Received: 11 November 2022 | Revised: 31 January 2023 | Accepted: 6 February 2023

DOI: 10.1002/cpe.7664

RESEARCH ARTICLE

WILEY

Multi-task deep neural networks for just-in-time software defect prediction on mobile apps

Qiguo Huang | Zhengliang Li | Qing Gu

Department of Computer Science and Technology, Nanjing University, Nanjing, China

Correspondence
Qing Gu, Department of Computer Science and Technology, Nanjing University, Nanjing, China.
Email: guq@nju.edu.cn

Funding information
National Science Foundation of China, Grant/Award Numbers: 61906085, 61972192, 41972111; The Second Tibetan Plateau Scientific Expedition and Research Program, Grant/Award Number: 2019QZKK0204; Collaborative Innovation Center of Novel Software Technology and Industrialization

Summary

With the development of smartphones, mobile applications play an irreplaceable role in our daily life, which characteristics often commit code changes to meet new requirements. This characteristic can introduce defects into the software. To provide immediate feedback to developers, previous researchers began to focus on just-in-time (JIT) software defect prediction techniques. JIT defect prediction aims to determine whether code commits will introduce defects into the software. It contains two scenarios, within-project JIT defect prediction and cross-project JIT defect prediction. Regardless of whether within-project JIT defect prediction or cross-project JIT defect prediction all need to have enough labeled data (within-project JIT defect prediction assumes that have plenty of labeled data from the same project, while cross-project JIT defect prediction assumes that have sufficient labeled data from source projects). However, in practice, both the source and target projects may only have limited labeled data. We propose the MTL-DNN method based on multi-task learning to solve this question. This method contains the data preprocessing layer, input layer, shared layers, task-specific layers, and output layer. Where the common features of multiple related tasks are learned by sharing layers, and the unique features of each task are learned by the task-specific layers. For verifying the effectiveness of the MTL-DNN approach, we evaluate our method on 15 Android mobile apps. The experimental results show that our method significantly outperforms the state-of-the-art single-task deep learning and classical machine learning methods. This result shows that the MTL-DNN method can effectively solve the problem of insufficient labeled training data for source and target projects.

KEYWORDS

Android applications, deep learning, just-in-time, mobile apps, multi-task learning, software defect prediction

1 | INTRODUCTION

Nowadays, mobile apps play an important role in our everyday life, providing users with rich functions, such as online shopping, navigation services, games, and video conferencing.[1-7] With the increase in size and complexity of mobile apps, defects are inevitably introduced during the development process. These defects may have negative impacts on the user experience. The defect-fixing in software defects as early as possible is a challenging task for developers. To solve this, previous studies began to focus on software defect prediction techniques,[8-13] which to predict whether code blocks have defects by building models based on machine learning technology. Software defect prediction is a hot research topic for mobile software quality assurance.[14-19]

Previous studies[14-16] mainly focus on code blocks at methods and classes, which leads to delays in detecting defects. To discover defects in time, researchers[20,21] proposed just-in-time (JIT) software defect prediction, which aims to determine whether a code commit will introduce defects into the software. JIT defect prediction provides developers with immediate feedback on defects, which helps them recheck codes while remaining familiar with codes.

According to the above-mentioned advantages of JIT defect prediction, Catolino et al.[22] was the first to introduce JIT defect prediction into mobile apps and they built a model based on the features of the commit level. Their empirical research results showed that (1) in terms of machine learning algorithms, the classification performance of Naive Bayes is better than that of logistic regression, decision table, and support vector machine; (2) in terms of ensemble methods, boosting is better than random forests, voting and bootstrap aggregating. Recently, Cheng et al. proposed the KAL[23] method for effort-aware JIT defect prediction, they first transformed original features into a high-dimensional feature space and adopted an adversarial learning technique to extract the common features. Experimental results showed that the KAL method achieved better performance than other comparative methods.

In the above studies, within-project defect prediction assumes that there are sufficient labeled training data from the same project,[22] while cross-project defect prediction assumes that there are plenty of labeled training data from the source project.[23] However, in real JIT mobile defect prediction applications, we may only have limited labeled training data from both the source and target projects. Specifically, consider the following scenario: a startup company, which currently has many mobile application projects under development. To identify whether these projects had defects when they were committed, the challenges faced by this company is that the labeled data for all of these projects is very limited. Hence, how to train when both the source and target projects have only limited labeled training data.

We take into account that these projects are all from the same company, thus, it is assumed that some or all of these projects are related to each other. Based on the correlations, this article proposes the MTL-DNN method inspired by multi-task learning.[24] This approach learns the common low-level features of related tasks by sharing hidden layers, and to ensure the uniqueness of tasks, each task has its own unique layer to learn high-level features. Then, we evaluate our method on 15 Apps, the experimental results show that the MTL-DNN approach effectively improves the performance of defect prediction.

This article makes the following contributions:

- To the best of our knowledge, this article first presents a new usage scenario in the field of JIT defect prediction on mobile. In this scenario, both the source and target projects have only limited labeled data.
- To address the problem that both source and target projects have limited labeled data, we propose a new method called MTL-DNN, which learns the common low-level features of related tasks by sharing hidden layers, and to ensure the uniqueness of tasks, each task has its own unique layer to learn high-level features.
- We evaluate our method on 15 Apps, the experimental results show that the proposed MTL-DNN approach significantly outperforms than state-of-the-art single-task deep learning and classical machine learning methods.

The rest of this article is summarized as follows. Section 2 introduces preliminary knowledge of multi-task learning. Section 3 describes our proposed MTL-DNN method in detail. Section 4 presents the experimental setup, including datasets, performance indicators, and statistical tests. Section 5 evaluates the defect prediction performance of the MTL-DNN. Section 6 briefly surveys related work. Section 7 discusses threats to validity and Section 8 summarizes our study.

2 | PRELIMINARY

To better explain our proposed MTL-DNN approach, this section gives the related concept of multi-task deep learning.

Multi-task learning uses the correlation among tasks to improve performance by training.[24,25] The two most commonly used methods of multi-task deep learning are hard parameter sharing and soft parameter sharing.[24] In hard parameter sharing, it is generally applied by sharing the hidden layers between all tasks, while keeping several task-specific output layers. In soft parameter sharing, each task has its own model with its own parameters. Multi-task deep learning framework based on hard parameter sharing is shown in Figure 1.

Given T tasks to be learned, whit each t task associated with a dataset $\mathcal{D}_t = \left\{x_n^t, y_n^t\right\}_{n=1}^{N_t}$ containing N_t samples. Suppose that all tasks have shared layers ω parameterized by $\theta_{\mathcal{E}} = \left\{\theta_{\mathcal{E},1}, \ldots, \theta_{\mathcal{E},L}\right\}$, where $\theta_{\omega,l}$ denotes the parameters in the lth network layer. Each task t has its own task-specific layers F^t parameterized by θ_F^t. Given parameters $\theta = \left(\theta_{\mathcal{E}}, \theta_{\mathcal{F}}^1, \ldots, \theta_{\mathcal{F}}^T\right)$ and sample x_n^t of task t as input, the multi-task network predicts $\hat{y}_n^t = \mathcal{F}^t\left(\mathcal{E}\left(x_n^t; \theta_{\mathcal{E}}\right); \theta_{\mathcal{F}}^t\right)$.[24] Our MTL-DNN method is proposed based on hard parameter sharing.

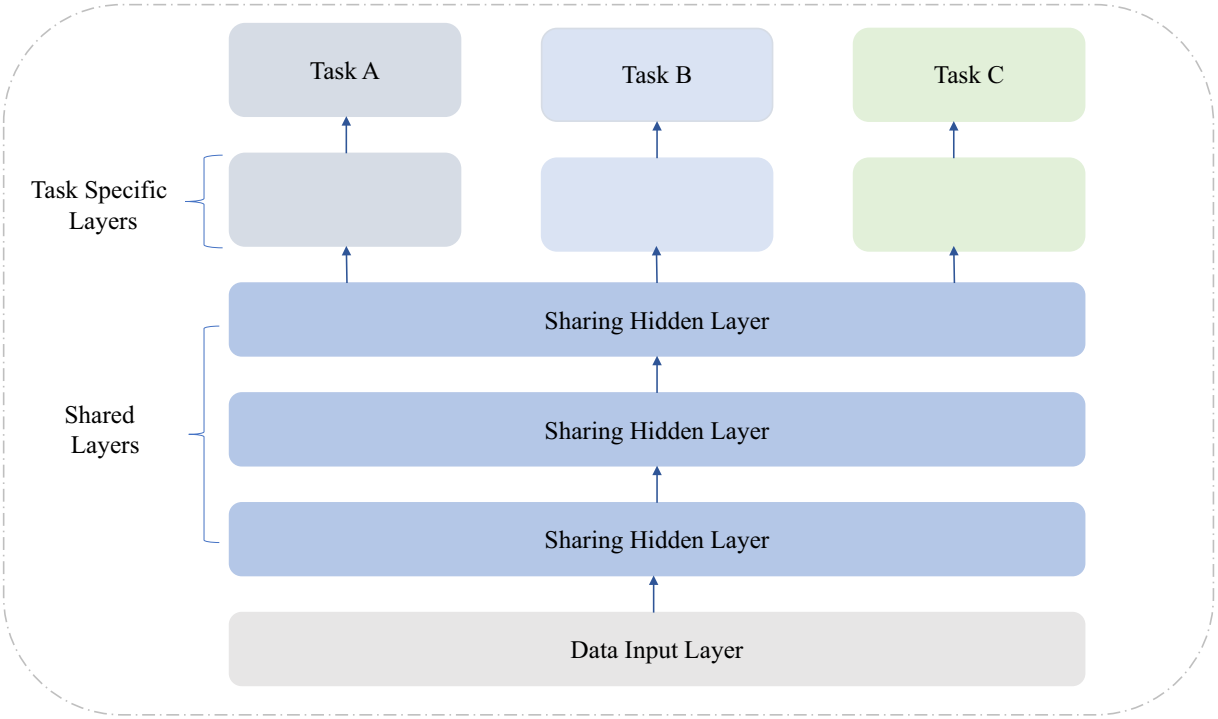

FIGURE 1 Multi-task deep learning framework based on hard parameter sharing.

3 | RESEARCH METHODOLOGY

In this section, we present the method of MTL-DNN proposed in this article. First, Section 3.1 introduces the framework for the implementation of the MTL-DNN model. And then, Sections 3.1 to 3.4 describe the settings of the activation function, loss function, and optimization algorithm in the MTL-DNN model, respectively. Finally, it introduces the implementation details of the MTL-DNN model in Section 3.5.

3.1 | Model framework

How to train both source and target projects with limited labeled data. To solve this question, Figure 2 shows a framework of the MTL-DNN model. In the MTL-DNN method, we treat each project as a task. So, in order to facilitate the representation of different related projects, we use different

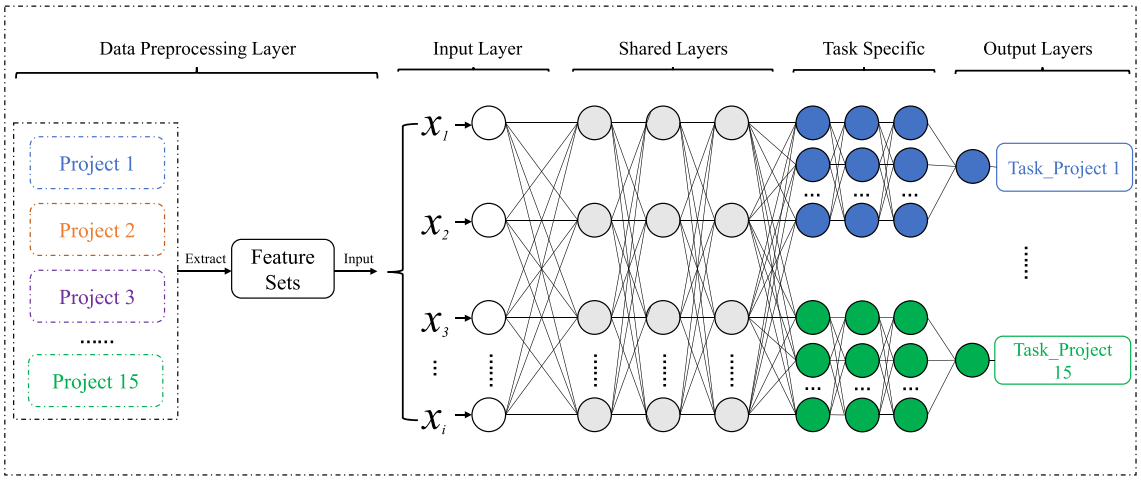

FIGURE 2 Framework of the MTL-DNN model.

colors in the figure. In the process of model construction: first, the commit-based features and labels of each project are collected, where the feature sets are

used as independent variables and label sets are used as dependent variables, a total of 15 projects are collected, see Section 4.1 for the description of the related data sets. Then, different from single-task learning, we use the features of all related projects as a whole input to the MTL-DNN model. Next, the common low-level features of all tasks are learned through three hidden layers (the hidden layers are implemented by multi-layer perceptron[26]). At the same time, to ensure the uniqueness of the task, each task has its own three task-specific layers to learn high-level features (the task-specific layers are implemented by fully-connected network). Finally, the results of each task are output (different colors represent the output of different projects). The settings for activation function, loss function, and optimization algorithm are used in the process of model training, see Sections 3.2 to 3.4.

3.2 | Activation function

Activation functions are composed of linear and nonlinear activation functions.[27] The output of a linear activation function generally does not have range limitations, which is not suitable for our study problem because our task is the classification task. The commonly used nonlinear activation functions in neural networks contain Logistic Sigmoid, Tanh, Rectified Linear Unit (ReLU), and softmax.[28] Logistic Sigmoid and Tanh are similar activation functions, where the Tanh is zero-centered but Logistic Sigmoid does not have to be. The disadvantages for them are computational expense and relatively inaccurate predictions near the edges. ReLU is a piecewise nonlinear function which is the identity function for positive input and zero for negative input. Hence, the output range of ReLU is $[0, \infty)$. ReLU, as shown in Equation (1). Softmax maps the output of neurons to the value in the interval (0,1), and the sum of all elements is equal to 1. It can be treated as probability, and the classification with the greatest probability is selected as the prediction target. Softmax, as shown in Equation (2), where z_j represents the output node number. We choose ReLU as the activation function of the shared hidden layer and specific task layer because of its fast-training speed, vanishing gradient problem, and overfitting reduction. The activation function of the classification task (activation function of output layer) uses the Softmax function.

$$\mathrm{ReLU}(x) = \max(0, x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{otherwise,} \end{cases} \tag{1}$$

$$\mathrm{Softmax}\,(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}. \tag{2}$$

3.3 | Loss function

Obviously, each task in our network is a binary classification task. We first construct the loss function of each task and then combine them as one loss function. For a binary classification problem, the widely used loss functions are Hinge, Square Hinge, and Binary Cross-Entropy. Hinge and Squared Hinge loss functions are mainly used in support vector machine models, and the target value is set at {-1,1}.[29] Binary Cross-Entropy is the default loss function for the binary classification task, and the target variable is set at {0,1}.[29] Binary Cross-Entropy, as shown in Equation (3). We choose Binary Cross-Entropy as the loss function because our binary classification tasks are encoded as 0 and 1. In the multi-task study in this article, we treat each task equally. Hence, the loss of the MTL-DNN multi-task method proposed in this article is the sum of each task loss. The combined loss function of multi-task is shown in Equation (4).

$$\begin{aligned} \text{Classification loss: } C^c = -\frac{1}{N}\sum_{i=1}^{N} & y_i \log(p\,(y_i)) \\ & + (1 - y_i)\log(1 - p\,(y_i))\,, \end{aligned} \tag{3}$$

$$\text{MTL-DNN loss:} C = \sum_{i=1}^{15} C_i^c. \tag{4}$$

3.4 | Gradient descent

Gradient descent is the main optimization algorithm for training neural networks. According to the number of samples used in each update of model parameters, there are three types of gradient descent algorithms:[30,31] (1) batch gradient descent (BGD)—uses all the data for training and then makes adjustments to the parameters; (2) stochastic gradient descent (SGD)—randomly selects an instance in the training set to compute the gradient at each iteration; and (3) mini-batch gradient descent (MBGD)—which is neither based on the full training sets nor based on a single instance, it is

a hybrid of BGD and SGD and calculates the gradient based on a small number of random instances for each iteration. BGD takes a long time to calculate gradient but its convergence is stable, SGD training speed is fast but it performs frequent updates with a high variance that cause the objective function to fluctuate heavily. MBGD is chosen as the optimizer for the developed MTL-DNN. The reason for choosing the MBGD algorithm: MBGD integrates the advantages of SGD and BGD. MBGD update parameters are defined as follows:[30]

$$\theta = \theta - \eta \cdot \nabla_\theta J\left(\theta; x^{(i:i+n)}; y^{(i:i+n)}\right), \tag{5}$$

where $x^{(i:i+n)}$ indicates that n training samples are randomly selected from the ith sample at each iteration, $y^{(i:i+n)}$ means the true value of random training samples, θ denotes w(weights) or b(biases) parameters, $\nabla_\theta J\left(\theta; x^{(i:i+n)}; y^{(i:i+n)}\right)$ indicates parameters gradient and η means learning rate.

3.5 | Implementation details

Algorithm 1 shows the detailed steps of the MTL-DNN method. First, we input the training data of multi-tasks, where the input layer performs a fully connected (FC) operation on the data of each task, and uses the BatchNorm function to normalize the results (line 1 to line 5). Then, this article uses each layer of the shared layers to perform FC operations on the output of the previous layer and uses dropout to prevent overfitting (line 6 to line 11). Next, this article uses each task layer of the multi-task layers to perform FC operation on the output of the shared layers to obtain the prediction results of each task (line 12 to line 14), and calculate the loss function according to the prediction results and data labels, obtain the derivative of the loss function with respect to each model parameter, and update the model parameters using gradient descent (line 15 to line 20). Finally, after several training iterations, the model parameter w_b is output (line 21). Table 1 shows the notations in Algorithm 1. In the process of model training, We trained our model using mini-batch gradient descent with a batch size of 128 samples, configured initial learning rate values as 0.0001, and also

Algorithm 1. MTL-DNN

Input: Training set x
Output: model parameter W_b

▷ **Input layer:**
1: **for** j=1,2,...,n **do**
2: $\gamma_j^i = \emptyset\left(w_j^i, x_j\right)$
3: **end for**
4: $\gamma^i = \left[\gamma_1^i, \gamma_2^i, \ldots, \gamma_n^i\right]$
5: $L^i = BN\left(\gamma^i\right)$
▷ **Shared layer:**
6: $\gamma_1^s = \left(w_j^S, L^i\right)$
7: $L_1^S = Dropout(\gamma_1^s, \mathrm{p}_1^s)$
8: **for** k=1,2,...,m **do**
9: $\gamma_k^s = \emptyset\left(w_k^S, L_{k-1}^S\right)$
10: $L_k^s = Dropout\left(\gamma_k^s, \mathrm{p}_1^s\right)$
11: **end for**
▷ **Multitask layer:**
12: **for** l=1,2,...,n **do**
13: $\gamma_l^t = \emptyset\left(w_l^t, L_m^S\right)$
14: **end for**
▷ **Model optimizer:**
15: **Initialize** b=0,$\alpha = 0.0001$
16: **while** stopping criteria not met **do**
17: $\boldsymbol{g}_b = \nabla \boldsymbol{C}$
18: $\boldsymbol{w}_b = \mathbf{w}_{b-1} - \alpha \cdot g_b$
19: $b = b + 1$
20: **end while**
21: **return** w_b

TABLE 1 Symbolic description of the MTL-DNN algorithm.

Symbol	Definition
x	Training set
n	Number of tasks
m	Number of shared layers
$\phi(w, x)$	The w parameters perform a fully connected operation on x
Li	Output of the *i*th layer
$Dropout(\gamma_k, p_k)$	Outputs in γ_k are discarded with probability p
w_k^s	Indicates the parameters of the *k*th layer in the s type layers
b	Iterative training times
α	Learning rate
C	Loss function
g_b	Gradient of the weight parameter w_b at the *b*th iteration
w_b	Parameters at the *b*th iteration

adopted the learning rate scheduler function (e.g., StepLR strategy) to dynamically adjust the learning rate during the training process. By using the StepLR strategy, the learning rate every 20 epochs became 0.9 of the original (the number of epochs = 600). For the initial settings of the weights and the Neuron biases, we used the default settings provided by PyThorch (PyThorch 1.10).

In addition, this article compares the training and inference time of the MTL-DNN and the STL-DNN (baseline method), respectively. Specifically, only one sample is used for each training or inference, that is, the batchsize is set to 1, and the unit is milliseconds. The average time is obtained through 1000 training or inference: (1) the run times for MTL-DNN and STL-DNN model training are 2.58 and 2.16 ms, respectively; (2) the run times for MTL-DNN and STL-DNN model inferencing are 1.95 and 1.57 ms, respectively. The experimental results show that the time-cost of MTL-DNN proposed in this article is acceptable.

4 | EXPERIMENTAL SETUP

In this section, we first introduce the datasets used in our study. Then, we describe performance indicators. Finally, we give the statistic test.

4.1 | Datasets

To evaluate the performance of our proposed MTL-DNN method, in our work, we employ 15 Android mobile apps[19,22] as data sets. These Android mobile apps are briefly described as follows: *Android Firewall* (AFall) is an application for the powerful iptables Linux firewall. It allows users to control which applications are permitted to access networks. *Alfresco* is an enterprise content management and business process management tool. *Android Sync* (Sync) is an Android synchronization manager, which syncs tasks securely and directly through the USB cable with Android devices. *Android Wallpaper* (wallpaper) provides a lot of beautiful wallpapers for Android phones. *AnySoft Keyboard* (keyboard) is an open-source, on-screen keyboard with multiple languages support with an emphasis on privacy, which is one of the most customizable keyboards available. *Apg* is an open source and free app for the Android system, which provides to encrypt, decrypt, digitally sign, and verify signatures for files. *Kiwix* is an offline Web reader, which is designed for Wikipedia offline. *Own Cloud Android* (Cloud) is a synchronization software, it allows users to better manage and synchronize data. *Page Turner* (Turner) is an eBook reader, which allows users to keep reading progress synchronized across multiple devices. *Notify Reddit* (Reddit) allows users to get notifications from their Android wearable devices. *Android Universal Image Loader* (Image) provides image loading, caching, and displaying. *Observable Scroll View* (Scroll) provides to listen to the scrolling status of the scrolling view and simply interact with the Toolbar. *Applozic Android SDK* (Applozic) makes real-time engagement with chat, video, and voice. *Delta Chat* (Delta) is an email-based instant messaging Chat tool, which takes advantage of the existing E-mail infrastructure and transforms the traditional style of sending and receiving E-mail into instant messaging (IM) mode. *Lottie* is an open-source tool, which adds animation effects to native applications. We choose these Android mobile apps as subjects of our study for the following reasons: (1) they belong to different application domains; (2) they are widely used in defect prediction of mobile application software;[19,22] and (3) they have a certain percentage of the defects which is suitable for our study.

TABLE 2 The basic information of the 15 apps.

Project	#LOC	#Commits	#Defective	#Clean	%DR
AFall	77,234	1025	414	611	40.4%
Alfresco	152,047	1004	214	790	21.3%
Sync	275,637	209	62	147	29.7%
Wallpaper	35,917	588	94	494	16.0%
Keyboard	114.784	2971	819	2152	27.6%
Apg	151,204	3780	1304	2476	34.5%
Kiwix	32,598	1373	350	1023	25.5%
Cloud	115,169	3700	830	2870	22.4%
Turner	30,943	164	23	141	14.0%
Reddit	9506	222	60	162	27.0%
Image	16,530	875	141	734	16.1%
Scroll	27,836	250	54	196	21.6%
Applozic	87,662	946	143	803	15.1%
Delta	96,971	2465	185	2280	7.5%
Lottie	57,291	235	36	199	15.3%

TABLE 3 The brief description of features.

Scope	Name	Definition
Size	LA	Lines of code added by the current change
	LD	Lines of code deleted by the current change
	LT	Lines of code in a file before the current change
Diffusion	NS	Number of modified subsystems involved in the current change
	ND	Number of modified directories involved in the current change
	NF	Number of modified files involved in the current change
	Entropy	Distribution of modified code across files involved in the current change
Purpose	FIX	Whether or not the current change is a bug fix
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem
History	NDEV	Number of developers changing the files
	AGE	Average time interval since the last change
	NUC	Number of unique changes to modified files

Table 2 summarizes basic information on Android mobile apps in this study. The second to the sixth columns in Table 2 represents the lines of code, the total number of commits, the number of defective commits, the number of clean commits, and the last column is the ratio of defective commits, respectively. From Table 2, we can find that the total number of commits for these apps is between 164 and 3780, which shows that these apps have committed different scales. Additionally, the ratio of defective commits for these apps is between 7.5% and 40.4%. Table 3 shows the change-level features of 5 different dimensions: size, code diffusion, purpose, developer experience, and file history. More specifically, the size represents the scale of code changes, including LA, LD, and LT features. Moser et al. found that the larger the scale of code changes, the more likely it is that defects are introduced.[32] Code diffusion means the distribution of code changes in related files. Purpose represents the goals of submitting code changes. The goals of developers submitting changes include bug fixes, implementation of new functions, code refactoring,

and documentations of addition[33] and so forth. Developer experience is used to quantify the developer experience of code changes. File history is used to quantify the modification history of change-related files. Prior studies showed that the more complex the file modification history is, the more likely it is to have defects.[32] The specific features of each dimension are shown in Table 3, these features are all used as the input of the MTL-DNN model. The reason for choosing these features is that they are widely used in JIT defect prediction.[20,34,35] In addition, the MTL-DNN framework proposed in this article is not only applicable to the above data sets but also applicable to JIT defect prediction of other mobile app data sets.

4.2 | Performance indicators

At present, most of the existing JIT defect prediction tasks on mobile apps are a binary classification problem, which is to confirm whether a code commit is defective or not. In our study, in order to evaluate the effectiveness of the proposed MTL-DNN method, we use two types of performance indicators, including F-measure and Matthews correlation coefficient (MCC). F-measure[36-41] and MCC[42,43] are widely used in previous defect prediction studies. The details of these indicators are defined as follows. The first performance indicator is F-measure, this indicator is the harmonic mean of the precision and recall. It is calculated as:

$$\text{F-measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}, \tag{6}$$

where Precision = TP/(TP + FP) and Recall = TP/(TP + FN). We set defective code commits as positive and non-defective code commits as negative. We combine the real results of a code commit with the predicted results of the model and divide them into true positive (TP), false positive (FP), true negative (TN), and false negative (FN). True positive (TP) denotes the number of a commits labeled as defective that are predicted as defective. False positive (FP) denotes the number of a commit labeled as non-defective that is predicted as defective. False negative (FN) denotes the number of a commit labeled as defective that is predicted as non-defective.

The second performance indicator is MCC, MCC describes the correlation coefficient between predicted results and real results, its definition is as follows:

$$MCC = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}, \tag{7}$$

where the definitions of TP, TN, FP, and FN are the same as those of TP, TN, FP, and FN in the F-measure. MCC is a comprehensive indicator that considers TP, TN, FP, and FN, simultaneously. The indicator value of the F-measure ranges from 0 to 1. The larger F-measure value denotes the better performance of the model. The value of MCC is in the range of [−1, 1], where 1 indicates that the predicted values are completely consistent with the actual results; 0 represents that the predicted results are worse than the random predicted results; −1 means that the predicted results are completely inconsistent with the actual results.

4.3 | Statistical tests

Statistical tests can help understand if there is a statistically significant difference in performance between the two methods. In our study, we first apply the Wilcoxon signed-rank test to inspect whether the difference in prediction performance between the MTL-DNN method and baseline method is significant. Wilcoxon signed-rank test is a nonparametric check method, which does not consider the underlying data to follow normal distribution.[44-46] If the *p*-value of the Wilcoxon signed-rank test is lower than 0.05, it indicates that the difference in performance between the two methods is significant, otherwise, not significant.

In addition, in order to further quantify the amount of difference between the MTL-DNN method and baseline method, we use Cliff's delta (δ) statistic[47-49] in our study. Cliff's delta is a non-parametric effect size measure that quantifies the amount of difference between two approaches. The value of delta ranges from −1 to 1, where $|\delta|=1$ means that there is no overlap between the performance of the two methods (i.e., all F-measure values from one method are higher than the F-measure values of the other method), $|\delta|=0$ indicates that there is a complete overlap in performance between the two methods. Table 4 shows the meanings of the different δ values.[47]

5 | RESULTS

In this section, we report the experimental results in detail for our three research questions.

TABLE 4 Cliff's delta and the effectiveness level.

Cliff's delta (δ)	Effectiveness level
$\lvert\delta\rvert < 0.147$	Negligible
$0.147 \leq \lvert\delta\rvert < 0.33$	Small
$0.33 \leq \lvert\delta\rvert < 0.474$	Medium
$\lvert\delta\rvert \geq 0.474$	Large

5.1 | RQ1: Whether our proposed MTL-DNN method can effectively improve JIT defect prediction performance of mobile apps?

5.1.1 | Motivation

In practice, JIT is used to predict which changes may have defects when they are initially committed. Using the prediction results, software developers can fix their changes before committing codes, which will save the effort of fixing bugs in the future. In this research question, we want to investigate whether our proposed method can improve the performance of JIT defect prediction compared to single-task learning based on deep neural networks (STL-DNN).

5.1.2 | Approach

To answer this RQ1, this article sets the single-task learning based on the deep neural networks (STL-DNN) as a baseline to explore whether our proposed MTL-DNN method can achieve better defect prediction performance compared to the baseline method. In this experiment, we used both F-measure and MCC performance indicators to evaluate MTL-DNN and STL-DNN approaches. To obtain a realistic comparison, we adopt 30 times 3-fold cross-validation. Specifically, at each 3-fold cross-validation, we randomize and then divide the data set into 3 parts of approximately equal size. Then, we test each part by the prediction model built with the remainder of the data set. This process is repeated 30 times to alleviate potential sampling bias. Based on these predictive values, this article uses Wilcoxon's signed-rank test[44] with a Bonferroni correction[50,51] to check whether the prediction performance of two methods has a significant difference. Wilcoxon's signed-rank test is a nonparametric statistical test, and the Bonferroni method is used to correct the p-value. The significance level α is set as 0.05,[52] if the p-value is less than 0.05, it indicates that there is a significant difference in performance between the two methods, otherwise, no significant difference.

Furthermore, we also use cliff's δ in Table 4, which is a nonparametric effect size measure. It is used to quantify the amount of difference between MTL-DNN and STL-DNN methods. The value of delta ranges from −1 to 1, where $|\delta|=1$ means that there is no overlap between the performance of the two methods, where $|\delta|=0$ indicates that there is a complete overlap. Table 4 shows the meanings of the different δ values.

5.1.3 | Results

Figures 3 and 4 respectively employ the boxplots to describe the distributions of F-measure and MCC performance indicators obtained from 30 times 3-fold cross-validation for MTL-DNN and STL-DNN methods with respect to each of the projects. For each method, the boxplots show the median, the 25th percentile, and the 75th percentile. From Figures 3 to 4, it is obvious that our proposed MTL-DNN performs substantially better than STL-DNN on almost all projects.

Tables 5 and 6 respectively present the mean values of F-measure and MCC performance indicators obtained by MTL-DNN and STL-DNN methods under 3×30 cross-validation. In both tables, the first column shows experimental projects. The second column and third column indicate performance values by the STL-DNN (baseline) method and the MTL-DNN method. The fourth column is the percentage of improvement for the MTL-DNN method over the STL-DNN method. The last two columns respectively indicate whether the performance of the MTL-DNN method is statistically significantly better than the STL-DNN method, and the effect size in terms of the cliff's δ. The last row in Tables 5 and 6 shows the average values for the 15 projects.

From Table 5, we have the following observations. For all the experimental projects, the MTL-DNN method has a larger F-measure than the STL-DNN in terms of F-measure. Among all the projects, the maximum performance is improved by 130.464% (AFall project), and the minimum is improved by 13.356% (Alfresco). On average, the MTL-DNN method leads to about 57.0% improvement over the STL-DNN in terms of the F-measure. This indicates that our proposed MTL-DNN method performs substantially better than the baseline method. From Table 5, we can see that The Wilcoxon signed-rank test p-values are very significant (<0.05),[44] Furthermore, the effect sizes of all projects are moderate to large (i.e. $|\delta| \geq 0.147$).[47] This shows that the performance of our proposed method is significantly better than the baseline method, and the performance difference between the two methods is large.[47]

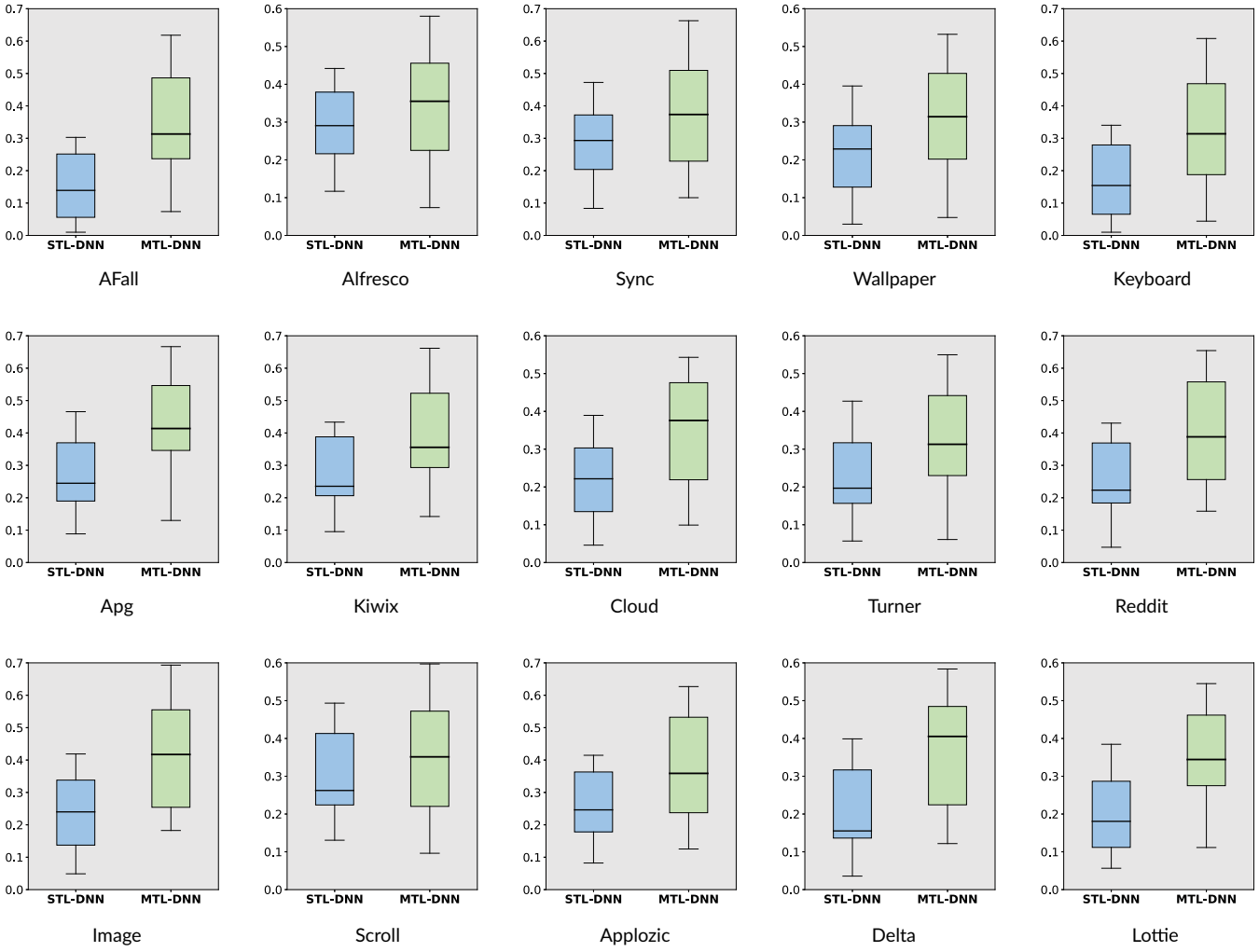

FIGURE 3 Performance comparison of F-measure under 30 times 3-fold cross-validation: MTL-DNN versus STL-DNN.

From Table 6, we can see that our proposed method has a larger MCC than the baseline method in terms of the MCC in all projects. The wallpaper project has the most improvement in MCC performance among all projects, and it improves by 164.407%. The smallest performance improvement is the delta project, which improved by 26.400%. On average, our proposed method leads to about 85.0% improvement over the STL-DNN in terms of the MCC. This shows that the MTL-DNN method performs better than the STL-DNN method. From Table 6, we find that The Wilcoxon signed-rank test *p*-values of all the projects are very significant (<0.001),[44] and the effect sizes are moderate to large (i. e. $|\delta| \geq 0.147$).[47] This indicates that the performance of our proposed method is significantly better than the baseline method, and the performance difference between the two methods is large.[47]

RQ1 Summary: *For the new research scenario that both the source and the target projects only contain limited labeled training data sets, the MTL-DNN method proposed in this article can achieve better software defect prediction performance than the STL-DNN method. The experimental results show that the MTL-DNN method can effectively utilize the shared information of related projects when building a JIT software defect prediction model.*

5.2 | RQ2: Whether MTL-DNN can improve JIT defect prediction performance compared with the classical single-task machine learning methods?

5.2.1 | Motivation

Support vector machine (SVM)[53,54] is a supervised learning model with relevant learning algorithms which is used to analyze data for the classification task. Random forest is an ensemble of decision trees, with each decision tree trained on a subset of the training data.[55] In this research

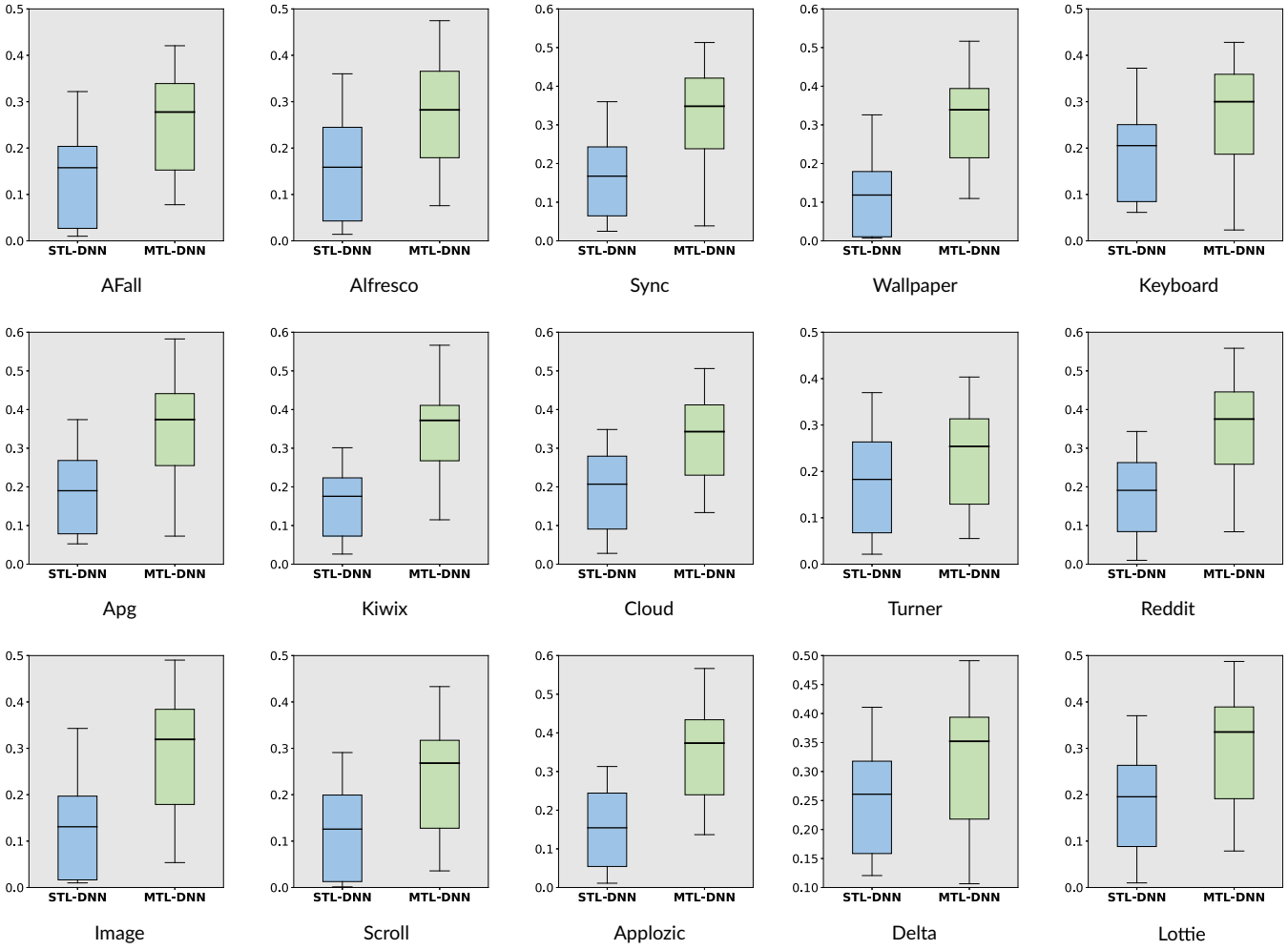

FIGURE 4 Performance comparison of MCC under 30 times 3-fold cross-validation: MTL-DNN versus STL-DNN.

question, we further investigate the MTL-DNN performance with two classical single-task machine learning models which are support vector machine (STL-SVM) and random forest (STL-RF). These two methods are selected because of their good performance in classification.

5.2.2 | Approach

To answer this RQ2, we first respectively use 30 times 3-fold cross-validation for the MTL-DNN, STL-SVM, and STL-RF methods. Then, based on these predictive performance values, we used Wilcoxon's signed-rank test[44] with a Bonferroni adjustment[50] to examine whether the prediction performance of the MTL-DNN method is significantly better than the STL-SVM and STL-RF methods. If the p-value from Wilcoxon's signed-rank test is less than 0.05,[52] it represents that performance of the MTL-DNN method is significantly better than the STL-SVM and STL-RF methods, otherwise, not significantly better. Finally, we further use cliff's δ method to quantify the amount of difference between the MTL-DNN, STL-SVM, and STL-FR methods. when the value of cliff's δ in Table 4 is greater than or equal to 0.147,[47] it respectively indicates that the performance difference of MTL-DDN versus STL-SVM and MTL-DDN versus STL-RF are not ignored.

5.2.3 | Results

Figures 5 and 6 respectively adopt the boxplots to describe the distributions of F-measure and MCC performance indicators obtained from 30 times 3-fold cross-validation for MTL-DNN, STL-SVM, and STL-RF methods for each of the system. From Figures 5 and 6, we can find that the performance of the MTL-DDN method is also better than STL-SVM and STL-RF methods for all the systems.

TABLE 5 Comparison result for MTL-DNN and STL-DNN in terms of mean F-measure.

Project	STL-DNN	MTL-DNN	%↑	*p*-value	$\lvert\delta\rvert$
AFall	0.151	**0.348**	**130.464%**	**<0.001**	**0.692**
Alfresco	0.292	**0.331**	**13.356%**	**0.039**	**0.177**
Sync	0.287	**0.373**	**29.865%**	**<0.001**	**0.318**
Wallpaper	0.213	**0.311**	**46.009%**	**<0.001**	**0.398**
Keyboard	0.167	**0.322**	**92.418%**	**<0.001**	**0.534**
Apg	0.270	**0.431**	**59.630%**	**<0.001**	**0.620**
Kiwix	0.274	**0.395**	**44.161%**	**<0.001**	**0.437**
Cloud	0.218	**0.345**	**58.257%**	**<0.001**	**0.415**
Turner	0.226	**0.326**	**44.248%**	**<0.001**	**0.512**
Reddit	0.253	**0.400**	**58.103%**	**<0.001**	**0.498**
Image	0.240	**0.423**	**76.250%**	**<0.001**	**0.611**
Scroll	0.299	**0.343**	**14.716%**	**0.047**	**0.170**
Applozic	0.259	**0.365**	**40.927%**	**<0.001**	**0.386**
Delta	0.210	**0.365**	**73.810%**	**<0.001**	**0.624**
Lottie	0.198	**0.357**	**80.303%**	**<0.001**	**0.666**
Avg	0.237	**0.362**	**57.501%**	**<0.001**	**0.471**

Note: The bold values indicate that our proposed method MTL-DNN outperforms other methods.

TABLE 6 Comparison result for MTL-DNN and STL-DNN in terms of mean MCC.

Project	STL-DNN	MTL-DNN	%↑	*p*-value	$\lvert\delta\rvert$
AFall	0.139	**0.255**	**83.453%**	**<0.001**	**0.557**
Alfresco	0.157	**0.273**	**73.885%**	**<0.001**	**0.536**
Sync	0.165	**0.324**	**96.363%**	**<0.001**	**0.667**
Wallpaper	0.118	**0.312**	**164.407%**	**<0.001**	**0.785**
Keyboard	0.187	**0.266**	**42.246%**	**<0.001**	**0.417**
Apg	0.187	**0.346**	**85.027%**	**<0.001**	**0.628**
Kiwix	0.159	**0.348**	**118.868%**	**<0.001**	**0.788**
Cloud	0.195	**0.327**	**67.692%**	**<0.001**	**0.599**
Turner	0.178	**0.230**	**29.213%**	**<0.001**	**0.283**
Reddit	0.180	**0.352**	**95.556%**	**<0.001**	**0.673**
Image	0.128	**0.287**	**124.219%**	**<0.001**	**0.643**
Scroll	0.122	**0.237**	**94.262%**	**<0.001**	**0.553**
Applozic	0.154	**0.350**	**127.273%**	**<0.001**	**0.763**
Delta	0.250	**0.316**	**26.400%**	**<0.001**	**0.376**
Lottie	0.185	**0.297**	**60.541%**	**<0.001**	**0.513**
Avg	0.167	**0.301**	**85.960%**	**<0.001**	**0.585**

Note: The bold values indicate that our proposed method MTL-DNN outperforms other methods.

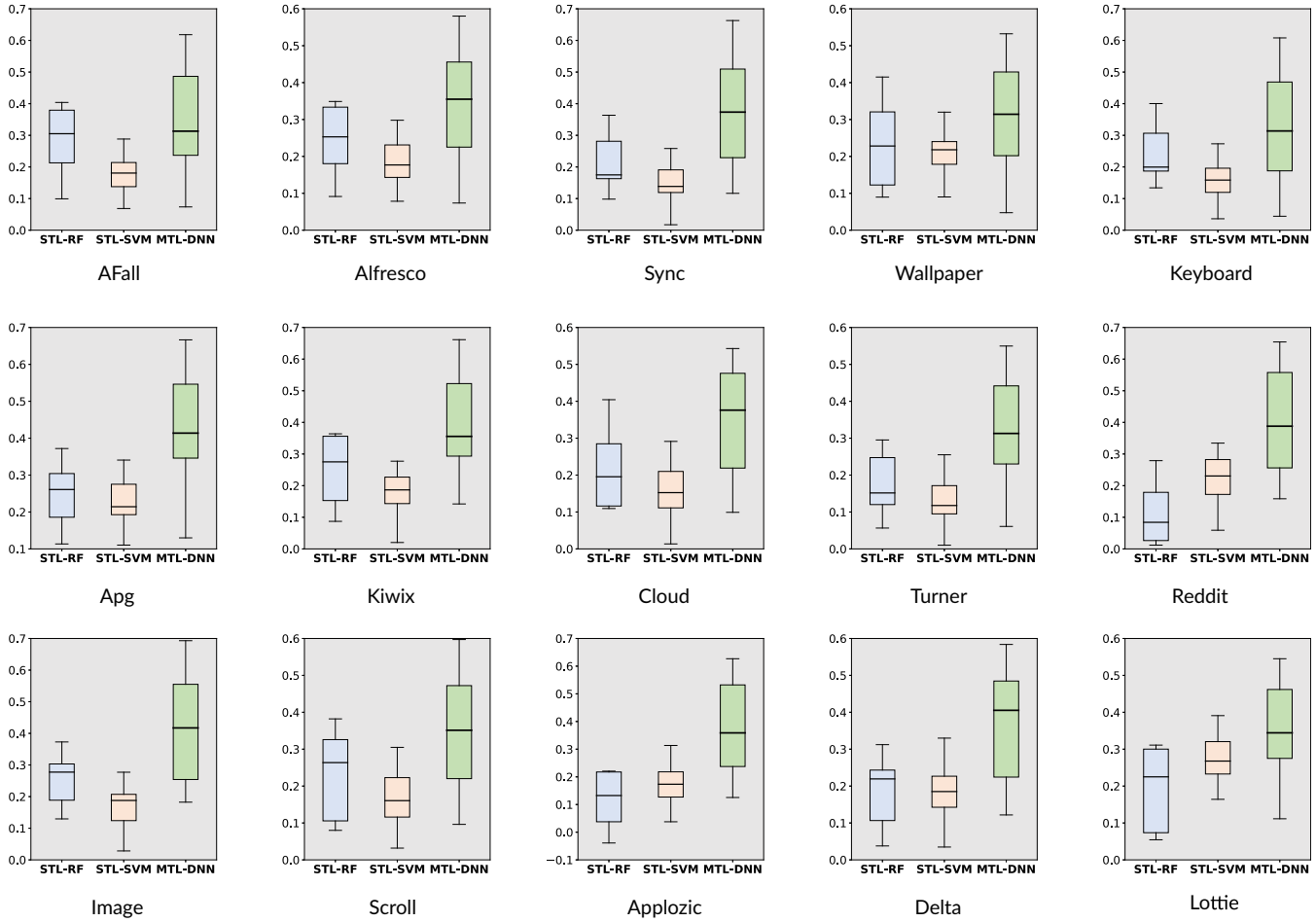

FIGURE 5 Performance comparison of F-measure under 30 times 3-fold cross-validation: MTL-DNN versus STL-SVM versus STL-RF.

Table 7 presents the F-measure for MTL-DNN versus STL-SVM and MTL-DNN versus STL-RF. For MTL-DNN versus STL-SVM, the MTL-DNN method has a larger value than STL-SVM in terms of F-measure. For all the systems, the minimum performance value of the F-measure is improved by 27.957 percent (Lottie), and the maximum performance value of the F-measure is improved by 154.687 percent (Turner). On average, the MTL-DNN method leads to about 96.7 percent improvement over the STL-SVM method. The *p*-values are very significant (<0.001).[44] Besides, the effect sizes are moderate to large.[47] For MTL-DNN versus STL-RF, the predictive performance of the MTL-DNN method is also better than the STL-RF method, we can improve the F-measure between 19.588% and 270.370%, and the average improvement is about 69.2 percentage points. The Wilcoxon signed-rank test *p*-values are also very significant (<0.05).[44] In addition, The effect sizes are moderate or large in terms of the Cliff's δ.[47]

Table 8 shows MCC performance indicator values for MTL-DNN versus STL-SVM and MTL-DNN versus STL-RF. From Table 8, we can find that the performance values of the MTL-DNN method are significantly better than the STL-SVM method and MTL-DNN method. For MTL-DNN versus STL-SVM, the highest improvement can be up to 154.687%, the *p*-values are very significant (<0.05).[44] Furthermore, the effect sizes are moderate or large in terms of the Cliff's δ.[47] For MTL-DNN versus STL-RF, from Table 8, we can observe that the highest improvement can be up to about 12 times, and the average improvement is about 2 times. The Wilcoxon signed-rank test *p*-values are also very significant (<0.001),[44] Besides, the effect sizes are moderate to large, this indicates that the performance difference between MTL-DNN versus STL-SVM and MTL-DNN versus STL-RF is large.[47]

RQ2 Summary: *For the support vector machine (STL-SVM) and the random forest (STL-RF), these two classical methods can achieve good performance in defect prediction. Compared with the two methods, we propose that the prediction performance of the MTL-DNN method is also significantly better than the STL-SVM method and the STL-FR method.*

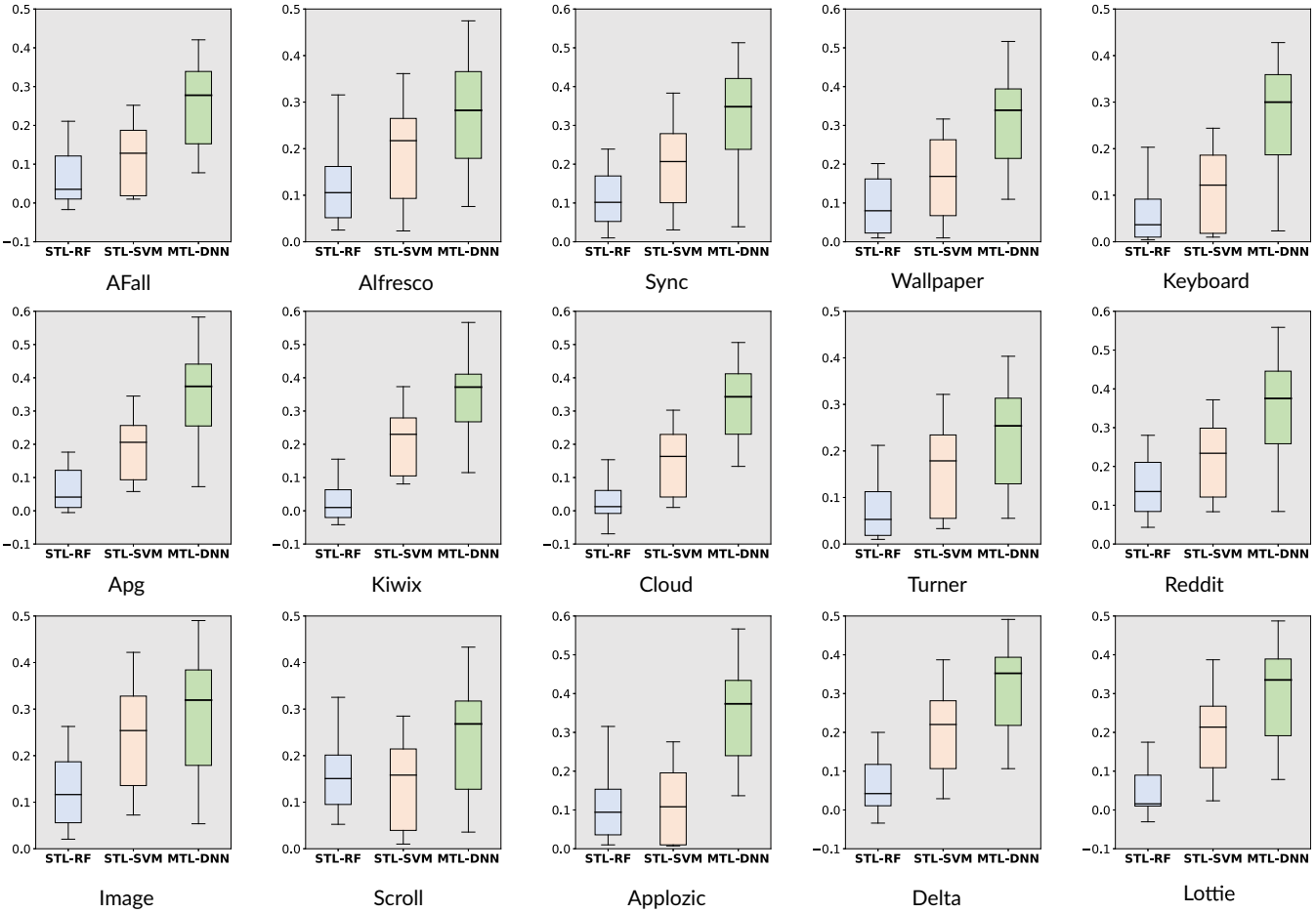

FIGURE 6 Performance comparison of MCC under 30 times 3-fold cross-validation: MTL-DNN versus STL-SVM versus STL-RF.

5.3 | RQ3: How does the number of labeled instances affect the performance of MTL-DNN?

5.3.1 | Motivation

In RQ1, this article adopts the 30 times 3-fold cross-validation method to investigate the lack of labeled data in the new scenario. The experimental results show that the MTL-DNN method proposed by us is significantly better than the STL-DNN method. However, whether the performance of our method has changed when the percentage of the labeled data is different, in this research question, we further study the impact of the proportion of differently labeled data on the performance of the MTL-DNN method.

5.3.2 | Approach

To answer this RQ3, first, we allocate the labeled data to training in different proportions, 20%, 40%, and 60%, respectively. Then, for different proportions of labeled data, we all use the 30 times 3-fold cross-validation method. Finally, based on these performance values, we take their average values as the experimental results of different proportions.

5.3.3 | Results

Table 9 shows the average performance of F-measure and MCC from the MTL-DNN method using different data proportions as the training sets. From Table 9, the first column shows experimental projects. The second column and third column, respectively, indicate the performance of F-measure and MCC. The last row in Table 9 represents the average values for all projects.

TABLE 7 Comparison result for MTL-DNN versus STL-SVM and MTL-DNN versus STL-RF in terms of mean F-measure.

	MTL-DNN versus STL-SVM					MTL-DNN versus STL-RF				
Project	**STL-SVM**	**MTL-DNN**	**%↑**	***p*-value**	**$\|\delta\|$**	**STL-RF**	**MTL-DNN**	**%↑**	***p*-value**	**$\|\delta\|$**
AFall	0.177	**0.348**	**96.610%**	**<0.001**	**0.666**	0.291	**0.348**	**19.588%**	**0.009**	**0.202**
Alfresco	0.185	**0.331**	**78.919%**	**<0.001**	**0.575**	0.248	**0.331**	**33.468%**	**<0.001**	**0.376**
Sync	0.148	**0.373**	**152.027%**	**<0.001**	**0.781**	0.210	**0.373**	**77.619%**	**<0.001**	**0.590**
Wallpaper	0.209	**0.311**	**48.804%**	**<0.001**	**0.444**	0.234	**0.311**	**32.906%**	**<0.001**	**0.331**
Keyboard	0.157	**0.322**	**105.096%**	**<0.001**	**0.579**	0.245	**0.322**	**31.429%**	**<0.001**	**0.275**
Apg	0.228	**0.431**	**89.035%**	**<0.001**	**0.804**	0.248	**0.431**	**73.790%**	**<0.001**	**0.743**
Kiwix	0.177	**0.395**	**123.164%**	**<0.001**	**0.820**	0.256	**0.395**	**54.297%**	**<0.001**	**0.498**
Cloud	0.155	**0.347**	**123.871%**	**<0.001**	**0.696**	0.214	**0.345**	**61.215%**	**<0.001**	**0.530**
Turner	0.128	**0.326**	**154.687%**	**<0.001**	**0.784**	0.174	**0.326**	**87.356%**	**<0.001**	**0.637**
Reddit	0.220	**0.400**	**81.818%**	**<0.001**	**0.621**	0.108	**0.400**	**270.370%**	**<0.001**	**0.897**
Image	0.168	**0.423**	**151.786%**	**<0.001**	**0.879**	0.255	**0.423**	**65.882%**	**<0.001**	**0.585**
Scroll	0.164	**0.343**	**109.146%**	**<0.001**	**0.653**	0.231	**0.343**	**48.485%**	**<0.001**	**0.438**
Applozic	0.174	**0.365**	**109.770%**	**<0.001**	**0.710**	0.117	**0.365**	**211.966%**	**<0.001**	**0.820**
Delta	0.189	**0.365**	**93.122%**	**<0.001**	**0.642**	0.187	**0.365**	**95.187%**	**<0.001**	**0.629**
Lottie	0.279	**0.357**	**27.957%**	**<0.001**	**0.414**	0.196	**0.357**	**82.143%**	**<0.001**	**0.658**
Avg	0.184	**0.362**	**96.739%**	**<0.001**	**0.671**	0.214	**0.362**	**69.159%**	**<0.001**	**0.547**

Note: The bold values indicate that our proposed method MTL-DNN outperforms other methods.

TABLE 8 Comparison result for MTL-DNN versus STL-SVM and MTL-DNN versus STL-RF in terms of mean MCC.

	MTL-DNN versus STL-SVM					MTL-DNN versus STL-RF				
Project	**STL-SVM**	**MTL-DNN**	**%↑**	***p*-value**	**$\|\delta\|$**	**STL-RF**	**MTL-DNN**	**%↑**	***p*-value**	**$\|\delta\|$**
AFall	0.118	**0.255**	**116.102%**	**<0.001**	**0.656**	0.065	**0.255**	**292.308%**	**<0.001**	**0.849**
Alfresco	0.194	**0.273**	**40.7216%**	**<0.001**	**0.401**	0.120	**0.273**	**127.500%**	**<0.001**	**0.723**
Sync	0.201	**0.324**	**61.194%**	**<0.001**	**0.546**	0.111	**0.324**	**191.892%**	**<0.001**	**0.837**
Wallpaper	0.166	**0.312**	**87.952%**	**<0.001**	**0.623**	0.091	**0.312**	**242.857%**	**<0.001**	**0.898**
Keyboard	0.116	**0.266**	**129.310%**	**<0.001**	**0.680**	0.061	**0.266**	**336.066%**	**<0.001**	**0.859**
Apg	0.191	**0.346**	**81.152%**	**<0.001**	**0.631**	0.063	**0.346**	**449.206%**	**<0.001**	**0.916**
Kiwix	0.214	**0.348**	**62.617%**	**<0.001**	**0.638**	0.025	**0.348**	**1292.000%**	**<0.001**	**0.990**
Cloud	0.147	**0.327**	**122.449%**	**<0.001**	**0.757**	0.029	**0.327**	**1027.586%**	**<0.001**	**0.991**
Turner	0.161	**0.230**	**42.857%**	**<0.001**	**0.396**	0.074	**0.230**	**210.811%**	**<0.001**	**0.782**
Reddit	0.222	**0.352**	**58.559%**	**<0.001**	**0.568**	0.150	**0.352**	**134.667%**	**<0.001**	**0.788**
Image	0.243	**0.287**	**18.107%**	**0.016**	**0.208**	0.126	**0.287**	**127.778%**	**<0.001**	**0.672**
Scroll	0.144	**0.237**	**64.583%**	**<0.001**	**0.487**	0.160	**0.237**	**48.125%**	**<0.001**	**0.384**
Applozic	0.111	**0.350**	**215.315%**	**<0.001**	**0.868**	0.111	**0.350**	**215.315%**	**<0.001**	**0.898**
Delta	0.202	**0.316**	**56.436%**	**<0.001**	**0.540**	0.059	**0.316**	**435.593%**	**<0.001**	**0.954**
Lottie	0.197	**0.297**	**50.761%**	**<0.001**	**0.464**	0.046	**0.297**	**545.652%**	**<0.001**	**0.932**
Avg	0.175	**0.301**	**72.000%**	**<0.001**	**0.564**	0.086	**0.301**	**250.116%**	**<0.001**	**0.832**

Note: The bold values indicate that our proposed method MTL-DNN outperforms other methods.

TABLE 9 The performance of when considering different percentage of labeled modules in terms of F-measure and MCC.

	F-measure			MCC		
Project	**20%**	**40%**	**60%**	**20%**	**40%**	**60%**
AFall	0.091	0.102	**0.162**	−0.019	0.044	**0.073**
Alfresco	0.031	0.082	**0.148**	−0.001	0.047	**0.139**
Sync	0.031	0.082	**0.198**	−0.039	0.056	**0.075**
Wallpaper	0.052	0.099	**0.144**	−0.041	0.034	**0.103**
Keyboard	0.037	0.069	**0.214**	−0.027	0.037	**0.100**
Apg	0.034	0.112	**0.164**	0.048	0.064	**0.081**
Kiwix	0.056	0.074	**0.168**	0.001	0.038	**0.048**
Cloud	0.082	0.079	**0.165**	−0.039	0.039	**0.111**
Turner	0.055	0.102	**0.151**	−0.009	0.031	**0.109**
Reddit	0.016	0.086	**0.149**	−0.014	0.049	**0.078**
Image	0.015	0.092	**0.161**	−0.037	0.061	**0.089**
Scroll	0.022	0.096	**0.156**	0.019	0.032	**0.117**
Applozic	0.010	0.075	**0.176**	−0.073	0.033	**0.052**
Delta	0.071	0.115	**0.131**	−0.037	0.031	**0.097**
Lottie	0.067	0.078	**0.225**	0.004	0.028	**0.122**
AVG	0.045	0.090	**0.167**	−0.018	0.042	**0.093**

Note: The bold values indicate that the performance of our proposed method improves with the increase of training data.

In terms of the F-measure, when using 20%, 40%, and 60% labeled data as training data, the average performance of the MTL-DNN method is 0.045, 0.090, and 0.167, respectively. Regarding the MCC, the average performance of the MTL-DNN method is −0.018, 0.042, and 0.093, respectively.

RQ3 Summary: *The experimental results show that the performance of the MTL-DNN method is effectively improved with the increase of training data. At the same time, we also find that the improvement of prediction performance is small by using a small proportion of labeled data, however the prediction performance is effectively improved when the training data is increased from 20% to 60%. This shows that the better the prediction performance will be obtained by using the more training data.*

6 | RELATED WORK

Traditional defect prediction techniques focus on predicting software entities at a coarse-grained level, such as the package and file level. This granularity presents challenges for the practical usage of these techniques. For example, predictive models are likely to predict files containing thousands of lines as defective.[56] However, it's challenging to discover and fix potential bugs for the developers.

To address the challenges mentioned above, prior studies have proposed different defect prediction techniques.[57-60] However, JIT defect prediction has many advantages over other defect prediction techniques.[61] Such as, it allows developers to review finer-granularity codes as change-level. JIT defect prediction methods mainly include supervised learning and unsupervised learning.

For the supervised learning methods, JIT defect prediction can be traced back to the studies of Mockus et al.,[62] who used some change features to predict the possibility of a change introducing software defects. The experimental results indicate that these change features are predictions that can be used to introduce defects in the change. For developers, JIT defect prediction is more practical than traditional defect prediction. Recently, Kim et al. extracted features from multiple data sources, such as change logs, source codes, and change metadata to build a predictive model. Their experimental results show that the model can effectively improve defect prediction performance.[63]In recent works, Zhang et al.[64] proposed the FENSE approach, a feature-based ensemble modeling approach to cross-project JITDP. This method considers the differences in project features and integrates models with higher transferability. Experimental results show that it can improve performance by 10%.

For the unsupervised learning methods, Yang et al.[21] established an unsupervised model based on change-level features, and the experimental results of six industrial-level systems represented that the unsupervised model performed well in predicting performance. Specifically, compared with the supervised JIT defect prediction models, the unsupervised JIT defect prediction models enable developers to detect more defects, which also checks the code that accounts for 20% of the total modified lines of code. Subsequently, Fu et al.[65] proposed to use the labeled training data to select appropriate features in the work on the unsupervised JIT defect prediction modeling technology by Yang, and then implement the unsupervised method of Yang, which can achieve better performance. In the latest studies, Liu et al.[66] proposed CCUM unsupervised model which is based on code churn. Through experiments of six open-source software projects, the results show that CCUM performs better than all the state-of-the-art supervised and unsupervised models in effort-aware JIT defect prediction.

Our study is different from the above studies, first, we propose a new scenario, which both source and target projects have limited labeled data. Then we propose MTL-DNN approach, experimental results indicate the effectiveness of our proposed method.

7 | THREATS TO VALIDITY

This section mainly discusses the threats to the construct, internal, and external validity of our study.

7.1 | Threats to the construct validity

There are two potential threats to the construct validity in our experiments. The first threat is the dependent variable in the data sets, which is a binary variable and indicates whether a committing introduces a defect. The second threat is independent variables in the data sets. The independent variables in this study include features such as LA, LD, and LT and so forth. For the data sets composed of dependent and independent variables, this data sets are widely used in mobile software defect prediction.[19,22] Hence, the dependent variable and independent variables in this study are considered acceptable.

7.2 | Threats to the internal validity

The threat to the internal validity of this article comes from the evaluation Indicators. We adopt F-measure and MCC as predictive performance indicators to evaluate the JIT mobile software defect prediction model. These two indicators are widely used as software defect prediction evaluation indicators.[36,41] Therefore, the prediction results of the defect prediction model are credible.

7.3 | Threats to the external validity

The external threat in this article is whether the research results of this article can be extended with other types of software systems. The experimental results of this article are consistent to the above fifteen open-source software systems. At the same time, there are more data sets, so the results are statistically significant. Even so, there is still no guarantee that the results of this study can be extended to other types of software systems. To eliminate this threat, we hope that more researchers repeat this experiment on more types of software systems.

8 | CONCLUSION

In this article, we have proposed a new approach called MTL-DNN, which is based on multi-task learning. Our method learns the common low-level features of related tasks by sharing hidden layers, and in order to ensure the uniqueness of tasks, each task has its own unique layer to learn high-level features. We evaluate our method on 15 Android mobile apps, a total of 19,768 committed instances. The experimental results show that our method significantly outperforms state-of-the-art single-task deep learning and classical machine learning methods. Specifically, (1) compared with single-task deep learning, F-measure and MCC are improved by an average of 57.50% and 85.96%, respectively; (2) in contrast with SVM, F-measure and MCC are improved by an average of 96.74% and 72.00%, respectively; (3) F-measure and MCC increased by 69.159% and 2.5 times on average, respectively, when compared with random forest. The results demonstrate that our proposed MTL-DNN method can effectively solve the problem of insufficient labeled data of source and target projects and improve the performance of defect prediction by utilizing the correlation between projects. This article only studies software defect prediction based on commit-level. However, it is not clear whether software defect prediction based on lines of code has practical value. An interesting future study in this article will be to extend this research to lines of code.

ACKNOWLEDGMENTS

National Science Foundation of China under Grant/Award nos. 61906085, 61972192, 41972111; The Second Tibetan Plateau Scientific Expedition and Research Program under Grant/Award no. 2019QZKK0204. This work is partially supported by Collaborative Innovation Center of Novel Software Technology and Industrialization.

CONFLICT OF INTEREST STATEMENT

The authors declare no conflict of interest.

DATA AVAILABILITY STATEMENT

Data available on request due to restrictions, for example, privacy or ethical.

ORCID

Qiguo Huang https://orcid.org/0000-0001-7912-7175

REFERENCES

1. Lee S, Dolby J, Ryu S. HybriDroid: static analysis framework for android hybrid applications. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. Association for Computing Machinery; 2016:250-261.
2. Martin W, Sarro F, Jia Y, Zhang Y, Harman M. A survey of app store analysis for software engineering. *IEEE Trans Softw Eng*. 2017;43(9):817-847. doi:10.1109/TSE.2016.2630689
3. Ye J, Chen K, Xie X, et al. An empirical study of GUI widget detection for industrial mobile games. Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery; 2021:1427-1437.
4. Li Y, Liu J, Cao B, Wang C. Joint optimization of radio and virtual machine resources with uncertain user demands in mobile cloud computing. *IEEE Trans Multimed*. 2018;20(9):2427-2438. doi:10.1109/TMM.2018.2796246
5. Li Y, Xia S, Zheng M, Cao B, Liu Q. Lyapunov optimization-based trade-off policy for mobile cloud offloading in heterogeneous wireless networks. *IEEE Trans Cloud Comput*. 2022;10(1):491-505. doi:10.1109/TCC.2019.2938504
6. Wang W, Wang Y, Duan P, Liu T, Tong X, Cai Z. A triple real-time trajectory privacy protection mechanism based on edge computing and blockchain in mobile crowdsourcing. *IEEE Trans Mob Comput*. 2022;1-18. doi:10.1109/TMC.2022.3187047
7. Wu Y, Guo H, Chakraborty C, Khosravi M, Berretti S, Wan S. Edge computing driven low-light image dynamic enhancement for object detection. *IEEE Trans Netw Sci Eng*. 2022.
8. Dalla Palma S, Di Nucci D, Palomba F, Tamburri DA. Within-project defect prediction of infrastructure-as-code using product and process metrics. *IEEE Trans Softw Eng*. 2022;48(6):2086-2104. doi:10.1109/TSE.2021.3051492
9. Yedida R, Menzies T. On the value of oversampling for deep learning in software defect prediction. *IEEE Trans Softw Eng*. 2022;48(8):3103-3116. doi:10.1109/TSE.2021.3079841
10. Fan Y, Xia X, da Costa DA, Lo D, Hassan AE, Li S. The impact of mislabeled changes by SZZ on just-in-time defect prediction. *IEEE Trans Softw Eng*. 2021;47(8):1559-1586. doi:10.1109/TSE.2019.2929761
11. Li K, Xiang Z, Chen T, Wang S, Tan KC. Understanding the automated parameter optimization on transfer learning for cross-project defect prediction: an empirical study. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. Association for Computing Machinery; 2020:566-577.
12. Chen J, Hu K, Yu Y, et al. Software visualization and deep transfer learning for effective software defect prediction. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. Association for Computing Machinery; 2020:578-589.
13. Li K, Xiang Z, Chen T, Tan KC. BiLO-CPDP: bi-level programming for automated model discovery in cross-project defect prediction. Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. Association for Computing Machinery; 2021:573-584.
14. Reyhani Hamedani M, Shin D, Lee M, Cho SJ, Hwang C. AndroClass: an effective method to classify android applications by applying deep neural networks to comprehensive features. *Wirel Commun Mob Comput*. 2018;2018:1250359. doi:10.1155/2018/1250359
15. Malhotra R. An empirical framework for defect prediction using machine learning techniques with android software. *Appl Soft Comput*. 2016;49:1034-1050. doi:10.1016/j.asoc.2016.04.032
16. Dong F, Wang J, Li Q, Xu G, Zhang S. Defect prediction in android binary executables using deep neural network. *Wirel Pers Commun*. 2018;102(3):2261-2285. doi:10.1007/s11277-017-5069-3
17. Sadaf S, Iqbal D, Buhnova B. AI-based software defect prediction for trustworthy android apps. Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022. Association for Computing Machinery; 2022:393-398.
18. Zhao K, Xu Z, Yan M, Tang Y, Fan M, Catolino G. Just-in-time defect prediction for android apps via imbalanced deep learning model. Proceedings of the 36th Annual ACM Symposium on Applied Computing. Association for Computing Machinery; 2021:1447-1454.
19. Zhao K, Xu Z, Yan M, Xue L, Li W, Catolino G. A compositional model for effort-aware just-in-time defect prediction on android apps. *IET Softw*. 2022;16(3):259-278. doi:10.1049/sfw2.12040
20. Kamei Y, Shihab E, Adams B, et al. A large-scale empirical study of just-in-time quality assurance. *IEEE Trans Softw Eng*. 2013;39(6):757-773. doi:10.1109/TSE.2012.70
21. Yang Y, Zhou Y, Liu J, et al. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Association for Computing Machinery; 2016:157-168.
22. Catolino G, Di Nucci D, Ferrucci F. Cross-project just-in-time bug prediction for mobile apps: an empirical assessment. Proceedings of the 2019 6th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft); 2019:99-110.

23. Cheng T, Zhao K, Sun S, Mateen M, Wen J. Effort-aware cross-project just-in-time defect prediction framework for mobile apps. *Front Comput Sci*. 2022;16(6):166207. doi:10.1007/s11704-021-1013-5
24. Sun T, Shao Y, Li X, et al. Learning sparse sharing architectures for multiple tasks. *Proc AAAI Conf Artif Intell*. 2020;34(05):8936-8943. doi:10.1609/aaai.v34i05.6424
25. Ding C, Wang K, Wang P, Tao D. Multi-task learning with coarse priors for robust part-aware person re-identification. *IEEE Trans Pattern Anal Mach Intell*. 2020;44:1474-1488.
26. Zhang Y, Yin C, Wu Q, He Q, Zhu H. Location-aware deep collaborative filtering for service recommendation. *IEEE Trans Syst Man Cybern Syst*. 2021;51(6):3796-3807. doi:10.1109/TSMC.2019.2931723
27. Torkamani M, Shankar S, Rooshenas A, Wallis P. Differential equation units: learning functional forms of activation functions from data. *Proc AAAI Conf Artif Intell*. 2020;34(04):6030-6037. doi:10.1609/aaai.v34i04.6065
28. Dubey SR, Singh SK, Chaudhuri BB. A comprehensive survey and performance analysis of activation functions in deep learning. CoRR:abs/2109.14545, 2021.
29. Isabella SJ, Srinivasan S, Suseendran G. A framework using binary cross entropy—gradient boost hybrid ensemble classifier for imbalanced data classification. *Webology*. 2021;18(1):104-120. doi:10.14704/WEB/V18I1/WEB18076
30. Ruder S. An overview of gradient descent optimization algorithms. CoRR:abs/1609.04747, 2016.
31. Xu X, Tian H, Zhang X, Qi L, He Q, Dou W. DisCOV: distributed COVID-19 detection on X-ray images with edge-cloud collaboration. *IEEE Trans Serv Comput*. 2022;15(3):1206-1219. doi:10.1109/TSC.2022.3142265
32. Moser R, Pedrycz W, Succi G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. Proceedings of the 30th International Conference on Software Engineering; 2008:181-190.
33. Herzig K, Just S, Zeller A. It's not a bug, it's a feature: how misclassification impacts bug prediction. Proceedings of the 2013 35th International Conference on Software Engineering (ICSE). IEEE; 2013:392-401.
34. Fukushima T, Kamei Y, McIntosh S, Yamashita K, Ubayashi N. An empirical study of just-in-time defect prediction using cross-project models. Proceedings of the 11th Working Conference on Mining Software Repositories. Association for Computing Machinery; 2014:172-181.
35. McIntosh S, Kamei Y. Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction. *IEEE Trans Softw Eng*. 2018;44(5):412-428. doi:10.1109/TSE.2017.2693980
36. Jing XY, 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(4):321-339. doi:10.1109/TSE.2016.2597849
37. Li Z, Jing XY, Wu F, Zhu X, Xu B, Ying S. Cost-sensitive transfer kernel canonical correlation analysis for heterogeneous defect prediction. *Autom Softw Eng*. 2018;25(2):201-245. doi:10.1007/s10515-017-0220-7
38. Arisholm E, Briand LC, Fuglerud M. Data mining techniques for building fault-proneness models in telecom java software. Proceedings of the 18th IEEE International Symposium on Software Reliability (ISSRE'07); 2007:215-224.
39. Rahman F, Posnett D, Devanbu P. Recalling the "imprecision" of cross-project defect prediction. Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. Association for Computing Machinery; 2012.
40. Wang S, Liu T, Tan L. Automatically learning semantic features for defect prediction. Proceedings of the 38th International Conference on Software Engineering. Association for Computing Machinery; 2016:297-308.
41. Qi L, Lin W, Zhang X, Dou W, Xu X, Chen J. A correlation graph based approach for personalized and compatible web APIs recommendation in mobile APP development. *IEEE Trans Knowl Data Eng*. 2022. doi:10.1109/TKDE.2022.3168611
42. Song Q, Guo Y, Shepperd M. A comprehensive investigation of the role of imbalanced learning for software defect prediction. *IEEE Trans Softw Eng*. 2019;45(12):1253-1269. doi:10.1109/TSE.2018.2836442
43. Li N, Shepperd M, Guo Y. A systematic review of unsupervised learning techniques for software defect prediction. *Inf Softw Technol*. 2020;122:106287. doi:10.1016/j.infsof.2020.106287
44. Wilcoxon F. Individual comparisons by ranking methods. In: Kotz S, Johnson NL, eds. *Breakthroughs in Statistics: Methodology and Distribution*. Springer; 1992:196-202.
45. Xu X, Jiang Q, Zhang P, et al. Game theory for distributed IoV task offloading with fuzzy neural network in edge computing. *IEEE Trans Fuzzy Syst*. 2022;30(11):4593-4604. doi:10.1109/TFUZZ.2022.3158000
46. Zhang Y, Cui G, Deng S, Chen F, Wang Y, He Q. Efficient query of quality correlation for service composition. *IEEE Trans Serv Comput*. 2021;14(3):695-709. doi:10.1109/TSC.2018.2830773
47. Cliff N. *Ordinal Methods for Behavioral Data Analysis*. Psychology Press; 2014.
48. Li Y, Liao C, Wang Y, Wang C. Energy-efficient optimal relay selection in cooperative cellular networks based on double auction. *IEEE Trans Wirel Commun*. 2015;14(8):4093-4104. doi:10.1109/TWC.2015.2416715
49. Zhang Q, Wang Y, Yin G, Tong X, Sai AMVV, Cai Z. Two-stage bilateral online priority assignment in spatio-temporal crowdsourcing. *IEEE Trans Serv Comput*. 2022;1-14. doi:10.1109/TSC.2022.3197676
50. Agrawal A, Fu W, Menzies T. What is wrong with topic modeling? And how to fix it using search-based software engineering. *Inf Softw Technol*. 2018;98:74-88. doi:10.1016/j.infsof.2018.02.005
51. Dai H, Yu J, Li M, et al. Bloom filter with noisy coding framework for multi-set membership testing. *IEEE Trans Knowl Data Eng*. 2022;1-14. doi:10.1109/TKDE.2022.3199646
52. Benjamini Y, Hochberg Y. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *J R Stat Soc B*. 1995;57(1):289-300. doi:10.1111/j.2517-6161.1995.tb02031.x
53. Xu R, Wang H. Multi-view learning with privileged weighted twin support vector machine. *Expert Syst Appl*. 2022;206:117787. doi:10.1016/j.eswa.2022.117787
54. Wang Q, Zhu C, Zhang Y, Zhong H, Zhong J, Sheng VS. Short text topic learning using heterogeneous information network. *IEEE Trans Knowl Data Eng*. 2022;1. doi:10.1109/TKDE.2022.3147766
55. Hein A, Jiang M, Thiyageswaran V, Guerzhoy M. Random forests for opponent hand estimation in gin rummy. *Proc AAAI Conf Artif Intell*. 2021;35(17):15545-15550. doi:10.1609/aaai.v35i17.17830

56. Koru AG, Zhang D, El Emam K, Liu H. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Trans Softw Eng*. 2009;35(2):293-304. doi:10.1109/TSE.2008.90
57. Giger E, D'Ambros M, Pinzger M, Gall HC. Method-level bug prediction. Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. Association for Computing Machinery; 2012:171-180.
58. Hata H, Mizuno O, Kikuno T. Bug prediction based on fine-grained module histories. Proceedings of the 34th International Conference on Software Engineering (ICSE); 2012:200-210.
59. Zimmermann T, Premraj R, Zeller A. Predicting defects for eclipse. Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007); 2007:9.
60. Hall T, Beecham S, Bowes D, Gray D, Counsell S. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans Softw Eng*. 2012;38(6):1276-1304. doi:10.1109/TSE.2011.103
61. Shihab E, Hassan AE, Adams B, Jiang ZM. An industrial study on the risk of software changes. Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. Association for Computing Machinery; 2012.
62. Mockus A, Weiss DM. Predicting risk of software changes. *Bell Labs Tech J*. 2000;5(2):169-180. doi:10.1002/bltj.2229
63. Kim S, Whitehead EJ, Zhang Y. Classifying software changes: clean or buggy? *IEEE Trans Softw Eng*. 2008;34(2):181-196. doi:10.1109/TSE.2007.70773
64. Zhang T, Yu Y, Mao X, Lu Y, Li Z, Wang H. FENSE: a feature-based ensemble modeling approach to cross-project just-in-time defect prediction. *Empir Softw Eng*. 2022;27(7):1-41.
65. Fu W, Menzies T. Revisiting unsupervised learning for defect prediction. Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. Association for Computing Machinery; 2017:72-83.
66. Liu J, Zhou Y, Yang Y, Lu H, Xu B. Code churn: a neglected metric in effort-aware just-in-time defect prediction. Proceedings of the 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE; 2017:11-19.

How to cite this article: Huang Q, Li Z, Gu Q. Multi-task deep neural networks for just-in-time software defect prediction on mobile apps. *Concurrency Computat Pract Exper*. 2023;e7664. doi: 10.1002/cpe.7664