>
</tr>
<tr>
<td>Average</td>
<td>48.56</td>
<td>1.87</td>
<td>4.10</td>
</tr>
</tbody>
</table>

To discern the contribution of each mechanism we compare the performance of these baseline methods in WPDP and CPDP. From the results in Table 5 and Table 6, we observe that the three mechanisms each contribute substantially to the accuracy of DTL-DBP, in both WPDP and CPDP. In terms of the F-measure, transfer learning contributes the least improvement, and self-attention and data augmentation contribute similarly to the final result. But in CPDP, transfer learning contributes to the F-measure the most. This is likely because of the applicability of transfer learning in the CPDP setting, and in a way, a validation of the approach.

We also noted the time cost for the 3 mechanisms in our proposed DTL-DBP. Table 7 shows the result. The most time spent during data augmentation is on converting code into images, i.e., source code visualization. For transfer learning, most time is spent on the MK-MMD calculation, and for self-attention, on the calculation of the attention layer in Fig. 4. E.g., for the project ant, Table 3 shows two sets of WPDP experiments, ant 1.5 → 1.6 and ant 1.6 → 1.7. On average, it takes 45.04 seconds, 13.01 seconds and 4.59 seconds for the 3 parts, respectively, for both the training data and the test data. Transfer learning takes the longest time, more than the sum of the other two. This is because of the large number of matrices needed to calculate MMD with the kernel function. The least time cost is incurred by the attention mechanism, and its contribution to WPDP is the largest of the three, i.e., it is most cost-effective.

The three mechanisms all contribute toward the accuracy of our proposed end-to-end framework, in WPDP and CPDP. Self-attention has the greatest contribution to WPDP, and transfer learning contributes most to CPDP.

6 Threats to Validity

Threats to internal validity come from experimental errors and the replication of the baseline methods. In order to compare and analyze the deep learning-based defect prediction techniques, we compare our proposed DTL-DBP method with Sementic, DBN-CP, DP-LSTM and DP-CNN. In addition, our method is also compared with the transfer learning-based method TCA+, which is the state-of-the-art CPDP technique. Since the original implementations were not
available, we re-implemented our own versions of the baselines. Although we strictly follow the procedures described in their work, our new implementations may not completely restore all of their original implementation details. And the randomness of the deep learning-based approach also makes the results of our implemented experiments different from the original. Since we have removed the entries in the PROMISE dataset that cannot retrieve the corresponding source files, our re-implemented experimental results may not be consistent with the original baselines.

The external threat to the validity of the results lies in the generalizability of the results. We have tested our method on 10 open source Java projects, including 14600 files. Defect predictions for instances of other languages, such as C, C++, etc., need to be validated by additional experiments in the future.

Threats to construct validity depend on the appropriateness of the evaluation measurement. The F-measure is used as our main evaluation measure, which has been applied in many previous efforts to evaluate defect prediction comprehensively.

7 CONCLUSIONS AND FUTURE WORK

Here we made two main contributions, code visualization for defect prediction and an improved deep transfer learning model. Our experimental results on 10 open source projects show that deep learning can be effectively applied directly for defect prediction after applying visualization methods to the code. Specifically, our approach, DTL-DP, performs at the top of the range of state-of-the-art WPDP approaches. For CPDP, DTL-DP improves on the state-of-the-art technique TCA+, built on traditional features, by 29.0% (in the F-measure). It also bests the deep learning-based approaches DBN-CP, DP-LSTM and DP-CNN by 8.8%, 24.8% and 15.5%, respectively.

DTL-DP still has some limitations. For some projects, a problem of negative transfer occurs, resulting in a worse prediction than a direct prediction. Large differences between two projects can cause such negative transfer. Reducing the impact of negative transfer is one of the problems to be solved in the future. Additionally, the amount of training data we had was small and class imbalance is one of the problems to be solved in the future. Additionally, the amount of training data we had was small and class imbalance is one of the problems to be solved in the future.

ACKNOWLEDGEMENTS

This work is partially supported by Zhejiang Natural Science Foundation (LY19F020205), National Natural Science Foundation of China (61502423, 61572439, 61702534), Zhejiang University Open Fund (2018KFJJO7), Signal Recognition Based on GAN, Deep Learning for Enhancement Recognition Project, Zhejiang Science and Technology Plan Project (LG18F030009, 2017C33149), and Zhejiang Outstanding Youth Fund (LR19F030001).

REFERENCES
