applied sciences

MDPI

Article

Transfer Convolutional Neural Network for Cross-Project Defect Prediction

Shaojian Qiu [1,2], Hao Xu [1,*], Jiehan Deng [1], Siyu Jiang [3] and Lu Lu [1,4]

[1] School of Computer Science and Engineering, South China University of Technology, Guangzhou 510006, China
[2] College of Software Engineering, South China Agricultural University, Guangzhou 510642, China
[3] School of Software Engineering, South China University of Technology, Guangzhou 510006, China
[4] Modern Industrial Technology Research Institute, South China University of Technology, Zhongshan 528400, China
* Correspondence: xuhao@scut.edu.cn

Received: 23 May 2019; Accepted: 24 June 2019; Published: 29 June 2019

Abstract: Cross-project defect prediction (CPDP) is a practical solution that allows software defect prediction (SDP) to be used earlier in the software lifecycle. With the CPDP technique, the software defect predictor trained by labeled data of mature projects can be applied for the prediction task of a new project. Most previous CPDP approaches ignored the semantic information in the source code, and existing semantic-feature-based SDP methods do not take into account the data distribution divergence between projects. These limitations may weaken defect prediction performance. To solve these problems, we propose a novel approach, the transfer convolutional neural network (TCNN), to mine the transferable semantic (deep-learning (DL)-generated) features for CPDP tasks. Specifically, our approach first parses the source file into integer vectors as the network inputs. Next, to obtain the TCNN model, a matching layer is added into convolutional neural network where the hidden representations of the source and target project-specific data are embedded into a reproducing kernel Hilbert space for distribution matching. By simultaneously minimizing classification error and distribution divergence between projects, the constructed TCNN could extract the transferable DL-generated features. Finally, without losing the information contained in handcrafted features, we combine them with transferable DL-generated features to form the joint features for CPDP performing. Experiments based on 10 benchmark projects (with 90 pairs of CPDP tasks) showed that the proposed TCNN method is superior to the reference methods.

Keywords: cross-project defect prediction; semantic feature learning; transfer learning; maximum mean discrepancy

1. Introduction

As software scale and complexity increase, software testing has become one of the most critical phases in the software lifecycle. Software defect prediction (SDP) based on static code inspection could help the software quality assurance team find defect-prone modules or files, so that they could allocate resources more efficiently. Many existing studies [1–3] proposed various approaches to perform SDP, but they were usually limited to within-project defect prediction (WPDP), which attempts to train prediction models by using historical data to detect undiscovered and future defects in the same project. In the early stages of the software lifecycle, however, with this approach, reliable predictors cannot be built due to the lack of software defect knowledge (e.g., data with a defective or clean label).

To allow SDP to be used in new projects earlier, cross-project defect prediction (CPDP) is proposed as a more practical solution. The CPDP approach aims to transfer defect prediction knowledge from

a mature project (source project with sufficient labeled data) to a new project (target project with no or very limited labeled data) so that the defect predictor trained by the source project can be used to predict whether the files of the target project are defective. The CPDP technique is a promising application of transfer learning [4].

In traditional CPDP methods, handcrafted features are commonly adopted to perform CPDP (e.g., Halstead features based on operators and operands [5], McCabe features based on dependencies [6], and CKfeatures based on the object-oriented concept [7]). In recent years, some researchers [8,9] suggested that the generic convolutional neural network (CNN) and deep belief network (DBN) models could extract semantic and structural features from project programs and applied them to perform SDP for better prediction performance. We call these features deep-learning-generated (DL-generated) features. Inspired by this, we believe that DL-generated features could also be used to improve CPDP performance. However, whether the features extracted from the program can be directly adopted to train the CPDP model requires further exploration.

In a previous study [9], it was assumed that the semantic features extracted by DBN can capture the common characteristics related to defects in the source code, so the features extracted from the source project can be directly applied to the target project. However, due to different scales, functions, and coding rules of software, the data in different projects would show distribution divergence [10,11]. As shown in Figure 1 (left), the classifiers learned in source projects do not necessarily transfer well to the target project when distribution divergence exists. If the distributions of the source and target project could be matched by a transfer learning approach, the new learning data representations would be project invariant [12], which would improve the transferability of the classifier across projects (Figure 1 (right)).

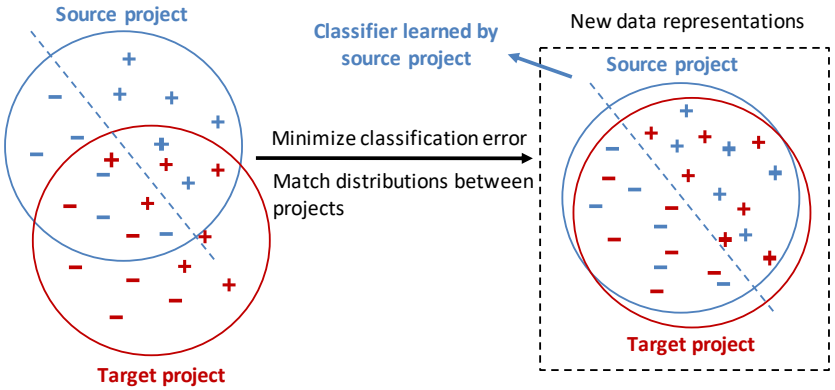


Figure 1. The data in different projects have distribution divergence, so that classifiers learned in a source project do not necessarily transfer well to the target project. If the distributions could be matched, it would enhance the transferability of the cross-project defect predictor.

To handle the impact of semantic features' distribution divergence between projects, we put forward a transfer convolutional neural network (TCNN) model to mine the transferable semantic (DL-generated) features for CPDP tasks. First, our approach parses the source file into integer vectors as the network inputs. Second, to obtain the TCNN model, a matching layer is added into CNN where the hidden representations of the source and target project-specific data are embedded into a reproducing kernel Hilbert space (RKHS) for different distribution matching. By simultaneously minimizing classification error and distribution divergence between projects, the constructed TCNN could extract the transferable DL-generated features. Third, without losing the information contained in handcrafted features, we combine the TCNN-generated features with the handcrafted features processed by transfer component analysis (TCA) to form the transferable joint features. Finally, we feed the joint features into a classifier to conduct CPDP.

The contributions of this paper are summarized as follows.

- In this paper, we proposed a new CPDP approach called TCNN to obtain the transferable semantic (TCNN-generated) features for cross-project prediction. The key improvement is that, considering the data distribution divergence between projects, TCNN transforms CNN by imbedding the representations of project-specific data to an RKHS for distribution matching
- Comprehensive experiments results showed that our TCNN can achieve better prediction performance over classic CPDP methods (e.g., NNFilter [13], data gravitation (DG) [14], TCA+ [10]) and state-of-the-art DL-based approaches (e.g., DBN [9], defect prediction through CNN (DPCNN) [8]) on 90 pairs of CPDP tasks formed by 10 open-source projects.

The rest of the paper is organized as follows. We briefly review the related work in Section 2. We elaborate the TCNN method in Section 3. We present our experimental settings in Section 4 and show the experimental results in Section 5. After that, we discuss why TCNN works, parameter selection, and threats to validity in Section 6. We conclude our work and point out possible future directions in Section 7.

2. Related Work

Most existing SDP studies [1–3] use machine learning techniques to predict defects and evaluate them under a within-project setting where the SDP model is trained and applied in the same project. In practice, we can hardly obtain sufficient training data for a new project, but there is abundant labeled data from public datasets provided by various organizations (e.g., PROMISE [15], AEEEM [16]). Based on these datasets, many CPDP methods have been proposed to train a predictor by using cross-project data.

In early CPDP studies, Zimmermann et al. [17] evaluated CPDP models on 12 real-world applications containing 622 pairs of cross-project prediction tasks. Only 3.4% of tasks achieved performance levels matching those of conventional SDP models indicating that cross-project prediction was still a challenge. He et al. [18] conducted large-scale experiments on 34 datasets and concluded that the CPDP approach based on data selection is comparable with the methods used in training data in the same project. In recent years, various researchers have been trying to enhance CPDP's performance. Ma et al. [14] proposed an algorithm called transfer naive Bayes (TNB) that reweights the instances of the source project by a data gravitation (DG) [19] method and feeds them into a naive Bayes classifier. The DG method can weaken the impact of irrelevant source data to some extent. Considering the data distribution divergence, Nam et al. [10] put forward a transferable feature learning method named TCA+, which extends a classic TCA [20] algorithm with customized normalization rules. In recent years, some researchers have explored the CPDP method with a small ratio of labeled within-project data. Xia et al. [21] proposed a hybrid model reconstruction approach (HYDRA), which first constructed N + 1 genetic algorithm (GA) classifiers by a small ratio of labeled within-project data and a GA step. Then, HYDRA adopted a boosting step to learn a weight for each classifier. In our paper, we focus on methods that only use source project labeled data because they can handle the cold-start issue of software defect data.

In the above CPDP studies, the discriminant features adopted to construct predictive models were elaborately extracted (e.g., Halstead [5], McCabe [6], and CK [7] features). These handcrafted features usually do not contain semantic features of the programs, which may result in poor performance of the SDP model [22]. In recent years, DL has been well studied as one of the techniques for automatic feature generation [22]. In the field of SDP, DL could also be used to extract the semantic features of software programs. To mine the relationship between semantic features and source code, Wang et al. [9] leveraged a powerful DBN to learn the semantic representations from the programs' abstract syntax trees (ASTs). Li et al. [8] attempted to build a framework called defect prediction through CNN (DPCNN) to generate semantic and structural features from programs automatically. It is worth mentioning that DPCNN applies the word embedding technique and combines the CNN-generated features with handcrafted features to enhance SDP's performance. Both [8] and [9] evaluated the methods for SDP tasks, and the results in terms of the F-measure showed that DL-generated features

can contribute to SDP performance. However, whether the DL-generated features produced by DBN and DPCNN can be directly used in the CPDP task needs further exploration. this is exactly what we want to discuss in this paper.

3. Methodology

3.1. Overall Framework

Figure 2 presents the overall framework of the CPDP method we propose. We adjusted the deep semantic feature learning procedure proposed by [9] underpinned by the TCNN building and transferable joint features learning. Specifically, our framework contains four steps: (1) program parsing and vector mapping, (2) TCNN building and DL-generated feature extracting, (3) transferable joint feature constructing, and (4) cross-project prediction performing.

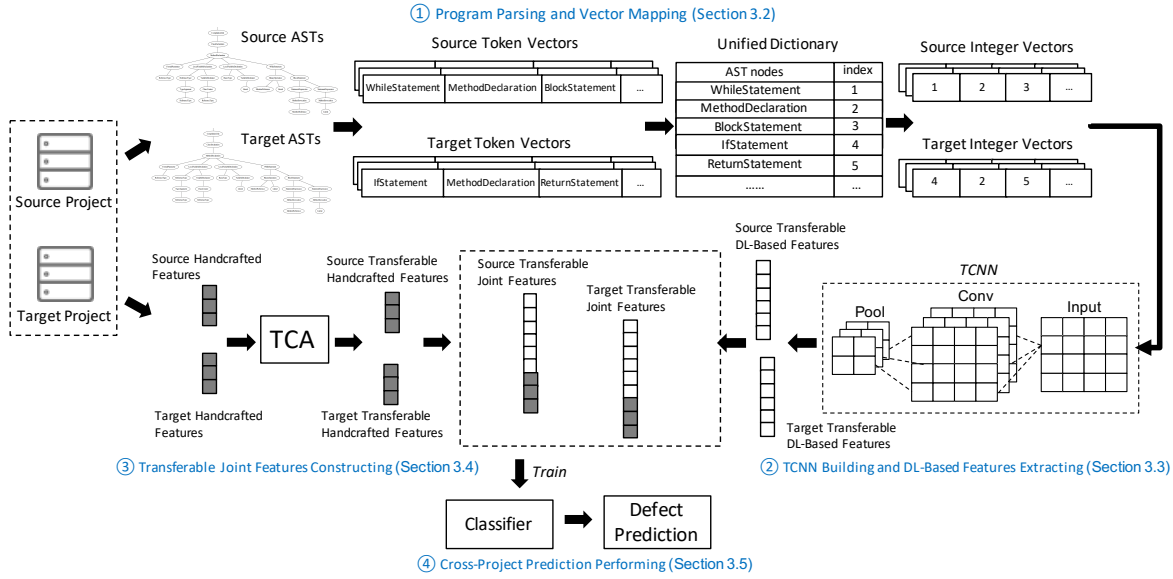


Figure 2. Overview of our TCNN framework to perform CPDP. AST, abstract syntax tree; TCNN, transfer convolutional neural network.

3.2. Program Parsing and Vector Mapping

Our model takes an input of Java files of the source project with known labels (i.e., defective or clean) and unlabeled Java files from the target project. Related work proved that ASTs can be used to detect source code and defects [22], so in this paper, we adopt an AST as the representation of source code. It is worth mentioning that ASTs can not only express the syntax features, but also the semantic features of the source code [8].

In this step, *Javalang* (https://pypi.org/project/javalang/0.9.2/) was applied to parse the Java files and generate corresponding token vectors. Each element in Java files can be represented as a node in the tree. Table 1 shows the categories and types of nodes we used. Referring to [9], considering that the names of methods and variables are usually project-specific, we use node type (e.g., MethodDeclarations and ClassInvocation) to label nodes rather than the specific name. Furthermore, the token vectors extracted by *Javalang* cannot be directly used as the input of our TCNN model, referring to [8], so we created a unified mapping dictionary between the node types and the integers to convert the token vectors to integer vectors. In addition, because CNN requires input vectors to have the same length, all input vectors were padded with zeros to the length of the longest vector.

Table 1. The types of AST nodes we used.

Node Category	Node Type
Nodes of method invocations and instance creations	MethodInvocation, SuperMethodInvocation, ClassCreator
Declaration-related nodes	PackageDeclaration, InterfaceDeclaration, ClassDeclaration, ConstructorDeclaration, MethodDeclaration, VariableDeclarator, FormalParameter
Control-flow-related nodes	IfStatement, ForStatement, WhileStatement, DoStatement, AssertStatement, BreakStatement, ContinueStatement, ReturnStatement, ThrowStatement, TryStatement, SynchronizedStatement, SwitchStatement, BlockStatement, CatchClauseParameter, TryResource, CatchClause, SwitchStatementCase, ForControl, EnhancedForControl
Other nodes	BasicType, MemberReference, ReferenceType, SuperMemberReference, StatementExpression,

3.3. TCNN Building and DL-Generated Feature Extracting

Relying on CNNs' powerful feature generation capabilities, we hope to use it to capture the semantics and local structure of the source code [8]. In this step, we extended the standard CNN architecture to the TCNN with a matching layer to extract transferable DL-generated features for CPDP. In particular, our TCNN consists of an embedded layer, a convolutional layer, a maximum pooling layer, a fully-connected layer, a matching layer, and an output layer with a single unit that ultimately acts as a classifier. The architecture of the TCNN is shown in Figure 3. In addition to using the sigmoid as the activation function of the output layer, all other layers adopted the ReLU as the activation function. We implemented our model by using *Pytorch* (https://pytorch.org), which is an open-source DL platform that provides efficient tools for neural networks' construction and flexible development.

In short, our TCNN model is a variant of the traditional CNN. The key difference between them is that the former adds a matching process after the fully-connected layer to measure the divergence between source and target projects, and the divergence is added to the loss calculation when the network is training. In other words, when CNN is training through labeled integer vectors from the source project, we simultaneously require that the distribution of the source and target projects becomes similar under the hidden representation of the fully-connected layer [12]. Specifically, we aim to optimize two complementary objective functions as follows: (1) minimizing the classification error J_C on the source labeled data and (2) minimizing the distribution divergence J_D between the source and target projects. The final optimization goal is represented as the following formula:

$$\min J_C + \lambda J_D \tag{1}$$

where λ in the formula is a positive regularization parameter.

Regarding classification errors J_C, letting $\Theta = \{W, b\}$ denote the set of all CNN parameters, the empirical risk of CNN is:

$$J_C = \frac{1}{m} \sum_{i=1}^{m} J(\Theta(X_i, y_i)) \tag{2}$$

where $J(\cdot)$ is the cross-entropy loss function, $\Theta(X_i)$ is the conditional probability that the CNN assigns X_i to label y_i, and m represents the number of source project instances.

Regarding the measurement of distribution divergence J_D, we adopted maximum mean discrepancy (MMD) [23,24], which could compare different distributions based on the distance between sample means of two datasets in an RKHS. Given P_s and P_t as the marginal distribution of source and target projects, m and n as the number of source and target project instances, let

$\mathbf{x_s} = \{x_1, \ldots, x_m\} \in \mathbb{R}^{m*d}$ denote the instances of the source project and $\mathbf{x_t} = \{x_{m+1}, \ldots, x_{m+n}\} \in \mathbb{R}^{n*d}$ denote the instances of target projects. The calculation of MMD (also known as J_D) is formulated as:

$$J_D = \text{MMD}(P_s, P_t) = \left\| \frac{1}{m} \sum_{i=1}^{m} \phi(x_i) - \frac{1}{n} \sum_{j=m+1}^{m+n} \phi(x_j) \right\|_{\mathcal{H}}^2 = \frac{1}{m^2} K - \frac{2}{m^2} \kappa^\top + \text{const.} \tag{3}$$

where $\|.\|_{\mathcal{H}}^2$ is the RKHS norm and $\phi(\cdot)$ represents a data matching on the RKHS. In addition, $K_{ij} := k(x_i, x_j), K \in \mathbb{R}^{m*m}$ and $\kappa_i = \frac{n}{m} \Sigma_{j=m+1}^{m+n} k(x_i, x_j)$. In our paper, we used the Gaussian kernel function: $k_{ij} = exp(-\|x_i - x_j\|^2 / 2\sigma)$, and σ is the super parameter of kernel width.

Combining with Formula (1) and Equations (2) and (3), the optimization objective function can be formulated as:

$$\min_{\Theta} \frac{1}{m} \sum_{i=1}^{m} J(\Theta(X_i, y_i)) + \lambda \text{MMD}(P_s, P_t) \tag{4}$$

Based on the above calculation method of the loss function, the transferable features can be extracted through multiple rounds of learning. In this paper, we used the minibatch stochastic gradient descent (SGD) algorithm and Adam optimizer to train the TCNN model.

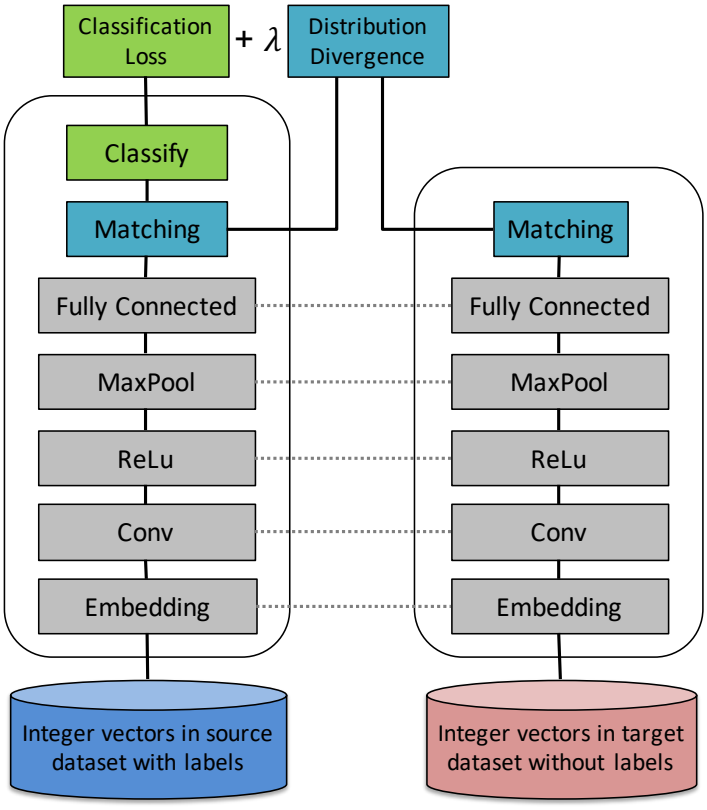


Figure 3. Our model considers both classification loss, as well as data distribution divergence.

3.4. Transferable Joint Feature Constructing

In this study, we hope that the valuable information carried by handcrafted features can be kept, so we combined them with the TCNN-generated features to form the joint features. Considering the transferability of handcrafted features, we adopted the classic distribution adaptation method, TCA [20], to map the handcrafted features into an RKHS and find transferable components by minimizing the MMD. In the subspace spanned by these transferable components, the characteristics

of the source and target data are preserved, and the data distributions in different projects are similar to each other. By using the TCA process, we could obtain the transferable handcrafted features. We further combined it with TCNN-generated features by Python's *concrete* method to obtain transferable joint features as input to Step 4. A detailed description of the TCA algorithm can be seen in [20].

3.5. Cross-Project Prediction Performing

Through the above steps, each file can be represented by corresponding transferable joint features. In this paper, we used logistic regression (LR) as the base classifier. We trained the LR classifier with the data generated by the files and the corresponding labels of the source project. Then, we used the trained model to predict if the files of the target project were defective.

4. Experimental Setup

In this section, we describe the detailed settings for our evaluation experiments.

4.1. Evaluated Datasets

To verify the validity of the TCNN method, we selected 10 open-source projects as our evaluation datasets. The source code and corresponding PROMISE data for all 10 projects are public and have been widely used in SDP research [21,25–27]. In our experiments, we extracted DL-generated features from the Java source code and adopted the static code metrics and data labels from the PROMISE repository.

Table 2 shows the essential information of selected projects, including project name, project version, number of instances, and defect rate (the percentage of defective instances). To guarantee the generality of the evaluation results, the experimental datasets consisted of projects with different sizes and defect rates (the maximum number of files was 1077, and the minimum number of files was 32; the minimum defect rate was 6.3%, and the maximum defect rate was 98.8%). Table 3 shows the static code metrics contained in the PROMISE repository, and for the descriptions, the readers are referred to [21].

Table 2. The 10 projects used in this paper.

Project Name	Project Version	# of Instances	Defect Rate
Camel	1.6	965	19.5%
Forrest	0.8	32	6.3%
Ivy	2.0	352	11.4%
Log4j	1.2	205	92.2 %
Lucene	2.4	340	59.7%
Poi	3.0	1077	63.6%
Synapse	1.2	256	33.6%
Velocity	1.6.1	229	34.1%
Xalan	2.7	909	98.8%
Xerces	1.4.4	588	74.3%

Table 3. List of 20 static metrics of PROMISE. The descriptions were given in [21].

Attribute	Description
dit	The maximum distance from a given class to the root of an inheritance tree
noc	Number of children of a given class in an inheritance tree
cbo	Number of classes that are coupled to a given class
rfc	Number of distinct methods invoked by code in a given class
lcom	Number of method pairs in a class that do not share access to any class attributes
lcom3	Another type of the lcom metric proposed by Henderson–Sellers
npm	Number of public methods in a given class
loc	Number of lines of code in a given class
dam	The ratio of the number of private/protected attributes to the total number of attributes in a given class
moa	Number of attributes in a given class that are of user-defined types
mfa	Number of methods inherited by a given class divided by the total number of methods that can be accessed by the member methods of the given class
cam	The ratio of the sum of the number of different parameter types of every method in a given class to the product of the number of methods in the given class and the number of different method parameter types in the whole class
ic	Number of parent classes that a given class is coupled to
cbm	Total number of new or overwritten methods that all inherited methods in a given class are coupled to
amc	The average size of methods in a given class
ca	Afferent coupling, which measures the number of classes that depend on a given class
ce	Efferent coupling, which measures the number of classes that a given class depends on
max_cc	The maximum McCabe's cyclomatic complexity (CC) score of methods in a given class
avg_cc	The arithmetic mean of McCabe's cyclomatic complexity (CC) scores of methods in a given class

Regarding the composition of the CPDP tasks, we first selected a project as the target project and then respectively used the remaining 9 projects as the source project (e.g., Ivy-2.0 data as a training set, camel-1.6 data as a test set). Thus, 90 test pairs were collected to perform the CPDP.

In fact, the class imbalance problem of software defect data is a typical characteristic in CPDP tasks [28,29]. As shown in Table 2, no matter which project was used as the source projects, the training data had a class imbalance problem. If the classifier was trained on a highly imbalanced dataset, the predictive model tended to support the majority class, and the ability to detect the minority class was weak. For this reason, it was necessary to handle the class imbalance problem. Because class imbalance learning is only used as an application in this paper, we adopted a standard random oversampling to avoid imbalanced data rather than complex approaches in some theoretical approaches [25,29].

4.2. Evaluation Metrics

To compare the predictive performance of different models, we used the F-measure as an evaluation metric [30], which has been widely used in previous CPDP studies. The definition of the F-measure is as follows.

In this paper, we defined the defective file as a positive instance. In contrast, the clean files were defined as negative instances. Using a dichotomous classifier to perform CPDP, we may capture four outcomes of test data: classifying truly defective instances as defective (true positive, TP); classifying clean instances as defective (false positive, FP); classifying truly defective instances as clean (false negative, FN); and classifying truly clean instances as clean (true negative, TN). These four outcomes could construct a confusion matrix (in Table 4) to help us define the F-measure.

Table 4. Confusion matrix.

	Truly Positive	**Truly Negative**
Predictive Positive	TP	FP
Predictive Negative	FN	TN

Based on the confusion matrix in Table 4, we can define the following two interim evaluation metrics.

Recall, also known as the probability of detection, is a measurement of the integrity that defines the probability of the number of TP instances compared with the number of all positive instances:

$$Recall = \frac{TP}{TP + FN}$$

Precision is a measure of exactness that defines the probabilities of the number of TP instances to the number of predictive positive instances:

$$Precision = \frac{TP}{TP + FP}$$

In fact, due to the trade-off between recall and precision, it is difficult to evaluate the models' prediction performance effectively by using only one of them. For this reason, we adopted the F-measure to access the prediction performance, which is the harmonic mean of the precision and recall. The F-measure is the most commonly-used evaluation metric in previous CPDP studies [10,21,25].

$$F - measure = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$$

4.3. Compared Models

To assess the performance of the TCNN model, we explored the following two research questions:

RQ1: Can TCNN perform better than the classic CPDP methods that use only traditional handcrafted features?

RQ2: How effective is the TCNN approach compared to state-of-the-art SDP methods adopting DL-learning features?

Thus, we selected two categories of compared methods in this paper, including the models with only handcrafted or DL-generated features. In summary, we compared our TCNN with LR, NNFilter [13], TNB [14], TCA [20], TCA+ [10], DBN [9], DPDBN [8], CNN [8], and DPCNN [8].

In our experiments, we adopted LR as the base classifier for all models because LR has been widely used in the previous defect prediction studies [8,10,21]. We used the implementation of *LogisticRegression* in *sklearn.linear_model*, and we adopted default parameters settings by *sklearn*. With regard to implementing DL-generated CPDP models, we followed the same code-parsing process

to generate integer vectors for neural networks. Considering that the process of random oversampling involves randomness, we conducted each method 20 times and recorded the average result.

The traditional CPDP models using only the 20 handcrafted features provided by PROMISE include the following methods:

- **LR:** Traditional SDP method, which builds an LR classifier only using handcrafted features.
- **NNFilter:** This method gathers similar instances together to construct a training set that is homogeneous with the target dataset [13].
- **DG:** The predict model is built based on the weighted training data whose weight is calculated by analogy with data gravitation [19].
- **TCA:** A classic transferable feature learning method [20]. We used the source code provided by its author.
- **TCA+.** A variant of TCA [10], which extends TCA with customized normalization rules.

The state-of-the-art SDP models using DL-generated features include the following methods:

- **DBN:** A standard DBN model to extract semantic features for SDP. Regarding the implementation of DBN, we adopted the same network architectures and parameters (i.e., 10 hidden layers and 100 nodes in each hidden layer) as in [9].
- **CNN:** An SDP method that extracts DL-generated features through standard CNN. When implementing CNN, referring to [8], we set the batch size as 32, the epoch number as 15, the embedding dimension as 30, the number of hidden nodes as 100, the number of filters as 10, and the filter length as 5.
- **DPDBN:** A variant of DBN, which concatenates the DBN-learned features with the handcrafted features.
- **DPCNN:** An SDP method that is an improved version of the CNN proposed by [8].

To clearly show the performance differences between methods, we adopted the Scott–Knott test [31] to compare the performance of different CPDP approaches. The Scott–Knott test divides the measurement means into statistically distinct groups by hierarchical clustering analysis. The two main limitations of the traditional Scott–Knott test are: (1) it assumes the data are in a normal distribution; (2) it may create groups that are trivially different from each other. To avoid the limitations of the Scott–Knott test, in this paper, we adopted its normality and effect size-aware variant, the Scott–Knott effect size difference (ESD) test [32,33]. The Scott–Knott ESD test would (1) correct the normal distribution of the input dataset and (2) merge any two statistically different groups of negligible effects. A detailed description of the Scott–Knott ESD test can be found in [32]. We can use the *sk_esd* function of *ScottKnottESD* (https://github.com/klainfo/ScottKnottESD) to make the implementation.

5. Results

Table 5 shows the average performance of each examined CPDP model with a given target project, calculated by the following procedure: (1) Given the target project, we separately evaluated the performance of a CPDP model trained by one source project. Through applying the nine source projects, we could obtain nine prediction models for a given target project. (2) We calculated the average performance of nine models and filled them into the table. The next-to-last row shows the win/tie/loss counts of TCNN versus other approaches.

Figure 4 presents the corresponding Scott–Knott ESD test results (including 90 pairs of CPDP tasks formed by 10 datasets). By comprehensive observation, TCNN outperformed the nine reference methods in our experiments. Following are the answers for two research questions.

Table 5. Comparison results with 10 approaches on 10 target projects. The better F-measures are in bold. DG, data gravitation; DPCNN, defect prediction through CNN.

Target Project	LR	NNFilter	DG	TCA	TCA+	DBN	CNN	DPDBN	DPCNN	TCNN
Camel	0.314	0.324	0.336	0.336	**0.337**	0.315	0.322	0.326	0.333	0.331
Forrest	0.183	0.181	0.143	0.129	0.127	0.129	0.145	0.139	0.185	**0.216**
Ivy	0.272	0.259	0.256	0.266	0.246	0.227	0.255	0.257	0.257	**0.303**
Log4j	0.631	0.643	0.639	0.681	0.676	0.658	0.647	0.687	0.663	**0.702**
Lucene	0.585	0.595	0.640	0.625	0.539	0.585	0.604	0.624	0.622	**0.658**
Poi	0.633	0.625	0.674	0.605	0.598	0.600	0.624	0.651	0.650	**0.686**
Synapse	0.507	0.507	**0.533**	0.528	0.542	0.449	0.490	0.510	0.510	0.528
Velocity	0.498	0.486	0.512	0.501	0.304	0.447	0.477	0.480	0.501	**0.522**
Xalan	0.599	0.611	0.653	0.660	0.667	0.665	0.656	0.683	0.668	**0.701**
Xerces	0.627	0.621	0.620	0.627	0.583	0.580	0.609	0.604	0.601	**0.670**
TCNN: W/T/L	**10**/0/0	**10**/0/1	**8**/0/2	**8**/0/2	**8**/0/2	**10**/0/0	**10**/0/0	**10**/0/0	**9**/0/1	
AVG	0.485	0.485	0.501	0.496	0.462	0.466	0.483	0.496	0.499	**0.532**

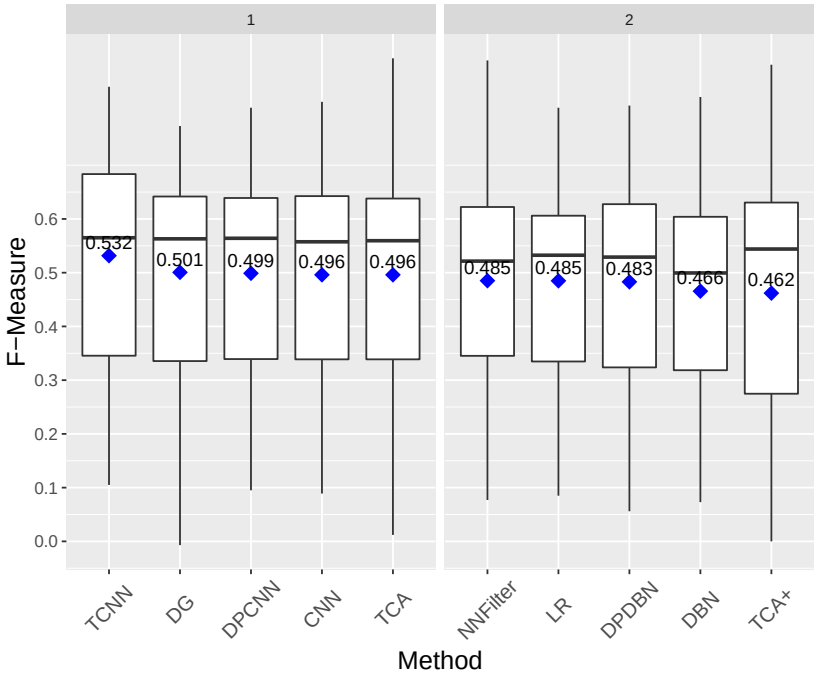


Figure 4. The Scott–Knott effect size difference (ESD) ranking of 10 different methods across the 90 pairs of cross-project defect prediction (CPDP) tasks. The blue diamond indicates the average F-measure of our studied methods.

5.1. ***Answer for RQ1:*** *Can TCNN Perform Better than the Classic CPDP Methods that Use Only Traditional Handcrafted Features?*

In Table 5, the next-to-last row shows that TCNN can win most performance comparisons in 90 combinations of CPDP tasks. The average F-Measure of TCNN was 0.532. Compared to five classic CPDP methods that use only traditional handcrafted features, TCNN outperformed DG, TCA, NNFilter, LR, and TCA+ by 6.2%, 7.2%, 9.5%, 9.6%, and 15.1%, respectively.

The following two points can be observed for this research question:

(1) TCNN, which considers concatenating the DL-generated features with the handcrafted features, could perform better than the CPDP methods that use only traditional handcrafted features.

(2) DG could achieve better performance in terms of average F-measure by comparing with other conventional CPDP methods. However, as shown in Figure 4, the DG model had a lower performance limit in 90 pairs of CPDP tasks.

5.2. ***Answer for RQ2:*** *How Effective Is the TCNN Approach Compared to State-of-the-Art SDP Methods Adopting DL-Learning Features?*

As Table 5 shows, our approach produced the best results in most performance comparisons. Even though the performance of TCNN was not always best, the average F-measure of TCNN was 0.532, which outperformed DPCNN, CNN, DPDBN, and DBN by 6.5%, 6.7%, 10.1%, and 14.2%, respectively.

The following two points can be observed for this research question:

(1) Overall, the methods using CNN-generated features (TCNN, DPCNN, CNN) would yield better predictive performance than DBN-generated features (DPDBN, DBN) in our 90 pairs of CPDP tasks.

(2) TCNN performed better than the SDP methods adopting CNN-generated or DBN-generated features directly.

In summary, the experimental results showed that the proposed TCNN method tended to produce better CPDP performance than the other nine methods we examined. We believe that the performance of CPDP can generally be improved by using the TCNN-generated transferable joint features. This improvement will more effectively help the software quality assurance team find defect-prone modules or files, so the team could allocate resources more efficiently.

6. Discussion

6.1. Why Does TCNN Work?

The experimental results in Section 5 showed that the TCNN method had better predictive performance than methods that directly adopted handcrafted features or DL-generated features in our investigated CPDP tasks. The possible reasons are summarized as follows:

(1) Compared with the CPDP methods based on handcrafted features (e.g., NNFilter, DG, TCA, TCA+), the TCNN model considered the semantic and structural features of the project source code. Our method maps the ASTs into an integer vector and feeds them into the TCNN for transferable features generation. After that, it concatenates DL-generated with the handcrafted features to make sure the defect knowledge carried in the handcrafted and DL-generated features can be utilized simultaneously when training the predictor.

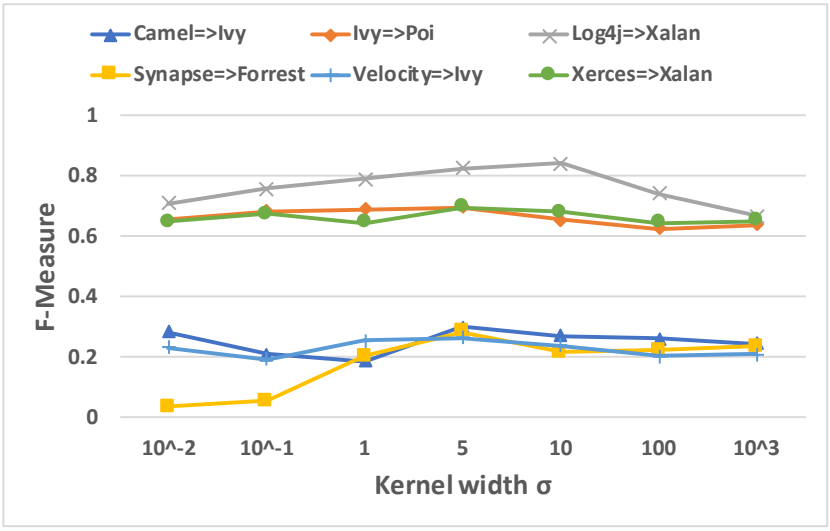
(2) Compared with the previous SDP methods that use DL-generated features (e.g., DBN, DPDBN, CNN, DPCNN), our TCNN model considers the data distribution divergence between the source and target datasets in the CPDP task. By adding the distribution matching layer in the process of training the CNN model, the hidden representations of the source and target project-specific layers are embedded into an RKHS where the different distributions can be matched [34]. The implementation of this matching process is that we adjusted the loss function calculation during TCNN training by merging the classification error with the distribution divergence. Finally, TCNN can obtain transferable DL-generated features for constructing the defect predictor. Although the literature [9] assumes that DBN-generated features can capture the common characteristics of defects between projects, we believe that TCNN-generated features could capture not only the common characteristics, but also the transferable components among different projects.

6.2. Performance under Different TCNN Parameter Settings

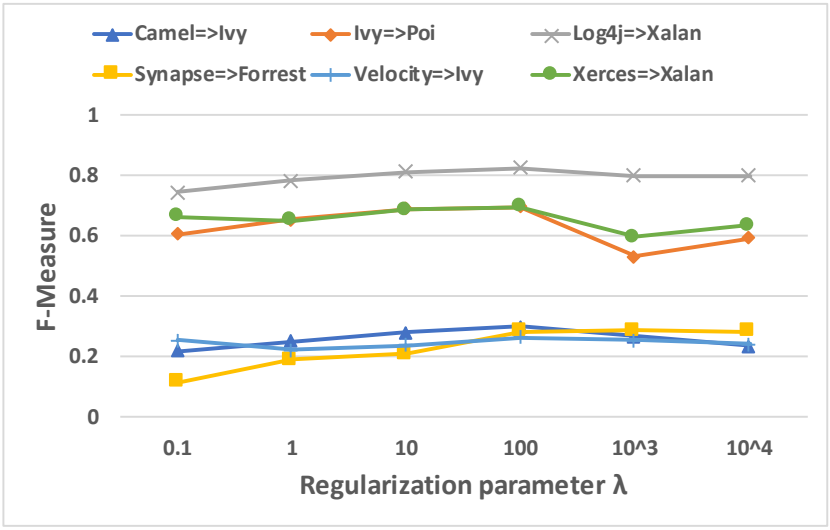
Our approach to TCNN involves some tunable parameters (i.e., the CNN parameters, the Gaussian kernel width σ, and the regularization parameter λ). In our experiments, we used the empirical CNN parameters of [8]. Herein, we focus on the analysis of the parameters σ and λ that affect the performance of TCNN.

The kernel width σ is used to adjust the MMD calculation to measure the distribution divergence. To illustrate the sensitivity of σ, we show the performance of the TCNN with various σ in Figure 5a. Due to space constraints, we only show the performance changes on six randomly-selected pairs of

CPDP tasks. From the results shown in Figure 5a, we finally chose five as the σ value in our experiments for better prediction performance. In our experiments, we selected regularization parameter λ from (0.1, 1, 10, 100, 10^3, 10^4). Similar to the presentation of a σ, we only show several test results of TCNN on the same six pairs of CPDP tasks. From Figure 5b, we can observe that if we set $\lambda = 100$, TCNN performed better than other selections in most CPDP tasks.



(**a**) Different Gaussian kernel width σ



(**b**) Different regularization parameter λ

Figure 5. Performance of TCNN under different parameter settings.

6.3. Time and Memory Overhead of TCNN

In this section, we evaluate the time and memory overhead of TCNN on the six pairs of CPDP tasks used in Section 6.2. The experimental environment was a Windows 10, 64-bit, Intel Xeon 3.9-GHz server with 16 GB RAM. We adopted the NVIDIA GTX1070 to accelerate the training process of the neural network model.

Figure 6 presents the model training time of TCA, TCA+, DBN, CNN, DPDBN, DPCNN, and TCNN on each CPDP task. We did not include LR, NNFilter, and DG in the figure because their training time was less than 100 ms. As shown in Figure 6, the training time of TCNN was reasonable, e.g., on average, it took about 23 seconds to train a model. In summary, TCNN model training time was longer than that of LR, NNFilter, DG, TCA, TCA+, DBN, and DPDBN and was close to that of

CNN and DPCNN. Although it took more time to train TCNN, the prediction performance of TCNN was better than other models. We believe that the training time of TCNN is still acceptable.

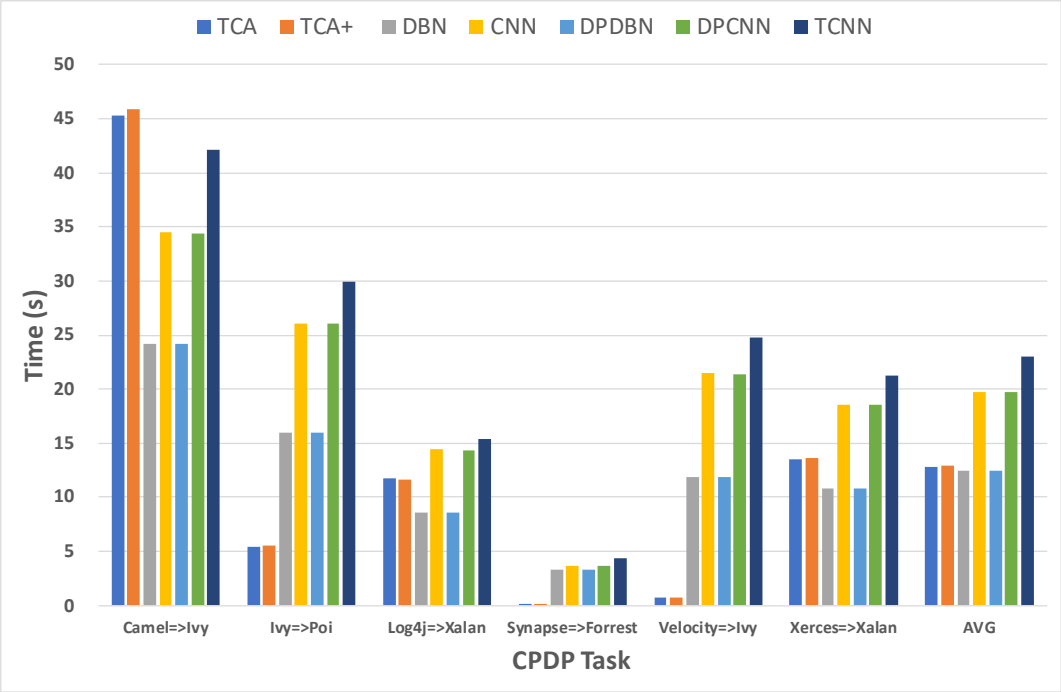


Figure 6. Comparison of model training time.

In addition, in our experiment, it took at most 1.6 GB of memory and 1.9 GB of video memory for the TCNN to generate the DL-generated features for both the training data and the test data. The above executing requirements are relatively easy to achieve, so we believe that our TCNN method is applicable in practice.

6.4. Threats to Validity

6.4.1. Implementation of Compared Methods

In our experiments, we compared our proposed TCNN with 10 referential methods (https://github.com/kevinqiu1990/TCNN). Because the original implementations of NNFilter, DG, TCA+, DBN, and DP-CNN have not yet been publicly released, we re-implemented our version by using *Python*. However, our implementation may not reflect all of the details in the comparison method. To make a fair comparison, we used a consistent LR implementation and oversampling step. When re-implementing DL-based models, we followed the same processes (e.g., code-parsing and vector mapping) and tools (e.g., *Pytorch*).

6.4.2. Experimental Results Might Not Be Generalizable

We conducted our experiments using 10 open-source projects. They might not be representative of all software projects. In addition, we only evaluated the TCNN of the project coded by the Java language. The methods we proposed may produce better or worse results in some commercial software or in those based on other programming languages (e.g., C# or Python).

6.4.3. The F-Measure Might Not Be the Only Appropriate Measures

In this paper, we chose the F-measure as the evaluation metric to compare the predictive power of the CPDP approaches. There are other measures (e.g., AUC and G-measure) that can be used

for performance evaluation of dichotomous classifiers. In fact, the F-measure as a comprehensive measurement is a commonly-used evaluation metric in SDP tasks [21,25,26,35–37].

6.4.4. Parameter Selection Does Not Take All Options into Account

In our experiments, we tried to adjust the parameters of the model to get better prediction performance. However, it is impractical to evaluate all possible combinations of parameters. We evaluated several combinations of parameters within a specific range based on previous research experience [8,22]. There may be a more appropriate combination of parameters for better predictive performance.

7. Conclusions

In this paper, we proposed a novel TCNN approach to perform CPDP that simultaneously considered the transferability of semantic and handcrafted features. Compared with the traditional CPDP studies, our TCNN model not only used handcrafted features, but also considered the semantic and structural features of the project source code. Moreover, based on the previous methods adopting DL-generated features, TCNN employed the matching layer to bridge the source and target datasets to mine the transferable semantic-based features by simultaneously minimizing classification error and distribution divergence between project. The experimental results showed that TCNN outperformed nine referential methods on ten real-world projects with 90 pairs of CPDP tasks.

Several problems remain to be investigated in future work. Firstly, we will try to apply other measurements for the calculation of distribution divergence. Secondly, we will perform more experiments on various SDP datasets to further explore the versatility of our approach.

Author Contributions: Conceptualization, S.Q.; data curation, J.D.; investigation, S.J.; methodology, S.Q., H.X.; supervision, H.X., L.L.; writing, original draft, S.Q.; writing, review and editing, H.X., J.D., S.J., and L.L.

Funding: This work was supported in part by the National Nature Science Foundation of China (No. 61370103), Guangzhou Produce & Research Fund (201902020004) , Zhongshan Produce & Research Fund(2018C1009) and the Fundamental Research Funds for the Central Universities.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Shepperd, M.; Bowes, D.; Hall, T. Researcher bias: The use of machine learning in software defect prediction. *IEEE Trans. Softw. Eng.* **2014**, *40*, 603–616.
2. Laradji, I.H.; Alshayeb, M.; Ghouti, L. Software defect prediction using ensemble learning on selected features. *Inf. Softw. Technol.* **2015**, *58*, 388–402.
3. Song, Q.; Guo, Y.; Shepperd, M. A comprehensive investigation of the role of imbalanced learning for software defect prediction. *IEEE Trans. Softw. Eng.* **2018**, doi:10.13140/RG.2.2.20023.52642.
4. Pan, S.J.; Yang, Q. A survey on transfer learning. *IEEE Trans. Knowl. Data Eng.* **2010**, *22*, 1345–1359.
5. Halstead, M.H. *Elements of Software Science*; Elsevier: New York, NY, USA, 1977; Volume 7.
6. McCabe, T.J. A complexity measure. *IEEE Trans. Softw. Eng.* **1976**, *4*, 308–320.
7. Chidamber, S.R.; Kemerer, C.F. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.* **1994**, *20*, 476–493.
8. Li, J.; He, P.; Zhu, J.; Lyu, M.R. Software defect prediction via convolutional neural network. In Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, Czech Republic, 25–29 July 2017; pp. 318–328.
9. Wang, S.; Liu, T.; Nam, J.; Tan, L. Deep semantic feature learning for software defect prediction. *IEEE Trans. Softw. Eng.* **2018**, doi:10.1109/TSE.2018.2877612.
10. Nam, J.; Pan, S.J.; Kim, S. Transfer defect learning. In Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 18–26 May 2013; pp. 382–391.
11. Herbold, S.; Trautsch, A.; Grabowski, J. A comparative study to benchmark cross-project defect prediction approaches. *IEEE Trans. Softw. Eng.* **2018**, *44*, 811–833.

12. Tzeng, E.; Hoffman, J.; Zhang, N.; Saenko, K.; Darrell, T. Deep domain confusion: Maximizing for domain invariance. *arXiv* **2014**, arXiv:1412.3474.
13. Turhan, B.; Menzies, T.; Bener, A.B.; Stefano, J.D. On the relative value of cross-company and within-company data for defect prediction. *Empir. Softw. Eng.* **2009**, *14*, 540–578.
14. Ma, Y.; Luo, G.; Zeng, X.; Chen, A. Transfer learning for cross-company software defect prediction. *Inf. Softw. Technol.* **2012**, *54*, 248–256.
15. Caglayan, B.; Kocaguneli, E.; Krall, J.; Peters, F.; Turhan, B. *The PROMISE Repository of Empirical Software Engineering Data*; West Virginia University Department of Computer Science: Morgantown, WV, USA, 2012.
16. D'Ambros, M.; Lanza, M.; Robbes, R. An extensive comparison of bug prediction approaches. In Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), Cape Town, South Africa, 2–3 May 2010; pp. 31–41.
17. Zimmermann, T.; Nagappan, N.; Gall, H.; Giger, E.; Murphy, B. 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, Amsterdam, The Netherlands, 24–August 2009; pp. 91–100.
18. He, P.; Li, B.; Liu, X.; Chen, J.; Ma, Y. An empirical study on software defect prediction with a simplified metric set. *Inf. Softw. Technol.* **2015**, *59*, 170–190.
19. Peng, L.; Yang, B.; Chen, Y.; Abraham, A. Data gravitation based classification. *Inf. Sci.* **2009**, *179*, 809–819.
20. Pan, S.J.; Tsang, I.W.; Kwok, J.T.; Yang, Q. Domain adaptation via transfer component analysis. *IEEE Trans. Neural Netw.* **2011**, *22*, 199–210.
21. Xia, X.; Lo, D.; Pan, S.J.; Nagappan, N.; Wang, X. Hydra: Massively compositional model for cross-project defect prediction. *IEEE Trans. Softw. Eng.* **2016**, *42*, 977–998.
22. Wang, S.; Liu, T.; Tan, L. Automatically learning semantic features for defect prediction. In Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), Austin, TX, USA, 14–22 May 2016; pp. 297–308.
23. Borgwardt, K.M.; Gretton, A.; Rasch, M.J.; Kriegel, H.P.; Schölkopf, B.; Smola, A.J. Integrating structured biological data by kernel maximum mean discrepancy. *Bioinformatics* **2006**, *22*, e49–e57.
24. Huang, J.; Gretton, A.; Borgwardt, K.; Schölkopf, B.; Smola, A.J. Correcting sample selection bias by unlabeled data. In Proceedings of the 19th International Conference on Neural Information Processing Systems, Vancouver, BC, Canada, 4–7 December 2006; pp. 601–608.
25. Jing, X.Y.; Wu, F.; Dong, X.; Xu, B. An improved SDA based defect prediction framework for both within-project and cross-project class-imbalance problems. *IEEE Trans. Softw. Eng.* **2017**, *43*, 321–339.
26. Qiu, S.; Lu, L.; Jiang, S. Multiple-components weights model for cross-project software defect prediction. *IET Softw.* **2018**, *12*, 345–355.
27. Zhang, X.; Ben, K.; Zeng, J. Cross-entropy: A new metric for software defect prediction. In Proceedings of the 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), Lisbon, Portugal, 16–20 July 2018; pp. 111–122.
28. Chen, L.; Fang, B.; Shang, Z.; Tang, Y. Negative samples reduction in cross-company software defects prediction. *Inf. Softw. Technol.* **2015**, *62*, 67–77.
29. Qiu, S.; Lu, L.; Jiang, S.; Guo, Y. An investigation of imbalanced ensemble learning methods for cross-project defect prediction. *Int. J. Pattern Recognit. Artif. Intell.* **2019**, doi:10.1142/S0218001419590377.
30. Han, J.; Pei, J.; Kamber, M. *Data Mining: Concepts and Techniques*; Elsevier: Amsterdam, The Netherlands, 2011.
31. Jelihovschi, E.; Faria, J.C.; Allaman, I.B. *The ScottKnott Clustering Algorithm*; Universidade Estadual de Santa Cruz-UESC: Ilheus, Brazil, 2014.
32. Tantithamthavorn, C.; McIntosh, S.; Hassan, A.E.; Matsumoto, K. An empirical comparison of model validation techniques for defect prediction models. *IEEE Trans. Softw. Eng.* **2017**, *43*, 1–18.
33. Herbold, S. Comments on ScottKnottESD in response to "An empirical comparison of model validation techniques for defect prediction models". *IEEE Trans. Softw. Eng.* **2017**, *43*, 1091–1094.
34. Long, M.; Cao, Y.; Wang, J.; Jordan, M.I. Learning transferable features with deep adaptation networks. *arXiv* **2015**, arXiv:1502.02791.
35. Ryu, D.; Choi, O.; Baik, J. Value-cognitive boosting with a support vector machine for cross-project defect prediction. *Empir. Softw. Eng.* **2016**, *21*, 43–71.

36. Feng, Z.; Keivanloo, I.; Ying, Z. Data Transformation in Cross-project Defect Prediction. *Empir. Softw. Eng.* **2017**, *22*, 3186–3218.
37. Tong, H.; Liu, B.; Wang, S. Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. *Inf. Softw. Technol.* **2018**, *96*, 94–111.