Information and Software Technology 165 (2024) 107250

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Improving effort-aware defect prediction by directly learning to rank software modules

Xiao Yu [a,b], Jiqing Rao [a], Lei Liu [a], Guancheng Lin [a], Wenhua Hu [a], Jacky Wai Keung [c], Junwei Zhou [a], Jianwen Xiang [a,b,*]

[a] *Hubei Key Laboratory of Transportation of Internet of Things, School of Computer Science and Artificial Intelligence, Wuhan University of Technology, Wuhan, China*
[b] *Wuhan University of Technology Chongqing Research Institute, Chongqing, China*
[c] *Department of Computer Science, City University of Hong Kong, Hong Kong, China*

ARTICLE INFO

Keywords:
Software Defect Prediction
Effort aware
Genetic algorithm

ABSTRACT

Context: Effort-Aware Defect Prediction (EADP) ranks software modules according to the defect density of software modules, which allows testers to find more bugs while reviewing a certain amount of Lines Of Code (LOC). Most existing methods regard the EADP task as a regression or classification problem. Optimizing the regression loss or classification accuracy might result in poor effort-aware performance.
Objective: Therefore, we propose a method called EALTR to improve the EADP performance by directly maximizing the Proportion of the found Bugs (PofB@20%) value when inspecting the top 20% LOC.
Method: EALTR uses the linear regression model to build the EADP model, and then employs the composite differential evolution algorithm to generate a set of coefficient vectors for the linear regression model. Finally, EALTR selects the coefficient vector that achieves the highest PofB@20% value on the training dataset to construct the EADP model. To further reduce the Initial False Alarms (IFA) value of EALTR, we propose a re-ranking strategy in the prediction phase.
Results: Our experimental results on eleven project datasets with 41 releases show that EALTR can find 5.83%–54.47% more bugs than the baseline methods whose IFA values are less than 10 and the re-ranking strategy significantly reduces the IFA value by 16.95%.
Conclusion: Our study verifies the effectiveness of directly optimizing the effort-aware metric (i.e., PofB@20%) to build the EADP model. EALTR is recommended as an effective EADP method, since it can help software testers find more bugs.

1. Introduction

*S*oftware *D*efect *P*rediction (SDP) has been concerned by more and more researchers in recent years [1–9]. The SDP technique predicts whether the new software modules are faulty or not based on some software features, e.g., *L*ines *O*f *C*ode (LOC) and code complexity. Accurate SDP prediction results assist software testers in assigning limited testing resources by focusing on the predicted faulty modules [10–13] and fault localization [14–18]. However, the traditional binary classification-based SDP models are not enough to offer a more practical guide for software testing, since they cannot differentiate the modules with different defect densities [19,20]. Obviously, software testers should allocate different efforts to inspect the modules with different defect densities. But the modules are considered the same and are assigned equal testing resources according to the binary classification-based SDP models [21,22]. Hence, Mende et al. [23] proposed the *E*ffort-*A*ware *D*efect *P*rediction (EADP) model to sort software modules by their defect density. It allows software testers to detect more bugs when testing a certain number of LOC and allocate testing resources more effectively [24,25].

1.1. Motivation

Researchers have proposed numerous EADP methods to rank new software modules based on the defect density, which considers the EADP task as a regression [26] or classification [19,27,28] problem.

* Corresponding author at: Hubei Key Laboratory of Transportation of Internet of Things, School of Computer Science and Artificial Intelligence, Wuhan University of Technology, Wuhan, China.
E-mail addresses: xiaoyu@whut.edu.cn (X. Yu), jqrao@whut.edu.cn (J. Rao), leiliu_cs@hotmail.com (L. Liu), gclin@whut.edu.cn (G. Lin), whu10@whut.edu.cn (W. Hu), jacky.keung@cityu.edu.hk (J.W. Keung), junweizhou@whut.edu.cn (J. Zhou), jwxiang@whut.edu.cn (J. Xiang).

https://doi.org/10.1016/j.infsof.2023.107250
Received 7 September 2022; Received in revised form 22 March 2023; Accepted 8 May 2023
Available online 13 May 2023

For example, Kamei et al. [26] proposed the EALR method, which used the linear regression model to build the relationship between the defect density and software features and employed the least square method to minimize the total difference between the actual defect density and the predicted density. However, the EALR model with higher prediction accuracy might lead to a worse ranking [29].

Example 1. Assuming that a developed project includes the three software modules, i.e., M_1, M_2, and M_3, whose defect densities are 0.3, 0.2, and 0, and LOC are 100, 150, and 50. The regression model A predicts that the defect densities of the three modules are 0.3, 0.05, and 0.1, whereas the regression model B predicts that their defect densities are 0.6, 0.05, and 0.01. Although the model A achieves a smaller total difference between the actual defect density and the predicted density, the ranking of the three modules according to the predicted densities by the model B is what the software testers desire for EADP.

The recently proposed CBS+ [27] and EASC [28] methods firstly use the classification algorithms (i.e., logistic regression and naive Bayes) to predict the possibility of modules being defective. Then, the methods suggest software testers first inspect the predicted defective modules in order according to the ratio of the predicted defective possibility to LOC of each module. When there is the remaining testing effort, software testers can inspect the predicted non-defective modules according to the ratio of the predicted defective possibility to LOC of each module until the testing resource runs out. In other words, CBS+ and EASC mainly regard the EADP task as a classification problem.

Example 2. The classifier C predicts that the defective probabilities of the three modules in Example 1 are 0.55, 0.9, and 0.1, whereas the classifier D predicts that the defective probabilities are 0.55, 0.3, and 0.1. The classifier C achieves higher binary classification accuracy (i.e., predicting the class labels of all three modules correctly), and the classifier D wrongly predicts the defect-proneness of M_2. When CBS+ and EASC use the classifier C to predict the defective possibility of modules, the predicted ranking of the three modules is M_2, M_1, and M_3. But CBS+ and EASC embedding the classifier D can sort the three modules correctly.

In summary, the existing EADP works considering the EADP task as a regression or classification problem might result in poor effort-aware performance.

1.2. Our work and contributions

To address the problem, we propose a method called *E*ffort-*A*ware *L*earning-*T*o-*R*ank (EALTR), which regards the EADP task as a ranking problem and constructs the EADP model by directly maximizing the *P*roportion *of* the found *B*ugs (PofB@20%) when inspecting the top 20% LOC. Specifically, EALTR uses the linear regression model to build the relationship between the defect density and software features. Then, it employs the composite differential evolution [30] algorithm to generate a set of coefficient vectors for the linear regression model by maximizing the PofB@20% value. Finally, it selects the coefficient vector that achieves the highest PofB@20% value on the training dataset to construct the linear regression model. In the prediction phase, we use the linear regression model to predict the defect density of new software modules and sort the modules according to the predicted values.

To further reduce the false alarms of EALTR, we propose a re-ranking strategy in the prediction phase. We place the top-ranked modules whose LOC are less than a threshold at the end of the ranking, if the module ranked first is non-defective or the Precision@20% value is less than 0.1 according to EALTR's predicted results on the training dataset. We call the EALTR method integrated with the re-ranking strategy EALTR*.

We compare EALTR with the six state-of-the-art EADP methods, i.e., DEJIT [31], ManualUp [32], EALR [26], CBS+ [27], EASC [28], and EATT [19]. In addition, we replace *R*andom *F*orest (RF) and *D*eep *F*orest (DF) [33] with their *L*ogistic *R*egression (LR) and *N*aive *B*ayes (NB) classifiers embedded in CBS+ and EASC, and build the EADP models (called CBS+(RF) and CBS+(DF)), because RF and DF have achieved a good SDP performance [27,28,34,35]. The results on 41 version datasets indicate that (1) The average IFA values of ManualUp and EATT are greater than 10, and software testers are reluctant to use the EADP models. (2) EALTR improves the average PofB@20% values of DEJIT by 5.83%, of CBS+ by 32.97%, of EASC by 54.47%, of CBS+(RF) by 19.02%, of CBS+(DF) by 15.97%, and of EALR by 25.17%, respectively. (3) The re-ranking strategy can reduce the average IFA value of EALTR by 16.95%.

We make the following main contributions:

- We propose the EALTR method to build the EADP model by directly optimizing the PofB@20% value, which can help software testers find more bugs when testing a certain amount of LOC.
- We propose the re-ranking strategy to further reduce initial false alarms of EALTR.

1.3. Our extensions

This article is an extended version of our previous work published in QRS 2021 - Companion [36] by adding the following updates:

(1) We propose EALTR*, which is an extended version of EALTR. The main difference is that we integrate a re-ranking strategy into EALTR to reduce the IFA values of EALTR. The proposed strategy places the top-ranked modules whose LOC are less than a threshold at the end of the ranking, if the trained EALTR model may have poor performance on the testing dataset in terms of IFA and Precision@20%. The results indicate that the re-ranking strategy can enhance the performance of EALTR in terms of IFA and Precision@20%.

(2) We further discuss the application of genetic algorithms to SDP in the related work.

(3) We perform a comprehensive comparison of EALTR against more baseline methods, including DEJIT, ManualUp, EALR, and EATT. In our previous study, we compared EALTR with only two methods (i.e., CBS+ and EASC). In this article, we firstly compare EALTR with DEJIT, which also uses differential evolution algorithm, but its optimized metric is *D*ensity-*P*ercentile-*A*verage (DPA) [31]. Then, we compare it with EALR optimized by the least square method to validate whether directly optimizing the PofB@20% value can build better EADP models. After that, we compare it with CBS+, EASC, and EATT, which regard the EADP task as a classification task. Finally, we compare it with an unsupervised method ManualUp, which achieved the best performance in terms of Recall@20% and PofB@20% in previous studies [37].

(4) We discuss the execution time of EALTR and EALTR*, since they use the time-consuming genetic algorithm to construct the EADP model. The results show that the time cost is acceptable.

(5) We investigate the performance of EATLR and EALTR* when testing the top 10% LOC, since testing resources may be able to inspect only the first 10% LOC. The experimental results show that EALTR and EALTR* still can find more bugs.

(6) We discuss the impact of two parameters on EALTR*.

(7) We further investigate and discuss the performance of EALTR and EALTR* in cross-project settings. The experimental results on 11 cross-project training and testing pairs still show that EALTR and EALTR* outperform the baseline methods.

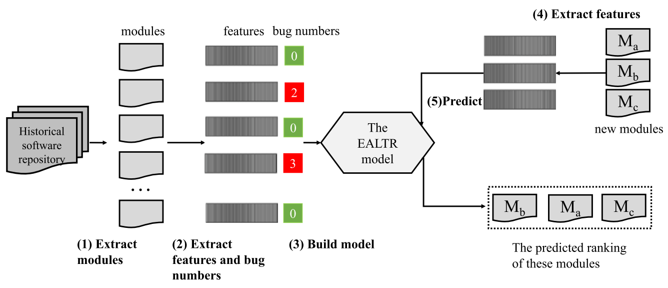

Fig. 1. The process of the EALTR model.

1.4. Organization

The rest of this paper is organized as follows. In Section 2, we describe our EALTR method and the re-ranking strategy. Sections 3 and 4 present the experimental setup and the experimental results, respectively. Section 5 discusses the impact of parameters and experimental scenario on EALTR and the potential threats to validity. Section 6 introduces the related work in EADP. Section 7 draws the conclusion.

2. The proposed approach

2.1. Overview

The EADP process contains the following five steps as displayed in Fig. 1. In the first step, the software modules are extracted from the historical software repository. The software features $\boldsymbol{x}_i$ and bug number y_i are then extracted in the second step. Therefore, a software module M_i could be denoted as $M_i = (\boldsymbol{x}_i, y_i)$. The whole training dataset containing n modules could be denoted as

$$D = \left\{M_1, M_2, \dots, M_n\right\}. \tag{1}$$

In the third step, the EALTR model is constructed based on D, which is a mixed set of defective and non-defective modules. Since Weyukeret et al. [38] indicated that linear models and additive models are good and realistic for SDP, we also use the linear regression model to build the EADP model:

$$y = f(\alpha, \mathbf{x}) = \sum_{i=1}^{d} \alpha_j x_j, \tag{2}$$

where x_j is the jth software feature value of the software module with d features, y is the predicted defect density, and the parameter α are obtained from the training software modules with known defect density. After we determine the parameter α, our EALTR model is constructed and can be used for prediction. In the fourth step, we extract the same features from new software modules. In the last step, we employ the trained EALTR model to predict the defect densities of the new software modules, then sort the modules according to the predicted values. For example, the predicted defect densities of the three new modules (i.e., M_a, M_b, and M_c) are y_a, y_b, and y_c, and $y_b > y_a > y_c$. Therefore, the final ranking is M_b, M_a, and M_c.

2.2. Model solving

To obtain the parameter α, researchers [38,39] usually use the least square method to minimize the total difference between the actual defect density and the predicted density. However, the trained EADP model optimized by the least square method may not achieve a good ranking evaluated by the EADP evaluation metrics (e.g., PofB@20%) as shown in Example 1. Since the primary objective of EADP is to detect more bugs when testing a certain number of LOC, we build the EALTR model by directly optimizing the effort-aware metric (i.e., PofB@20%).

Since the effort-aware metrics of EADP models are usually non-differentiable, we use the genetic algorithm to obtain the parameter α. In particular, we adopt the composite differential evolution algorithm [30], which has been widely used in the software engineering research area and shows good performance [29]. Algorithm 1 shows the pseudocode of using the composite differential evolution algorithm to obtain the parameter α of the linear model. We firstly create m solutions at random to compose P_0, and calculate the PofB@20% value of each solution in P_0 (Lines 1–2). Then, we perform t_{max} iterations to evolve the population. In each iteration, we conduct the selection, crossover, and mutation operations for each solution in the current population to create the new population (Line 5). Next, we calculate the PofB@20% value of each solution in the new population (Line 6). Finally, we return the best solution α that achieves the highest PofB@20% value (Line 9). We set $m = 100$ and $t_{max} = 100$, because preliminary experiments show higher m and t_{max} values do not significantly promote the performance of EALTR.

Algorithm 1: Estimation of α

```
input: Training dataset, $D$
    Objective function, PofB@20%
    Number of solutions in a population, $m$
    Number of maximal generation, $t_{max}$
output: $\alpha$
1: $P_0$=$m$ generated solutions randomly;
2: Calculate the PofB@20% value of each solution in $P_0$;
3: Set the generation number $t$=1;
4: while $t<t_{max}$ do
5:     For each solution in $P_{t-1}$, do selection, crossover, and
       mutation to generate new population $P_t$;
6:     Calculate the PofB@20% value of each solution in $P_t$;
7:     $t$=$t$+1;
8: end while
9: return The best solution $\alpha$;
```

2.3. Re-ranking

The experimental results in Section 4.1 show EALTR does not achieve the best performance in terms of IFA. If the top-ranked modules recommended by the EADP model are all actually non-defective, software testers would stop inspecting subsequent predicted defective modules. Therefore, we propose a re-ranking strategy to further reduce the IFA value, and call the EALTR method integrated with the re-ranking strategy EALTR*.

Specifically, we consider the trained EALTR model may have poor performance on the testing dataset in terms of IFA, if the module ranked first is non-defective or the Precision@20% value is less than 0.1 according to its predicted results on the training dataset. Then, we sort the non-defective modules in the training dataset by LOC from small to large, and then select the LOC of the module at the position of proportion p as the threshold. Next, we use the trained EALTR model to predict the ranking of the modules in the testing dataset. If the LOC of the top-ranked q modules are less than the threshold, the modules are considered to be non-defective. Finally, the modules are placed at the end of the ranking.

We show a simple example in Fig. 2. Suppose the module ranked first is non-defective or the Precision@20% value of the trained EALTR model on the training dataset is less than 0.1, and $p = 1/2$, that is, the median LOC value of all non-defective modules in the training dataset (110) is set as the threshold. Therefore, M_g and M_e in the testing dataset are placed at the end of the ranking.

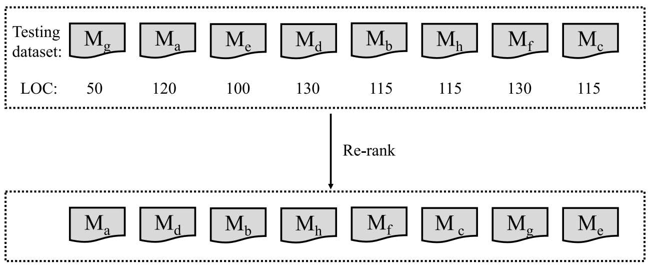
Testing dataset:
M_g M_a M_e M_d M_b M_h M_f M_c
LOC: 50 120 100 130 115 115 130 115
Re-rank
M_a M_d M_b M_h M_f M_c M_g M_e

Fig. 2. The example of the re-ranking strategy.

3. Experimental setup

3.1. Datasets

We employ the more practical cross-version validation to conduct the experiments. In addition, since the EALTR method aims to find more bugs, we only select the defect datasets with the information of bug numbers. Therefore, we use 41 versions of 11 software projects from the PROMISE repository [40]. The detailed information of the datasets is displayed in Table 1, where #Modules is the number of modules in the dataset, #Bugs is the total number of bugs, %Faulty is the percentage of faulty modules, and Avg is the average number of bugs. The datasets contain 20 software features. Due to the space limit, please refer to [41] for detailed information.

3.2. Evaluation metrics

We restrict our effort to 20% of the total effort in our study, i.e., inspecting 20% LOC. The number 20% is commonly used as a cutoff value to set the effort required for the software testing [37,42–44]. Suppose there is a dataset with M modules, N bugs, and P LOC. Among the M modules, K modules are defective. When testing 20% LOC, we have inspected m modules containing n bugs. Among the m modules, k modules are defective. In addition, when testers detect the first actual defective module, they have inspected m' modules. We use the following evaluation metrics in the experiments, which are also widely used in the fields of artificial intelligence [45–49] and software engineering [21,50–54].

Recall@20% is the proportion of the inspected actual defective modules among all the actual defective modules in the dataset. A higher Recall@20% value denotes that more defective software modules could be found.

$$Recall@20\% = k/K \tag{3}$$

PofB@20% is the *P*roportion *of* the found *B*ugs among all bugs in the dataset. A higher PofB@20% value indicates that more bugs could be detected. PofB@20% is equal to Recall@20%, when every defective module only contains one bug. Previous studies [27,55] denoted PofB@20% as Recall@20%, since they regarded the class label of modules as the number of defects.

$$PofB@20\% = n/N \tag{4}$$

Precision@20% is the proportion of the inspected actual defective modules among all the inspected modules. A lower Precision@20% value denotes that software testers will meet more false alarms, thus affecting their confidence in the EADP model negatively [27].

$$Precision@20\% = k/m \tag{5}$$

PMI@20% is the *P*roportion of *M*odule *I*nspected. A higher PMI@20% value indicates that under the same number of LOC (i.e., 20% LOC) to inspect, software testers need to inspect more modules.

$$PMI@20\% = m/M \tag{6}$$

Table 1
The details of the experimental datasets.

Project	Version	Module	#Bugs	%Faulty	Avg
Ant	1.3	125	33	16	1.65
	1.4	178	47	22.5	1.18
	1.5	293	35	10.9	1.09
	1.6	351	184	26.2	2
	1.7	745	338	22.3	2.04
Camel	1.0	339	14	3.8	1.08
	1.2	608	522	35.5	2.42
	1.4	872	335	16.6	2.31
	1.6	965	500	19.5	2.66
Ivy	1.1	111	233	56.8	3.7
	1.4	241	18	6.6	1.12
	2.0	352	56	11.4	1.4
Jedit	3.2	372	382	33.1	4.24
	4.0	306	226	24.5	3.01
	4.1	312	217	25.3	2.75
	4.2	367	106	13.1	2.21
	4.3	492	12	2.2	1.09
Log4j	1.0	135	61	25.2	1.79
	1.1	109	86	33.9	2.32
	1.2	205	498	92.2	2.63
Lucene	2.0	195	268	46.7	2.95
	2.2	247	414	58.3	2.88
	2.4	340	632	59.7	3.11
Poi	1.5	237	342	59.5	2.43
	2.0	314	39	11.8	1.05
	2.5	385	496	64.4	2
	3.0	442	500	63.6	1.78
Synapse	1.0	157	21	10.2	1.31
	1.1	222	99	27	1.65
	1.2	256	145	33.6	1.69
Velocity	1.4	196	210	75	1.43
	1.5	214	331	66.4	2.33
	1.6	229	190	34.1	2.44
Xalan	2.4	723	156	15.2	1.42
	2.5	803	531	48.2	1.37
	2.6	885	625	46.4	1.52
	2.7	909	1213	98.8	1.35
Xerces	init	162	167	47.5	2.17
	1.2	440	115	16.1	1.62
	1.3	453	193	15.2	2.8
	1.4	588	1596	74.3	3.65

IFA is the number of *I*nitial *F*alse *A*larms encountered before software testers detect the first bug. A higher IFA value denotes that software testers require to inspect more modules to find the first bug.

$$IFA = m' \tag{7}$$

Popt is based on the cumulative lift chart shown in Fig. 3. In the chart, the x-axis is the cumulative percentage of LOC to inspect, and the y-axis is the cumulative percentage of found bugs. There are three curves in the chart, corresponding to the prediction model, the optimal model, and the worst model. Software modules are ranked by the decreasing predicted defect densities according to the prediction model, software modules are ranked by the decreasing actual defect densities according to the optimal model, and software modules are ranked by the increasing actual defect densities according to the worst model. *Popt* is computed as follows:

$$Popt = \frac{\text{Area(prediction)} - \text{Area(worst)}}{\text{Area(optimal)} - \text{Area(worst)}} \tag{8}$$

where Area() represents the area under the corresponding curve. A larger *Popt* value denotes a smaller difference between the prediction model and the optimal model. Different from the above evaluation measures, *Popt* evaluates the global ranking of the predicted software modules.

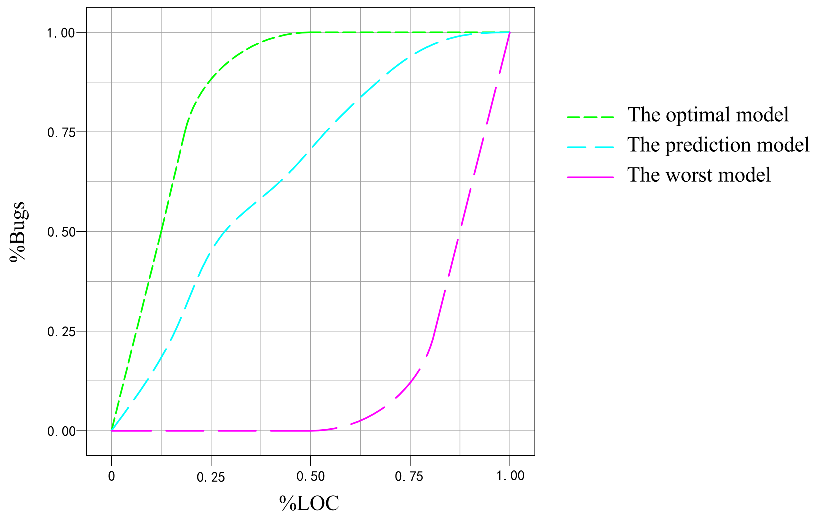

Fig. 3. A cumulative lift chart.

3.3. Baseline methods

To verify the effectiveness of EALTR, we compare it with the following state-of-the-art EADP methods.

(1) DEJIT [31]: It also uses a differential evolution algorithm but employs the DPA metric as the optimization metric for the algorithm. In this paper, we use the same experimental conditions as EALTR to implement DEJIT, in order to compare the differences in results between optimizing PofB@20% and DPA metrics.

(2) CBS+ [27]: It firstly identifies potentially defective and non-defective software modules using the trained Logistic Regression (LR) model. Then, it separately sorts the predicted defective modules and non-defective modules by the ratio between the predicted defect probability and LOC. Finally, it appends the sorted predicted non-defective modules at the end of the sorted defective ones.

(3) EASC [28]: It has the same procedure as CBS+. The only difference is that EASC employs Naive Bayes (NB) to predict whether software modules are defective, while CBS+ uses logistic regression. In addition, we replace RF and DF [33] with their LR and NB classifiers embedded in CBS+ and EASC, and construct EADP models (called CBS+(RF) and CBS+(DF)), because RF and DF have achieved good performance in SDP [28,34,56].

(4) EATT [19]: It firstly employs the tri-training method to train three classifiers. Then, it uses the majority voting strategy to calculate the defect probability of a new software module. Finally, it ranks all new modules by the ratio between the defect probability and LOC. We compare EALTR with CBS+, EASC, CBS+(RF), CBS+(DF), and EATT, which regard the EADP task as a classification task.

(5) ManualUp [32]: It considers that the modules with fewer LOC are more likely to be defective, so it sorts the modules in ascending order of LOC.

(6) EALR [26]: It assumes that there is a linear relationship between the defect density and the feature values. Therefore, it builds the same linear regression model as EALTR, but uses the least squares method to obtain the parameter solution with the optimization objective of minimizing the total difference between actual defect densities and predicted defect densities. We compare EALTR with EALR to validate whether directly optimizing the PofB@20% value can build better EADP models.

3.4. Experimental procedure

We employ the more practical cross-version validation [57] in the experiments. For example, we utilize Ant 1.3 as the training data and

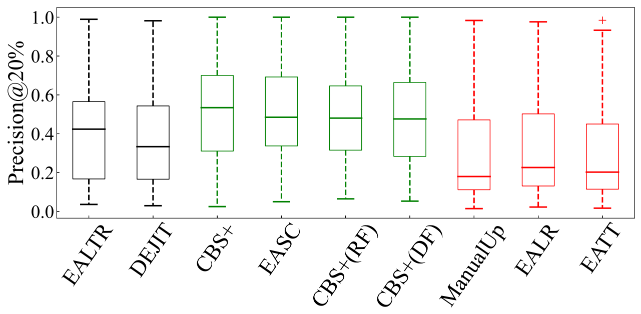

Fig. 4. The boxplot of the Precision@20% values of the nine methods. (The red boxplot indicates EALTR significantly outperforms the corresponding method, while the green boxplot indicates EALTR significantly performs worse than the corresponding method.).

Ant 1.4 as the testing data. We run all EADP methods 20 times on each training and testing pair and calculate the average value to avoid sample biases.

The Wilcoxon signed-rank test [58] is used in our experiments to analyze the significance of the difference between two different EADP methods on all testing datasets. Because multiple comparisons with EALTR are performed, the *B*enjamini–*H*ochberg (BH) [59] procedure is also used to adjust p-values in the experiments. If the *p*-value after BH correction is less than 0.05, it indicates that there is a statistically significant difference between the two methods.

4. Experimental results

We evaluate the performance of our EALTR model by answering the following three *R*esearch *Q*uestions (RQs).

4.1. RQ1: Does EALTR outperform state-of-the-art EADP methods?

Motivations: We propose the EALTR method, which constructs the EADP model by directly optimizing the PofB@20% value. To verify the superiority of EALTR, we compare it with DEJIT, CBS+, EASC, CBS+(RF), CBS+(DF), ManualUp, EALR, and EATT.

Methods: We analyze the performance results of the nine EADP methods in terms of the six evaluation metrics. Tables 2, 3, 4, and 5 present the detailed IFA, PofB@20%, *Popt*, and Recall@20% values on each cross-version pair. The bold values in each row in the tables indicate that the corresponding method obtains the best performance on the cross-version pair. The row W/D/L (Win/Draw/Loss) indicates

Table 2
The IFA value of the nine methods on each cross-version experiment when inspecting the top 20% LOC.

Cross-version	EALTR	DEJIT	CBS+	EASC	CBS+(RF)	CBS+(DF)	ManualUp	EALR	EATT
Ant1.3–1.4	7.9	1.65	2	4	1.35	1.3	**1**	2	1.4
Ant1.4–1.5	10.1	17.15	**3**	47	22.45	16.65	39	71	43.2
Ant1.5–1.6	0.3	**0**	**0**	17	3.1	2.7	54	3	49.55
Ant1.6–1.7	0.6	**0**	5	8	5.75	4.8	59	**0**	55.8
Camel1.0–1.2	**0**	**0**	1	2	2.5	2.4	4	**0**	13.45
Camel1.2–1.4	0.75	8.7	3	5	15.5	11.15	71	**0**	8.55
Camel1.4–1.6	**0**	**0**	**0**	3	4.65	1.7	7	**0**	24
Ivy1.1–1.4	15.35	21.75	1	15	**0.65**	4.1	17	35	18.25
Ivy1.4–2.0	19.75	**0.8**	39	9	12.2	12.75	38	1	36.45
Jedit3.2–4.0	0.3	**0.15**	7	6	3.1	2.3	15	7	17.85
Jedit4.0–4.1	0.05	0.25	1	6	3	1.75	8	**0**	13.3
Jedit4.1–4.2	**0.45**	5.95	8	7	6.2	1.7	32	27	31.95
Jedit4.2–4.3	**1.25**	1.95	8	21	4.95	4.2	66	66	71.9
Log4j1.0–1.1	**0**	**0**	3	3	2.95	2.95	3	2	2.5
Log4j1.1–1.2	0	0	0	0	0	0	0	0	0
Lucene2.0–2.2	0.05	1	2	1	1	0.95	2	**0**	2.5
Lucene2.2–2.4	**0**	**0**	**0**	**0**	1.3	1.35	1	1	0.45
Poi1.5–2.0	11.05	16.45	3	5	13.25	11.95	23	**2**	25.4
Poi2.0–2.5	0.3	**0**	**0**	1	0.95	0.75	17	**0**	19
Poi2.5–3.0	4.2	1.4	3	2	2.05	**0.5**	1	17	7.4
Synapse1.0–1.1	1	4.05	1	4	2.4	0.75	1	**0**	1.3
Synapse1.1–1.2	0.15	**0**	4	**0**	3.9	4.4	9	5	9.55
Velocity1.4–1.5	0.05	**0**	**0**	**0**	**0**	**0**	**0**	**0**	2.05
Velocity1.5–1.6	6.4	10.75	**0**	1	10	10.15	13	13	17.15
Xalan2.4–2.5	1.55	**0**	**0**	1	**0**	0.15	4	**0**	7.7
Xalan2.5–2.6	7	20.15	11	**2**	3.6	3.7	6	**2**	11.9
Xalan2.6–2.7	0	0	0	0	0	0	0	0	0
Xerces init-1.2	8.1	1.4	1	10	1.55	1.45	2	**0**	1.55
Xerces1.2–1.3	14.45	9.75	1	3	6.75	11.9	**0**	8	18.8
Xerces1.3–1.4	0.6	0.05	**0**	**0**	**0**	**0**	3	**0**	0.25
Average	3.723	4.112	**3.567**	6.100	4.503	3.948	16.533	8.733	17.105
W/D/L		11/6/13	12/5/13	15/3/12	18/2/10	17/2/11	21/3/6	11/4/15	25/2/3
p-value		0.694	0.968	0.612	0.529	0.746	0.005	0.745	0.000

the number of cross-version pairs, on which EALTR achieves a better, equal, or worse performance than other methods. Figs. 4 and 5 show the distribution of the Precision@20% and PMI@20% values of the nine methods across all cross-version pairs, and Table 6 presents the average Precision@20% and PMI@20% values of the nine methods.

Results: From these tables and figures, we have the following observations:

(1) As shown in Table 2, the average **IFA** values of ManualUp and EATT are greater than 10, and it is generally not acceptable that the top 10 software modules recommended by EADP models are all actually non-defective [27]. The adjusted p-values indicate there is a significant difference between EALTR and ManualUp, EATT. Except for the above two methods, EALTR obtains the second-best average performance among the nine methods, and CBS+ achieves the best performance. EALTR wins DEJIT, EASC, CBS+(RF), CBS+(DF), ManualUp, EALR and EATT on 11, 15, 18, 17, 21, 11, and 25 datasets respectively.

(2) As shown in Table 3, except for ManualUp and EATT whose IFA values are greater than 10, EALTR obtains the best average performance in terms of **PofB@20%**. EALTR wins DEJIT, CBS+, EASC, CBS+(RF), CBS+(DF), and EALR on 18, 23, 26, 22, 21, and 23 datasets, respectively. EALTR improves the average PofB@20% values of DEJIT by 5.83%, of CBS+ by 32.97%, of EASC by 54.47%, of CBS+(RF) by 19.02%, of CBS+(DF) by 15.97%, and of EALR by 25.17%, respectively. According to the adjusted p-values, EALTR significantly outperforms CBS+, EASC, CBS+(RF), CBS+(DF), and EALR. DEJIT employs the same differential evolution algorithm but is with optimization of the DPA metric, and there is no statistically significant difference between it and EALTR. However, EALTR exhibits superior performance and outperforms DEJIT on the majority of datasets.

(3) As shown in Table 4, except for ManualUp and EATT, EALTR obtains the best average ***Popt*** value and wins the other six baseline

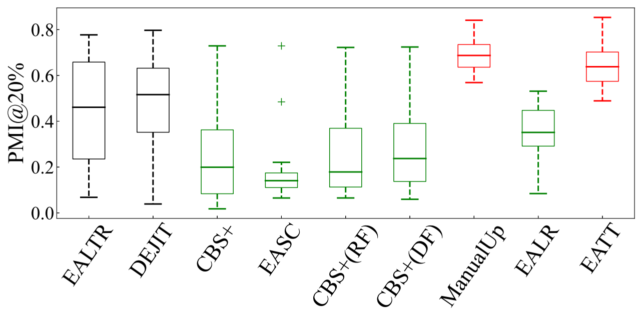

Fig. 5. The boxplot of the PMI@20% values of the nine methods. (The red boxplot indicates EALTR significantly outperforms the corresponding method, while the green boxplot indicates EALTR significantly performs worse than the corresponding method.).

methods on 17, 20, 26, 20, 21, and 21 datasets, respectively. EALTR significantly performs better than CBS+, EASC, CBS+(RF), CBS+(DF), and EALR according to the adjusted p-values, and can improve their average value by 10.77%, 37.73%, 10.05%, 10.05%, and 12.60%, respectively. Compared with DEJIT, EALTR also demonstrates superior average values, exhibiting an enhancement of 2.26%.

(4) As shown in Table 5, except for ManualUp and EATT, DEJIT obtains the best average value on **Recall@20%**, but there is no significant difference between it and EALTR, with an average performance difference of only 0.77%. In comparison to the other five baseline methods, EALTR outperforms them on 20, 22, 19, 18, and 19 datasets, respectively. EALTR improves the average Recall@20% value of CBS+ by 37.50%, of EASC by 59.75%, of CBS+(RF) by 21.84%, of CBS+(DF) by 19.94%, of EALR by 27.91%, respectively. The adjusted p-values

Table 3
The PofB@20% value of the nine methods on each cross-version experiment when inspecting the top 20% LOC.

Cross-version	EALTR	DEJIT	CBS+	EASC	CBS+(RF)	CBS+(DF)	ManualUp	EALR	EATT
Ant1.3–1.4	0.226	0.112	0.128	0.234	0.163	0.203	**0.468**	0.043	0.441
Ant1.4–1.5	0.280	0.277	0.314	0.229	0.254	0.309	0.286	**0.343**	0.254
Ant1.5–1.6	0.230	0.230	0.185	**0.234**	0.168	0.192	0.152	0.141	0.161
Ant1.6–1.7	0.242	0.243	0.254	0.195	0.243	0.245	0.175	**0.281**	0.186
Camel1.0–1.2	0.250	0.191	0.272	0.186	0.257	0.224	**0.410**	0.255	0.357
Camel1.2–1.4	0.329	0.320	0.236	0.263	**0.465**	0.455	0.331	0.301	0.326
Camel1.4–1.6	0.383	0.328	0.316	0.288	0.390	**0.391**	0.370	0.306	0.366
Ivy1.1–1.4	**0.325**	0.253	0.222	0.222	0.292	0.300	0.278	0.222	0.306
Ivy1.4–2.0	0.229	0.266	0.107	**0.286**	0.208	0.192	0.196	0.250	0.209
Jedit3.2–4.0	0.340	**0.349**	0.257	0.261	0.313	0.313	0.199	0.292	0.264
Jedit4.0–4.1	0.331	**0.373**	0.309	0.341	0.366	0.354	0.235	0.313	0.302
Jedit4.1–4.2	**0.335**	0.275	0.198	0.245	0.245	0.242	0.160	0.226	0.231
Jedit4.2–4.3	0.471	0.279	0.250	0.167	0.396	0.400	0.417	0.167	**0.508**
Log4j1.0–1.1	0.420	0.390	0.360	0.407	0.350	0.391	0.163	**0.453**	0.297
Log4j1.1–1.2	0.143	0.371	0.175	0.205	0.175	0.187	**0.546**	0.319	0.475
Lucene2.0–2.2	0.355	0.362	0.254	0.215	0.277	0.237	**0.386**	0.309	0.364
Lucene2.2–2.4	0.304	0.408	0.381	0.304	0.409	0.411	0.473	0.373	**0.479**
Poi1.5–2.0	0.377	0.269	0.231	0.205	0.264	0.286	**0.462**	0.359	0.301
Poi2.0–2.5	0.280	0.244	0.101	0.107	0.162	0.148	**0.494**	0.143	0.318
Poi2.5–3.0	0.404	0.322	0.392	0.174	0.345	0.347	**0.422**	0.306	0.387
Synapse1.0–1.1	0.247	0.253	0.253	0.232	0.246	0.276	0.273	0.202	**0.307**
Synapse1.1–1.2	0.307	**0.361**	0.228	0.186	0.234	0.219	0.310	0.303	0.274
Velocity1.4–1.5	0.512	0.461	**0.535**	0.492	0.519	0.524	0.480	0.181	0.503
Velocity1.5–1.6	**0.467**	0.401	0.426	0.237	0.409	0.383	0.458	0.284	0.459
Xalan2.4–2.5	0.514	0.429	0.171	0.147	0.161	0.172	0.539	0.405	**0.541**
Xalan2.5–2.6	0.427	0.433	0.317	0.269	0.397	0.387	**0.474**	0.307	0.469
Xalan2.6–2.7	0.583	0.553	0.261	0.203	0.341	0.334	**0.635**	0.378	0.620
Xerces init-1.2	0.693	0.695	0.652	0.148	0.682	0.636	0.730	0.591	**0.787**
Xerces1.2–1.3	0.436	0.399	0.176	0.166	0.220	0.418	**0.466**	0.326	0.463
Xerces1.3–1.4	0.409	0.432	0.215	0.216	0.213	0.213	0.451	0.327	**0.464**
Average	0.363	0.343	0.273	0.235	0.305	0.313	**0.381**	0.290	**0.381**
W/D/L		18/0/12	23/0/7	26/0/4	22/0/8	21/0/9	12/0/18	23/0/7	16/0/14
p-value		0.100	0.000	0.000	0.006	0.007	0.572	0.003	0.637

Table 4
The *Popt* value of the nine methods on each cross-version experiment.

Cross-version	EALTR	DEJIT	CBS+	EASC	CBS+(RF)	CBS+(DF)	ManualUp	EALR	EATT
Ant1.3–1.4	0.621	0.468	0.535	0.380	0.488	0.540	**0.759**	0.388	0.758
Ant1.4–1.5	0.587	0.594	**0.696**	0.604	0.629	0.663	0.559	0.624	0.573
Ant1.5–1.6	0.565	0.555	**0.589**	0.560	0.552	0.560	0.493	0.481	0.506
Ant1.6–1.7	0.563	0.589	**0.609**	0.537	0.592	0.588	0.452	0.607	0.464
Camel1.0–1.2	0.536	0.481	0.570	0.483	0.498	0.470	0.650	0.585	**0.654**
Camel1.2–1.4	0.564	0.589	0.601	0.542	**0.749**	0.742	0.568	0.546	0.585
Camel1.4–1.6	0.630	0.600	0.587	0.522	**0.709**	0.703	0.623	0.597	0.636
Ivy1.1–1.4	**0.583**	0.535	0.503	0.509	0.532	0.536	0.580	0.487	0.578
Ivy1.4–2.0	0.518	**0.554**	0.408	0.501	0.512	0.500	0.521	0.542	0.522
Jedit3.2–4.0	0.629	0.631	0.716	0.673	**0.772**	0.763	0.612	0.637	0.623
Jedit4.0–4.1	0.629	0.709	0.739	0.595	**0.783**	0.770	0.639	0.614	0.656
Jedit4.1–4.2	**0.641**	0.543	0.618	0.600	0.653	0.640	0.507	0.558	0.522
Jedit4.2–4.3	0.736	0.678	0.730	0.406	**0.759**	0.727	0.672	0.625	0.699
Log4j1.0–1.1	**0.806**	0.745	0.729	0.727	0.679	0.721	0.481	0.723	0.550
Log4j1.1–1.2	0.589	0.759	0.386	0.321	0.392	0.410	**0.913**	0.670	0.910
Lucene2.0–2.2	**0.735**	0.707	0.560	0.472	0.610	0.576	0.677	0.577	0.683
Lucene2.2–2.4	0.715	0.745	0.676	0.526	0.653	0.648	0.789	0.663	**0.797**
Poi1.5–2.0	0.654	0.592	0.503	0.382	0.515	0.493	**0.669**	0.657	0.643
Poi2.0–2.5	0.739	0.688	0.379	0.215	0.326	0.342	**0.837**	0.526	0.811
Poi2.5–3.0	0.729	0.673	0.666	0.389	0.648	0.656	0.750	0.638	**0.752**
Synapse1.0–1.1	0.543	0.565	0.567	0.568	0.521	**0.580**	0.542	0.550	0.559
Synapse1.1–1.2	0.651	**0.664**	0.537	0.445	0.579	0.577	0.593	0.605	0.608
Velocity1.4–1.5	0.872	0.853	0.884	0.877	0.885	0.884	**0.886**	0.408	0.882
Velocity1.5–1.6	0.763	0.689	0.736	0.398	0.730	0.694	0.784	0.450	**0.786**
Xalan2.4–2.5	0.811	0.759	0.683	0.266	0.545	0.579	0.832	0.726	**0.836**
Xalan2.5–2.6	0.762	0.767	0.574	0.675	0.662	0.646	0.793	0.666	**0.795**
Xalan2.6–2.7	0.934	0.921	0.392	0.257	0.491	0.483	**0.970**	0.762	0.969

(*continued on next page*)

Table 4 (*continued*).

Cross-version	EALTR	DEJIT	CBS+	EASC	CBS+(RF)	CBS+(DF)	ManualUp	EALR	EATT
Xerces init-1.2	0.903	**0.916**	0.832	0.415	0.830	0.802	0.850	0.833	0.854
Xerces1.2–1.3	0.600	0.569	**0.738**	0.546	0.650	0.661	0.646	0.715	0.649
Xerces1.3–1.4	0.747	0.768	0.637	0.402	0.572	0.569	0.804	0.636	**0.809**
Average	0.679	0.664	0.613	0.493	0.617	0.617	0.682	0.603	**0.689**
W/D/L		17/0/13	20/0/10	26/0/4	20/0/10	21/0/9	13/0/17	21/0/9	12/0/18
p-value		0.115	0.049	0.000	0.051	0.051	0.688	0.006	0.463

Table 5
The Recall@20% value of the nine methods on each cross-version experiment when inspecting the top 20% LOC.

Cross-version	EALTR	DEJIT	CBS+	EASC	CBS+(RF)	CBS+(DF)	ManualUp	EALR	EATT
Ant1.3–1.4	0.244	0.079	0.100	0.200	0.175	0.188	**0.525**	0.050	0.476
Ant1.4–1.5	0.303	0.303	0.313	0.250	0.272	**0.323**	0.313	0.313	0.278
Ant1.5–1.6	0.229	0.236	0.185	**0.283**	0.140	0.158	0.261	0.120	0.257
Ant1.6–1.7	0.147	0.127	0.235	0.229	0.252	0.252	**0.277**	0.175	0.276
Camel1.0–1.2	0.242	0.137	0.366	0.148	0.236	0.236	**0.602**	0.194	0.485
Camel1.2–1.4	0.440	0.431	0.200	0.228	**0.521**	0.493	0.462	0.331	0.448
Camel1.4–1.6	0.501	0.358	0.303	0.176	0.252	0.274	**0.537**	0.298	0.505
Ivy1.1–1.4	**0.366**	0.284	0.250	0.250	0.328	0.338	0.313	0.250	0.344
Ivy1.4–2.0	0.263	0.280	0.125	0.200	0.226	0.211	0.250	**0.300**	0.260
Jedit3.2–4.0	0.325	**0.488**	0.347	0.373	0.457	0.455	0.427	0.347	0.485
Jedit4.0–4.1	0.188	0.308	0.329	0.342	0.398	0.365	0.380	0.266	**0.441**
Jedit4.1–4.2	0.358	0.392	0.333	0.354	**0.398**	0.382	0.271	0.313	0.348
Jedit4.2–4.3	0.432	0.255	0.273	0.182	0.341	0.368	0.455	0.182	**0.477**
Log4j1.0–1.1	0.268	0.247	0.324	**0.351**	0.331	0.347	0.243	0.324	0.338
Log4j1.1–1.2	0.119	0.399	0.148	0.175	0.157	0.147	**0.614**	0.354	0.533
Lucene2.0–2.2	0.292	0.478	0.319	0.257	0.344	0.293	**0.597**	0.319	0.562
Lucene2.2–2.4	0.265	0.468	0.443	0.261	0.444	0.451	**0.616**	0.429	0.595
Poi1.5–2.0	0.381	0.284	0.243	0.216	0.269	0.276	**0.459**	0.378	0.316
Poi2.0–2.5	0.326	0.270	0.040	0.073	0.104	0.084	**0.569**	0.137	0.405
Poi2.5–3.0	0.538	0.466	0.509	0.181	0.459	0.456	**0.544**	0.431	0.528
Synapse1.0–1.1	0.165	0.219	0.150	0.300	0.183	0.256	0.383	0.183	**0.404**
Synapse1.1–1.2	0.263	**0.355**	0.198	0.209	0.198	0.187	0.349	0.314	0.324
Velocity1.4–1.5	0.608	0.643	0.683	0.683	0.688	**0.689**	0.683	0.275	0.664
Velocity1.5–1.6	0.567	0.592	0.551	0.308	0.547	0.533	**0.615**	0.321	0.599
Xalan2.4–2.5	0.595	0.495	0.106	0.129	0.117	0.124	**0.628**	0.470	0.610
Xalan2.5–2.6	0.481	0.506	0.297	0.241	0.436	0.425	**0.545**	0.365	0.538
Xalan2.6–2.7	0.678	0.655	0.228	0.160	0.312	0.304	**0.714**	0.410	0.695
Xerces init-1.2	0.713	0.716	0.634	0.141	0.653	0.603	0.761	0.577	**0.802**
Xerces1.2–1.3	0.629	0.559	0.072	0.203	0.156	0.338	0.652	0.217	**0.654**
Xerces1.3–1.4	0.633	0.617	0.092	0.121	0.096	0.087	**0.762**	0.382	0.760
Average	0.385	0.388	0.280	0.241	0.316	0.321	**0.494**	0.301	0.480
W/D/L		15/1/14	20/0/10	22/0/8	19/0/11	18/0/12	4/0/26	19/0/11	6/0/24
p-value		0.888	0.020	0.009	0.181	0.162	0.001	0.016	0.000

Table 6
The average Precision@20% and PMI@20% values of the nine methods.

Metrics	EALTR	DEJIT	CBS+	EASC	CBS+(RF)	CBS+(DF)	ManualUp	EALR	EATT
Precision@20%	0.395	0.378	0.502	**0.507**	0.480	0.476	0.292	0.326	0.299
PMI@20%	0.428	0.459	0.245	**0.170**	0.246	0.263	0.687	0.350	0.643

indicate there is a significant difference between EALTR and CBS+, EASC, and EALR.

(5) As shown in Figs. 4 and 5 and Table 6, EASC achieves the highest median **Precision@20%** value and the lowest median **PMI@20%** value. CBS+, EASC, CBS+(RF), and CBS+(DF) significantly perform better than EALTR in term of Precision@20%, while EALTR significantly outperforms ManualUp, EALR, and EATT. In addition, EALTR requires software testers to inspect significantly more modules than CBS+, EASC, CBS+(RF), CBS+(DF), and EALR. But the Precision@20% and PMI@20% values of EALTR are acceptable.

(6) ManualUp and EATT perform the worst in terms of IFA, Precision@20%, and PMI@20%. ManualUp sorts the modules in the ascending order of LOC and EATT sorts the modules in the descending order of the ratio between the predicted defective probability and LOC. In other words, they are likely to put the modules with fewer LOC at the top of the ranking. Therefore, software testers need to test more modules (higher PMI@20% value) when inspecting the top 20% LOC. In addition, the modules with fewer LOC are likely to be non-defective, so the IFA values of ManualUp and EATT are high, and the Precision@20% values are low.

(7) Both DEJIT and EALTR employ the differential evolution algorithm to construct linear models. The fundamental distinction between the two lies in the optimization metrics utilized when searching for optimal model parameters. The EALTR model concentrates on identifying a greater number of bugs within constrained resources and introduces the PofB@20% metric. Conversely, DEJIT proposes the DPA metric, which is based on the concept of *F*ault-*P*ercentile-*A*verage (FPA) [29] as a global ranking criterion. In this experiment, upon examining their experimental results across six metrics and 30 cross-version pairs individually, with the exception of nearly identical performance in Recall@20%, EALTR demonstrates more expressive average results than DEJIT across all other metrics, particularly outperforming in 18

and 17 cross-version pairs on PofB@20% and *Popt* respectively. Thus, it can be stated that in practical defect detection scenarios, we advocate for utilizing the EALTR method optimized for the PofB@20% metric.

(8) CBS+, EASC, CBS+(RF), and CBS+(DF) firstly use the classification algorithms to predict the defective possibility of modules. Then, the predicted defective modules are inspected firstly. Since the modules with more LOC are likely to be predicted as defective ones, there are fewer inspected modules when testing the top 20% LOC based on the ranking results of the four methods. Therefore, although CBS+, EASC, CBS+(RF), and CBS+(DF) achieve a high Precision@20% value and low IFA value, the Recall@20% and PofB@20% values are low. In addition, CBS+(RF) and CBS+(DF) perform better than CBS+ and EASC in terms of PofB@20%, *Popt*, and Recall@20%, which shows that ensemble algorithms are useful for constructing an EADP model.

(9) EALTR significantly performs better than EALR in terms of IFA, PofB@20%, *Popt*, and Recall@20%. Therefore, directly optimizing the PofB@20% value for the linear regression model is useful for building better EADP models than using the least square method.

Answer to RQ1

Except for ManualUp and EATT whose IFA values are greater than 10, EALTR improves the PofB@20% value by 5.83%–54.47% and the *Popt* value by 2.26%–37.73%.

4.2. RQ2: Can the re-ranking strategy improve the performance of EALTR in terms of IFA and Precision@20%?

Motivations: The experiment results in RQ1 show that the average IFA value of EALTR is 3.723, which is worse than that of CBS+. Software testers would be frustrated and unlikely to keep up the inspection of the predicted defective modules, if they could not obtain satisfactory result within the first few inspected modules. Therefore, we propose a re-ranking strategy to reduce the IFA value, and call the EALTR method integrated with the re-ranking strategy EALTR*

Methods: We analyze the performance of EALTR* in terms of the six evaluation metrics. Table 7 presents the Recall@20%, PofB@20%, Precision@20%, PMI@20%, IFA, and *Popt* values on each cross-version pair. The row W/D/L indicates the number of cross-version pairs, on which EALTR* obtains a better, equal, or worse performance than EALTR. The p-values indicate there is a significant difference between EALTR and EALTR*.

Results: EALTR* significantly performs better than EATLR in terms of IFA, and reduces the average IFA value of EALTR by 16.95% (from 3.723 to 3.092). The row W/D/L shows that EALTR* wins and draws EALTR on 14 and 16 datasets, respectively. In addition, the average Precision@20% value increases from 0.395 to 0.400 and the average PMI@20% value reduces from 0.428 to 0.417. However, it is inevitable to bring some sacrifice of PofB@20% (from 0.363 to 0.361), Recall@20% (from 0.385 to 0.382), and *Popt* (from 0.679 to 0.676). In summary, the re-ranking strategy can reduce the IFA value to a large degree with a little sacrifice of the found bugs. Therefore, we recommend to use the re-ranking strategy, if the predicted defective modules are assigned to a few software testers and the negative impacts of initial false alarms on software testers' confidence on the EADP method should be seriously considered.

Answer to RQ2

The re-ranking strategy can reduce the average IFA value by 16.95% and increase the average Precision@20%.

4.3. RQ3: What is the execution time for EALTR?

Motivations: EALTR utilizes the genetic algorithm to obtain the optimal parameter of the linear regression model, which is a time-consuming task. Hence, we explore the time efficiency of EALTR.

Methods: The experimental environment is a Windows 10 64-bit personal computer with 8 GB RAM. Table 8 presents the average training time and testing time of EALTR and the compared methods over the 30 cross-version pairs.

Results: Since EALTR* integrates the re-ranking strategy in the prediction phase, the training times of EALTR and EALTR* are the same and the testing time of EALTR* is longer than that of EALTR. As for DEJIT, since it optimizes the DPA metric which has a higher computational complexity than PofB@20%, its training time is longer and equivalent to 1.36 times that of EALTR. This is also one of the relative drawbacks of using the DEJIT method. Since CBS+(DF) employs the deep ensemble learning algorithm (i.e., DF) as the underlying classifier of CBS+, its training time is longer than those of CBS+, EASC, and CBS+(RF). ManualUp is the unsupervised method, so its training time is 0 s. EATT builds three classification models using a tri-training strategy, so its training time is longer than those of CBS+ and EASC. The training and testing times of EALTR are reasonable. We need an average 119.2333 s to train the EADP model, and 0.0138 s to rank software modules using the model. Notice that the model does not need to be updated all the time. A trained model can be used to rank many software modules. Therefore, the efficiency of EALTR is applicable in practice.

Answer to RQ3

The training time and testing time of EALTR are reasonable in practice.

5. Discussion

5.1. The impact of the percentage of inspected LOC

By default, we set the percentage of the inspected LOC as 20%. When testing resources are fairly limited, software testers might assign the testing resources to only the top 10% LOC. So we investigate the performance of EALTR and the compared methods when testing the top 10% LOC. Table 9 presents the average PofB@10%, Recall@10%, *Popt*, Precision@10%, IFA, and PMI@10% values of the ten methods across 30 cross-version pairs and Fig. 6 shows the distribution of their values of the methods across all cross-version pairs.

Both EATLR and EALTR* outperform DEJIT on all metrics except for Recall@10%, and still perform better than CBS+, EASC, CBS+(RF), CBS+(DF), and EALR in terms of PofB@20%, Recall@20%, and *Popt* when inspecting the top 10% LOC. Specifically, EALTR achieves a 2.25% improvement in *Popt* and a 10.05% improvement in Precision@10% while reducing IFA by 8.91% when compared to DEJIT. For the remaining five methods, EALTR improves their average PofB@20% values by 28.14%, 58.52%, 11.46%, 12.04%, and 30.49%, respectively, their average Recall@20% values by 6.10%–51.68%, and their average *Popt* values by 9.69%–37.73%. The adjusted p-values show that there is a significant difference between EALTR and EALTR*, CBS+, EASC, and EALR in terms of PofB@10%. ManualUp and EATT are significantly performs better than EALTR in terms of PofB@10% and Recall@20%, but their IFA values are larger than 10 and they significantly perform worse than EATLR and EALTR* in terms of Precision@10% and PMI@10%. In summary, EALTR and EALTR* still can find more bugs when inspecting the top 10% LOC.

Table 7
The performance of EALTR* on each cross-version experiment.

Cross-version	Recall@20%	PofB@20%	Precision@20%	PMI@20%	IFA	*Popt*
Ant1.3–1.4	0.223	0.248	0.233	0.229	7.75	0.602
Ant1.4–1.5	0.306	0.283	0.070	0.496	6.250	0.587
Ant1.5–1.6	0.229	0.230	0.432	0.174	0.30	0.565
Ant1.6–1.7	0.147	0.242	0.569	0.068	0.60	0.563
Camel1.0–1.2	0.234	0.245	0.466	0.199	0	0.531
Camel1.2–1.4	0.440	0.329	0.127	0.578	0.40	0.564
Camel1.4–1.6	0.501	0.383	0.181	0.547	0	0.630
Ivy1.1–1.4	0.366	0.325	0.036	0.677	15.30	0.583
Ivy1.4–2.0	0.258	0.225	0.092	0.432	12.00	0.514
Jedit3.2–4.0	0.325	0.340	0.485	0.228	0.30	0.629
Jedit4.0–4.1	0.188	0.331	0.615	0.146	0.05	0.629
Jedit4.1–4.2	0.358	0.338	0.189	0.284	0.45	0.640
Jedit4.2–4.3	0.432	0.471	0.047	0.240	1.25	0.736
Log4j1.0–1.1	0.268	0.420	0.663	0.143	0	0.806
Log4j1.1–1.2	0.119	0.143	0.989	0.112	0	0.589
Lucene2.0–2.2	0.290	0.354	0.659	0.263	0.05	0.733
Lucene2.2–2.4	0.265	0.304	0.535	0.300	0	0.715
Poi1.5–2.0	0.381	0.377	0.090	0.510	8.75	0.654
Poi2.0–2.5	0.327	0.282	0.517	0.401	0	0.740
Poi2.5–3.0	0.538	0.404	0.579	0.590	3.40	0.729
Synapse1.0–1.1	0.158	0.243	0.478	0.093	0.65	0.536
Synapse1.1–1.2	0.258	0.303	0.295	0.313	0.15	0.646
Velocity1.4–1.5	0.608	0.512	0.620	0.654	0.05	0.872
Velocity1.5–1.6	0.567	0.467	0.284	0.681	6.40	0.763
Xalan2.4–2.5	0.575	0.499	0.432	0.641	0.85	0.791
Xalan2.5–2.6	0.480	0.426	0.330	0.676	6.85	0.761
Xalan2.6–2.7	0.672	0.579	0.982	0.675	0	0.926
Xerces init-1.2	0.706	0.685	0.153	0.744	8	0.893
Xerces1.2–1.3	0.623	0.433	0.124	0.765	12.85	0.597
Xerces1.3–1.4	0.626	0.405	0.717	0.649	0.10	0.741
Average	0.382	0.361	0.400	0.417	3.092	0.676
W/D/L	2/15/13	3/14/13	19/10/1	20/10/0	14/16/0	6/10/14
p-value	0.004	0.006	0.000	0.000	0.001	0.003

Table 8
The average training time and testing time of the ten methods.

Method	Training Time	Testing Time
EALTR	119.2333 s	0.0138 s
EALTR*	119.2333 s	0.0176 s
DEJIT	162.1573 s	0.0137 s
CBS+	0.0207 s	0.0007 s
EASC	0.0007 s	0.0008 s
CBS+(RF)	0.1233 s	0.0117 s
CBS+(DF)	1.7409 s	0.0636 s
ManualUp	0 s	0.0085 s
EALR	0.0037 s	0.0008 s
EATT	0.1054 s	0.0169 s

5.2. The impact of the number of re-ranking modules and LOC threshold

If the LOC of the top-ranked *q* modules are less than the LOC (as the threshold) of the module located at a proportion *p* in the non-defective modules (sorted in ascending order by their LOC size) in the training dataset, the modules are considered to be non-defective. Then, we place the modules at the end of the ranking. In this subsection, we investigate the effect of the number of re-ranking modules (*q*) and the LOC threshold. We change the values of *q* from 0 to 40 with an interval of 5 and *p* from 1/8 to 7/8 with an interval of 1/8, and record the average performance of EALTR* across the 30 cross-version pairs in Fig. 7.

The average PofB@20%, Recall@20%, *Popt*, and PMI@20% values decrease, when we increase the values of *q* or *p*. The Precision@20% value increases along with the values of *q* or *p*. The detailed reasons are as follows. Since the re-ranking strategy places the modules with fewer LOC at the end of the ranking, the top-ranked modules tend to contain more LOC. Therefore, the number of required inspected modules (PMI@20%) decreases when inspecting the top 20% LOC. Since software testers inspect fewer modules, the numbers of the found defective modules (Recall@20%) and bugs (PofB@20%) decrease accordingly. The IFA value exhibits a downward trend as the value of *p* decreases or the value of *q* increases. As shown in Fig. 7(e), the minimum value of IFA is achieved when using the combination of $q = 40$ and $p = 1/8$. The values of all performance metrics at this combination are also specifically indicated. The results show that at the lowest point of the IFA value, only a small sacrifice is made for other metrics. Therefore, we suggest to use the combination of $q = 40$ and $p = 1/8$ as the default setting to achieve the lowest IFA value.

5.3. The performance under cross-project setup

Cross-project defect prediction [1,6,60–64] is a valuable and necessary research area, primarily due to the difficulty in obtaining within-project data. Therefore, we conduct cross-project defect prediction to validate the efficacy of our approach in this domain. To achieve this, we utilize eleven software projects' initial version datasets as our experimental datasets. Due to the difficulty of selecting a training set to create a superior model, we opt to use one dataset for within-project testing and the remaining ten for cross-project training at each iteration. We run the procedure 20 times on each training and testing pair and calculate the average value to avoid sample biases. Our cross-project experimental setup is consistent with that of Huang et al.'s study [27]. Table 10 presents the average of PofB@20%, Recall@20%, *Popt*, Precision@20%, IFA, and PMI@20% values achieved by the ten methods across eleven cross-project pairs and Fig. 8 illustrates the distribution of these metrics.

Table 9
The average PofB@10%, Recall@10%, *Popt*, Precision@10%, IFA, and PMI@10% values of the ten methods on cross-version experiment. (* indicates there is significant difference between EALTR and the corresponding method, while $^{+}$ indicates there is significant difference between EALTR* and the corresponding method.).

Metrics	EALTR	EALTR*	DEJIT	CBS+	EASC	CBS+(RF)	CBS+(DF)	ManualUp	EALR	EATT
PofB@10%	0.214^{+}	0.212*	0.212	0.167^{*+}	0.135^{*+}	0.192	0.191	0.269^{*+}	0.164^{*+}	0.259^{*+}
Recall@10%	0.226^{+}	0.223*	0.240	0.181	0.149	0.213	0.209	0.353^{*+}	0.171	0.337^{*+}
Popt	0.679^{+}	0.676*	0.664	0.613*	0.493^{*+}	0.617	0.617	0.682	0.603^{*+}	0.689
Precision@10%	0.427^{+}	0.433*	0.388	0.481	0.469	0.467	0.463	0.271^{*+}	0.291^{*+}	0.274^{*+}
IFA	3.723^{+}	3.092*	4.087	3.567	6.1	4.458	4.393	16.533^{*+}	8.033	17.313^{*+}
PMI@10%	0.270^{+}	0.261*	0.311	0.169	0.120^{*+}	0.176*	0.182	0.542^{*+}	0.222	0.501^{*+}

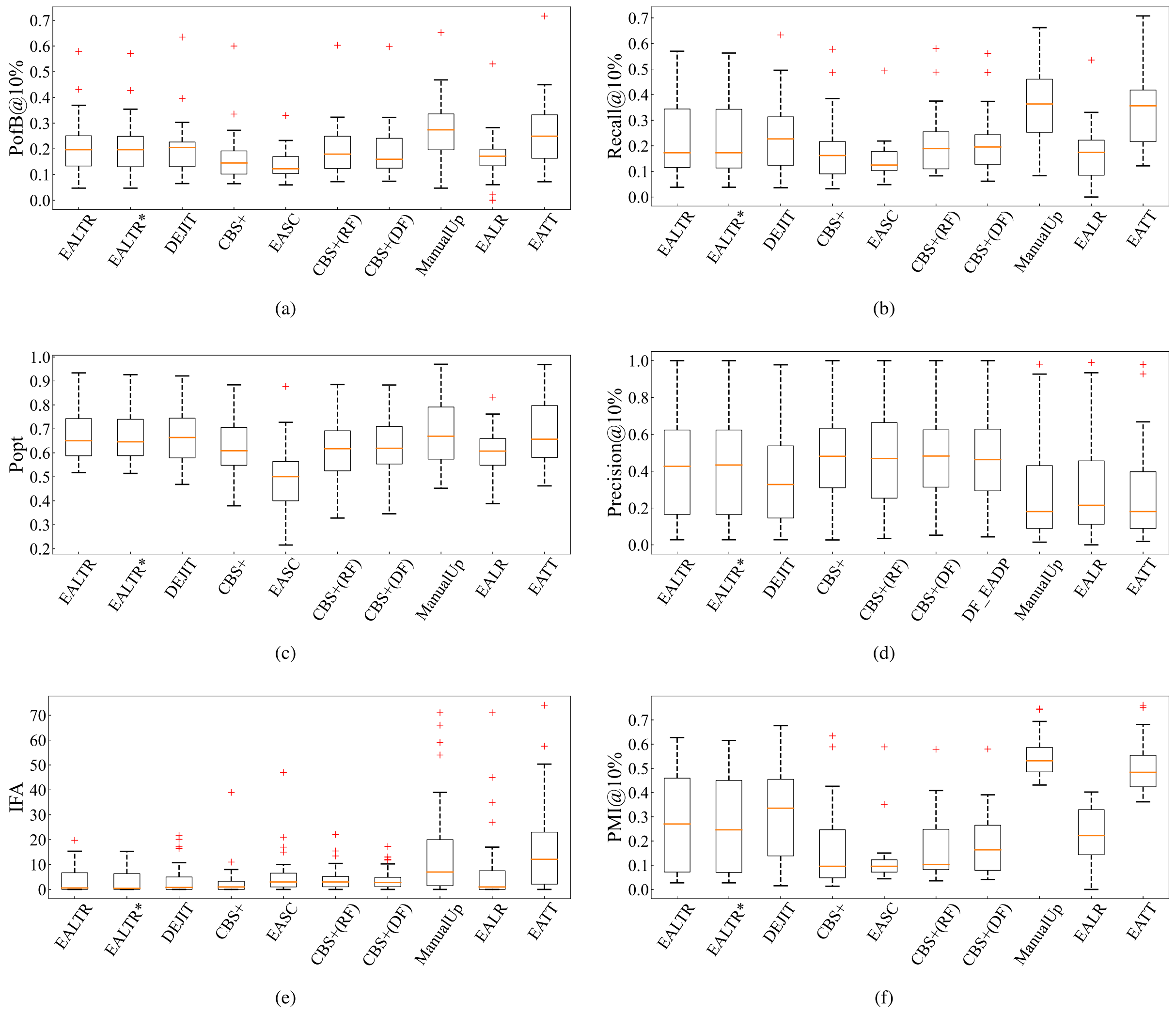

Fig. 6. The boxplot of the PofB@10%, Recall@10%, *Popt*, Precision@10%, IFA, and PMI@10% values of the ten methods on cross-version experiment.

Due to the limited number of cross-project pairs, no training dataset satisfies the conditions for the re-ranking strategy. As a result, EALTR and EALTR* exhibit the same results. There are significant differences between DEJIT, ManulUp, and EATT compared with EALTR in terms of the IFA, with values of 11.282, 10.909, and 8.409 respectively being excessively high and thus unacceptable. EALTR still outperforms CBS+, EASC, CBS+(RF), CBS+(DF), and EALR on PofB@20%, Recall@20%, and *Popt* with improvements of 25.36%, 42.39%, 89.07%, 110.98%, and 74.75% on PofB@20%, 63.00%–174.07% on Recall@20%, and 12.44%–34.79% on *Popt*, respectively. The adjusted *p*-value shows that there are significant differences between EALTR and CBS+(RF) and CBS+(DF) methods in terms of PofB@20% and Recall@20%. In summary, the EALTR and EALTR* methods still outperform baseline methods under the cross-project scenario.

5.4. Threats to validity

(1) We select 41 versions of 11 software projects from the PROMISE repository for the experiment, since we use the more practical cross-version setup for EADP. The projects come from different application domains, have a different number of modules and defective ratios, and are widely used in previous EADP studies [28], which contributes to generalizing the experimental results to some degree. But we still cannot guarantee that EALTR and EALTR* can perform the best for other software projects. However, we provide a detailed description of our method, which will help future researchers to replicate our experiment in their defect projects.

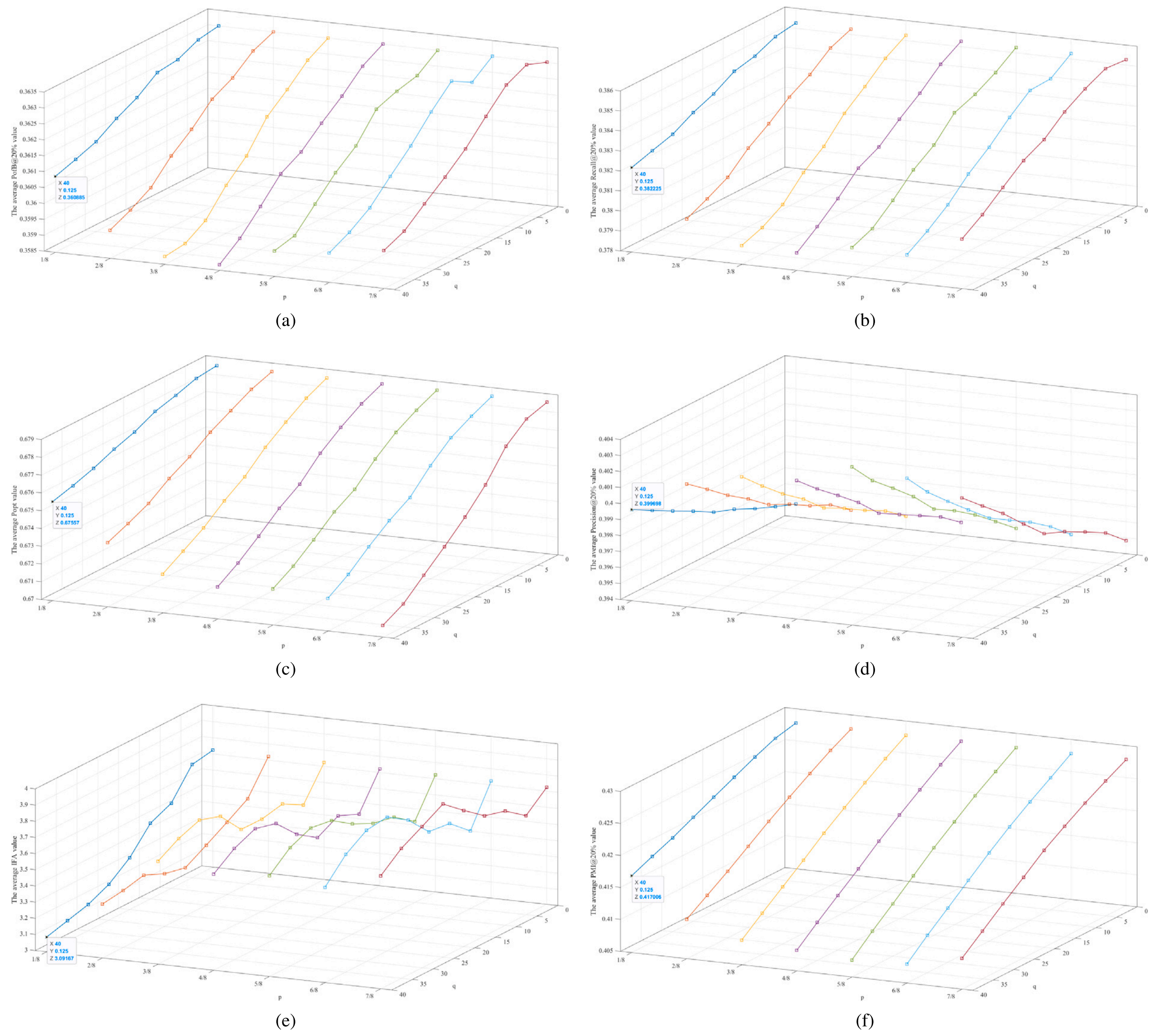

Fig. 7. The average performance of EALTR* across the 30 cross-version pairs with different *p* and *q* values. (We have labeled the values for $q = 40$, $p = 1/8$.)

Table 10
The average PofB@20%, Recall@20%, *Popt*, Precision@20%, IFA, and PMI@20% values of the ten methods on cross-project experiment. (* indicates there is significant difference between EALTR and the corresponding method, while $^+$ indicates there is significant difference between EALTR* and the corresponding method.).

Metrics	EALTR	EALTR*	DEJIT	CBS+	EASC	CBS+(RF)	CBS+(DF)	ManualUp	EALR	EATT
PofB@20%	0.346	0.346	0.344	0.276	0.243	0.183^{*+}	0.164^{*+}	0.324	0.198	0.363
Recall@20%	0.370	0.370	0.440^{*+}	0.227	0.204	0.190^{*+}	0.165^{*+}	0.430	0.135^{*+}	0.458^{*+}
Popt	0.678	0.678	0.635	0.603	0.503	0.533	0.504^{*+}	0.631	0.524	0.646
Precision@20%	0.304	0.304	0.284	0.533^{*+}	0.555^{*+}	0.490^{*+}	0.500^{*+}	0.271^{*+}	0.546^{*+}	0.294
IFA	4.205	4.205	11.282^{*+}	1.818	3.545	2.718	5.341	10.909^{*+}	3.182	8.409^{*+}
PMI@20%	0.462	0.462	0.602^{*+}	0.156^{*+}	0.099^{*+}	0.128^{*+}	0.126^{*+}	0.658^{*+}	0.087^{*+}	0.609^{*+}

(2) We utilize a range of effort-aware metrics to evaluate the efficacy of the models [27], namely Recall@20%, PofB@20%, Precision@20%, PMI@20%, *Popt*, and IFA. Since EADP model aims to identify more defects and defective modules and establish an accurate global ranking based on predicted defect density, we employ PofB@20%, Recall@20%, and *Popt*. The inclusion of Precision@20% is necessary, as it is commonly paired with Recall@20%. Moreover, we employ PMI@20% to mitigate the additional effort costs incurred by excessive module checking. To ensure that software testers' confidence is not unduly undermined, we also take into consideration the IFA value, which has been shown in previous studies [27] to be a critical determinant.

(3) The randomness of the genetic algorithm could affect the experimental results of EALTR, so we repeat the cross-version experiments 20 times and obtain the average value of the 20 results as the final result on each cross-version pair. In addition, the Wilcoxon signed-rank test with the BH correctness is also used to ascertain the practical performance significance between difference EADP methods.

(4) Although existing SDP studies have pointed out that some feature selection methods could improve the performance of classification-based SDP models, little research has been designed to investigate the

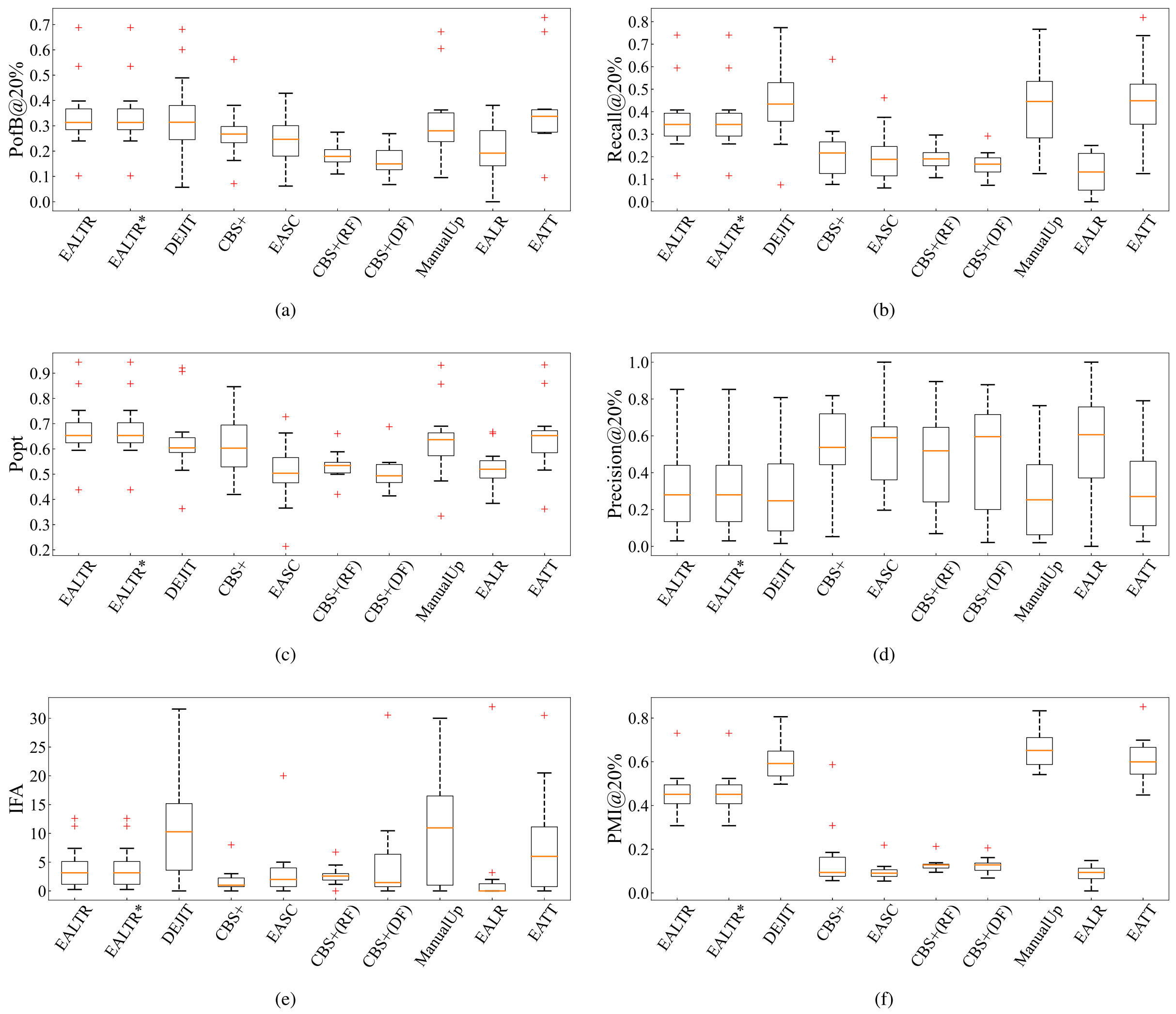

Fig. 8. The boxplot of the PofB@20%, Recall@20%, Popt, Precision@20%, IFA, and PMI@20% values of the ten methods on cross-project experiment.

impact of feature selection methods on EADP models. We are less aware of the effectiveness of feature selection methods for EADP models. So we do not apply the feature selection methods to the software datasets like Ni et al.'s study [28]. In addition, we do not employ imbalanced learning techniques [10,11,41,65–68] to address the data imbalance problem, since the related imbalanced learning studies for EADP are limited.

(5) We set the parameters of the baseline methods according to the their works or using the default parameters. For example, we regard the modules whose predicted defective probability is larger than or equal to 0.5 as the defective ones for CBS+, EASC, CBS+(RF), and CBS+(DF). We set the same classification threshold as Huang et al. [27] and Ni et al. [28], and it is also a common practice. The best parameters for different software projects may be different, which may lead to somewhat different results.

(6) We utilize the linear regression model to construct the EADP model, which might not be the best alternative. In the future, we will try to employ some nonlinear models to construct the EADP model and compare the performance differences.

6. Related work

6.1. EADP

Mende et al. [23] for the first time put forward the concept of "effort-aware" and proposed two ranking strategies. Kamei et al. [69] further investigated linear models, regression trees, random forests, and other classification algorithms for EADP. Subsequently, Kamei et al. [26] proposed the EALR method for effort-aware Just-In-Time (JIT) defect prediction, which used the linear regression algorithm to predict the defect density of software code changes. Bennin et al. [70, 71] and Yu et al. [21] investigated which algorithm was the best-performing one for EADP and assessed the impact of data re-sampling methods for EADP. Yang et al. [37] found that the unsupervised ManualUp method performed better than some simple supervised methods for effort-aware JIT defect prediction. Yan et al. [72] also applied ManualUp to file-level EADP and compared it with some supervised algorithms. Huang et al. [27,73] reviewed Kamei et al.'s [26] and Yang et al.'s [37] works and pointed out the shortcomings of ManualUp. ManualUp leads to more required inspected modules than supervised methods, and the IFA value is high. Therefore, they proposed the CBS+ algorithm, which required testers to test fewer code changes and could significantly reduce the IFA value. Ni et al. [28] proposed a file-level cross-project EADP method called EASC, which uses naive Bayes as the underlying classifier. Subsequently, Ni et al. [74] verified the effectiveness of CBS+ on JavaScript projects. Qiao et al. [75] proposed to utilize neural networks and deep learning methods to construct effort-aware JIT software defect prediction models. Li et al. [19] proposed the semi-supervised method named Effort-Aware Tri-Training (EATT), which used a greedy strategy to sort code changes. Qu et al. [76] and Du et al. [77] proposed to utilize *k*-core decomposition on software class dependency networks to improve EADP in software systems. Subsequently, Qu et al. [78] employed the developer information to improve the performance of EADP models. Xu et al. [44], Zhao et al. [53], and Cheng et al. [54] proposed the effort-aware JIT defect prediction methods for android apps. «arka et al. [24] conducted an empirical study on the trends and effectiveness of effort-aware metrics in EADP. Li et al. [12] investigated the impact of feature selection techniques on the performance of CBS+.

However, the existing EADP studies mainly consider the EADP task as a regression or classification problem and aim to optimize the regression loss or classification accuracy. But, the low regression loss or high classification accuracy might result in poor effort-aware performance. Therefore, we propose the EALTR method by directly optimizing the PofB@20% value. Recently, Yang et al. [31] also proposed a EADP model called DEJIT by directly optimizing the DPA metric, which is a metric used to measure global ranking performance. However, the primary objective of EADP is to detect more bugs within the top 20% of LOC, thus directly optimizing PofB@20% is more effective than DPA. Additionally, in contrast to the work by Yang et al. [31], we also propose a re-ranking strategy to reduce IFA and increase Precision@20% values.

6.2. Genetic algorithms for SDP

Recently, some researchers have proposed using genetic algorithms to train SDP models. For example, Shuai et al. [79] proposed a dynamic support vector machine method for SDP, and used the genetic algorithm to maximize the geometric classification accuracy. Arun et al. [80] proposed an oversampling method to generate synthetic defective modules using the genetic algorithm for SDP. Wahono et al. [81,82] applied genetic algorithm to select representative features for SDP. Ryu et al. proposed [83] a multi-objective naive Bayes algorithm for cross-project SDP using the multi-objective genetic algorithm. Hosseini et al. [84] proposed to utilize the Nearest Neighbor (NN) filter embedded in the genetic algorithm to select the valuable cross-project modules. Chen et al. [55] proposed a multi-objective optimization-based EADP method call MULTI, which aimed to maximizing the number of found defects and minimizing the inspected LOC. Since existing EADP studies generally fix the inspected percentage of LOC as 20%, we optimize only one objective (i.e., PofB@20%). Yang et al. [29] proposed an SDP method named LTR to rank software modules based on the bug numbers and aimed to find more bugs when inspecting a certain number of modules. LTR used the genetic algorithm to directly optimize the ranking performance (i.e., FPA) to construct the model. Inspired by their work [29], we propose the EALTR method to build the EADP model by directly maximizing the PofB@20% value.

7. Conclusion

This study proposes an EADP method called EALTR, that learns to rank software modules by directly optimizing the effort-aware metric (i.e., PofB@20%). The method employs the linear regression model to build the relationship between the defect density and software features. Different from Kamei et al.'s work [26], we use the composite differential evolution algorithm to solve the EADP model. The coefficient vector of the linear regression that obtains the best PofB@20% value on the training dataset is used to build the EADP model. Then, we propose a re-ranking strategy to increase the Precision@20% value and reduce the IFA value of EALTR. We evaluate the proposed EALTR method on 30 cross-version pairs and compare it with 8 baseline EADP methods. The results show that EALTR can find more defective modules and bugs with low IFA value, and the re-ranking strategy can reduce the false alarms.

CRediT authorship contribution statement

Xiao Yu: Data curation, Formal analysis, Methodology, Writing – original draft. **Jiqing Rao:** Methodology, Software, Writing – original draft. **Lei Liu:** Data curation, Methodology, Software, Writing – original draft. **Guancheng Lin:** Formal analysis, Software, Resources. **Wenhua Hu:** Software, Visualization, Writing – review & editing. **Jacky Wai Keung:** Project administration, Software, Validation. **Junwei Zhou:** Investigation, Project administration, Visualization. **Jianwen Xiang:** Methodology, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is partially supported by the National Key Research and Development Program (Grant No. 2022YFB3104001), the National Natural Science Foundation of China (Grant No. 61806151, 62202350), the Key Research and Development Program of Hubei Province (Grant No. 2022BAA050, 2020AAA001), the Key Research and Development Program of Hainan Province (Grant No. ZDYF2021GXJS014), and the Natural Science Foundation of Chongqing (Grant No. cstc2021jcyj-msxmX1146, cstc2021jcyj-msxmX1115).

References

[1] J. Bai, J. Jia, L.F. Capretz, A three-stage transfer learning framework for multi-source cross-project software defect prediction, Inf. Softw. Technol. 150 (2022) 106985.
[2] C. Pornprasit, C. Tantithamthavorn, DeepLineDP: Towards a deep learning approach for line-level defect prediction, IEEE Trans. Softw. Eng. (2022).
[3] C. Zhou, P. He, C. Zeng, J. Ma, Software defect prediction with semantic and structural information of codes based on Graph Neural Networks, Inf. Softw. Technol. (2022) 107057.
[4] L. Gong, G.K.K. Rajbahadur, A.E. Hassan, S. Jiang, Revisiting the impact of dependency network metrics on software defect prediction, IEEE Trans. Softw. Eng. (2021).
[5] X. Yu, J. Keung, Y. Xiao, S. Feng, F. Li, H. Dai, Predicting the precise number of software defects: Are we there yet? Inf. Softw. Technol. 146 (2022) 106847.
[6] L. Gong, S. Jiang, L. Jiang, An improved transfer adaptive boosting approach for mixed-project defect prediction, J. Softw. 31 (2019) e2172.
[7] S. Stradowski, L. Madeyski, Machine learning in software defect prediction: A business-driven systematic mapping study, Inf. Softw. Technol. (2022) 107128.
[8] L. Gong, H. Zhang, J. Zhang, M. Wei, Z. Huang, A comprehensive investigation of the impact of class overlap on software defect prediction, IEEE Trans. Softw. Eng. (2022).
[9] N. Zhang, S. Ying, K. Zhu, D. Zhu, Software defect prediction based on stacked sparse denoising autoencoders and enhanced extreme learning machine, IET Softw. 16 (2022) 29–47.
[10] S. Feng, J. Keung, X. Yu, Y. Xiao, M. Zhang, Investigation on the stability of SMOTE-based oversampling techniques in software defect prediction, Inf. Softw. Technol. 139 (2021) 106662.
[11] Y. Gao, Y. Zhu, Y. Zhao, Dealing with imbalanced data for interpretable defect prediction, Inf. Softw. Technol. 151 (2022) 107016.
[12] F. Li, W. Lu, J.W. Keung, X. Yu, L. Gong, J. Li, The impact of feature selection techniques on effort-aware defect prediction: An empirical study, IET Softw. (2023).
[13] X. Yu, H. Dai, L. Li, X. Gu, J.W. Keung, K.E. Bennin, F. Li, J. Liu, Finding the best learning to rank algorithms for effort-aware defect prediction, Inf. Softw. Technol. (2023) 107165.
[14] Z. Zhang, Y. Lei, T. Su, M. Yan, X. Mao, Y. Yu, Influential global and local contexts guided trace representation for fault localization, ACM Transactions on Software Engineering and Methodology (2022) http://dx.doi.org/10.1145/3576043.
[15] D. Yang, Y. Lei, X. Mao, Y. Qi, X. Yi, Seeing the whole elephant: Systematically understanding and uncovering evaluation biases in automated program repair, ACM Trans. Softw. Eng. Methodol. (2022) http://dx.doi.org/10.1145/3561382.
[16] X. Yu, J. Liu, Z. Yang, X. Liu, The Bayesian network based program dependence graph and its application to fault localization, J. Syst. Softw. 134 (2017) 44–53.

[17] H. Xie, Y. Lei, M. Yan, Y. Yu, X. Xia, X. Mao, A universal data augmentation approach for fault localization, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 48–60.

[18] X. Yu, J. Liu, Z.J. Yang, X. Liu, X. Yin, S. Yi, Bayesian network based program dependence graph for fault localization, in: 2016 IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW, IEEE, 2016, pp. 181–188.

[19] W. Li, W. Zhang, X. Jia, Z. Huang, Effort-aware semi-supervised just-in-time defect prediction, Inf. Softw. Technol. 126 (2020) 106364.

[20] X. Chen, D. Zhang, Y. Zhao, Z. Cui, C. Ni, Software defect number prediction: unsupervised vs supervised methods, Inf. Softw. Technol. 106 (2019) 161–181.

[21] X. Yu, K.E. Bennin, J. Liu, J.W. Keung, X. Yin, Z. Xu, An empirical study of learning to rank techniques for effort-aware defect prediction, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2019, pp. 298–309.

[22] Y. Khatri, S.K. Singh, Towards building a pragmatic cross-project defect prediction model combining non-effort based and effort based performance measures for a balanced evaluation, Inf. Softw. Technol. (2022) 106980.

[23] T. Mende, R. Koschke, Effort-aware defect prediction models, in: 2010 14th European Conference on Software Maintenance and Reengineering, IEEE, 2010, pp. 107–116.

[24] J. Çarka, M. Esposito, D. Falessi, On effort-aware metrics for defect prediction, Empir. Softw. Eng. 27 (2022) 152.

[25] M. Ulan, W. Löwe, M. Ericsson, A. Wingkvist, Weighted software metrics aggregation and its application to defect prediction, Empir. Softw. Eng. 26 (2021) 1–34.

[26] Y. Kamei, E. Shihab, B. Adams, A.E. Hassan, A. Mockus, A. Sinha, N. Ubayashi, A large-scale empirical study of just-in-time quality assurance, IEEE Trans. Softw. Eng. 39 (2012) 757–773.

[27] Q. Huang, X. Xia, D. Lo, Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction, Empir. Softw. Eng. 24 (2019) 2823–2862.

[28] C. Ni, X. Xia, D. Lo, X. Chen, Q. Gu, Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction, IEEE Trans. Softw. Eng. (2020).

[29] X. Yang, K. Tang, X. Yao, A learning-to-rank approach to software defect prediction, IEEE Trans. Reliab. 64 (2014) 234–246.

[30] Y. Wang, Z. Cai, Q. Zhang, Differential evolution with composite trial vector generation strategies and control parameters, IEEE Trans. Evol. Comput. 15 (2011) 55–66.

[31] X. Yang, H. Yu, G. Fan, K. Yang, DEJIT: a differential evolution algorithm for effort-aware just-in-time software defect prediction, Int. J. Softw. Eng. Knowl. Eng. 31 (2021) 289–310.

[32] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, A. Bener, Defect prediction from static code features: current results, limitations, new approaches, Automated Softw. Eng. 17 (2010) 375–407.

[33] Z.-H. Zhou, J. Feng, Deep forest, 2017, arXiv preprint arXiv:1702.08835.

[34] T. Zhou, X. Sun, X. Xia, B. Li, X. Chen, Improving defect prediction with deep forest, Inf. Softw. Technol. 114 (2019) 204–216.

[35] C. Liu, D. Yang, X. Xia, M. Yan, X. Zhang, A two-phase transfer learning model for cross-project defect prediction, Inf. Softw. Technol. 107 (2019) 125–136.

[36] J. Rao, X. Yu, C. Zhang, J. Zhou, J. Xiang, Learning to rank software modules for effort-aware defect prediction, in: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C), IEEE, 2021, pp. 372–380.

[37] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, H. Leung, Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 157–168.

[38] E.J. Weyuker, T.J. Ostrand, R.M. Bell, Comparing the effectiveness of several modeling methods for fault prediction, Empir. Softw. Eng. 15 (2010) 277–295.

[39] K. Gao, T.M. Khoshgoftaar, A comprehensive empirical study of count models for software fault prediction, IEEE Trans. Reliab. 56 (2007) 223–236.

[40] G. Boetticher, The PROMISE repository of empirical software engineering data, 2007, http://promisedata.org/repository.

[41] X. Yu, J. Liu, J.W. Keung, Q. Li, K.E. Bennin, Z. Xu, J. Wang, X. Cui, Improving ranking-oriented defect prediction using a cost-sensitive ranking SVM, IEEE Trans. Reliab. 69 (2019) 139–153.

[42] T.J. Ostrand, E.J. Weyuker, R.M. Bell, Predicting the location and number of faults in large software systems, IEEE Trans. Softw. Eng. 31 (2005) 340–355.

[43] Z. Xu, S. Ye, T. Zhang, Z. Xia, S. Pang, Y. Wang, Y. Tang, MVSE: Effort-aware heterogeneous defect prediction via multiple-view spectral embedding, in: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security, QRS, IEEE, 2019, pp. 10–17.

[44] Z. Xu, K. Zhao, T. Zhang, C. Fu, M. Yan, Z. Xie, X. Zhang, G. Catolino, Effort-aware just-in-time bug prediction for mobile apps via cross-triplet deep feature embedding, IEEE Trans. Reliab. (2021).

[45] Z. He, F. Peters, T. Menzies, Y. Yang, Learning from open-source projects: An empirical study on defect prediction, in: 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, IEEE, 2013, pp. 45–54.

[46] Y. Chen, X. Lu, S. Wang, Deep cross-modal image–voice retrieval in remote sensing, IEEE Trans. Geosci. Remote Sens. 58 (2020) 7049–7061.

[47] C. He, J. Wu, Q. Zhang, Characterizing research leadership on geographically weighted collaboration network, Scientometrics 126 (2021) 4005–4037.

[48] Y. Chen, S. Xiong, L. Mou, X.X. Zhu, Deep quadruple-based hashing for remote sensing image-sound retrieval, IEEE Trans. Geosci. Remote Sens. 60 (2022) 1–14.

[49] Y. Chen, X. Lu, X. Li, Supervised deep hashing with a joint deep network, Pattern Recognit. 105 (2020) 107368.

[50] X. Ma, J. Keung, Z. Yang, X. Yu, Y. Li, H. Zhang, CASMS: Combining clustering with attention semantic model for identifying security bug reports, Inf. Softw. Technol. 147 (2022) 106906.

[51] Y. Zhen, J.W. KEUNG, Y. Xiao, X. Yan, J. Zhi, J. ZHANG, On the significance of category prediction for code-comment synchronization, ACM Trans. Softw. Eng. Methodol. (2022).

[52] Y. Chen, H. Dai, X. Yu, W. Hu, Z. Xie, C. Tan, Improving ponzi scheme contract detection using multi-channel textcnn and transformer, Sensors 21 (2021) 6417.

[53] K. Zhao, Z. Xu, M. Yan, L. Xue, W. Li, G. Catolino, A compositional model for effort-aware Just-In-Time defect prediction on android apps, IET Softw. 16 (2022) 259–278.

[54] T. Cheng, K. Zhao, S. Sun, M. Mateen, J. Wen, Effort-aware cross-project just-in-time defect prediction framework for mobile apps, Front. Comput. Sci. 16 (2022) 1–15.

[55] X. Chen, Y. Zhao, Q. Wang, Z. Yuan, MULTI: Multi-objective effort-aware just-in-time software defect prediction, Inf. Softw. Technol. 93 (2018) 1–13.

[56] K. Zhao, Z. Xu, T. Zhang, Y. Tang, M. Yan, Simplified deep forest model based Just-in-Time defect prediction for android mobile apps, IEEE Trans. Reliab. (2021).

[57] Y. Zhao, Y. Wang, Y. Zhang, D. Zhang, Y. Gong, D. Jin, ST-TLF: Cross-version defect prediction framework based transfer learning, Inf. Softw. Technol. 149 (2022) 106939.

[58] F. Wilcoxon, Individual comparisons by ranking methods, in: Breakthroughs in Statistics, Springer, 1992, pp. 196–202.

[59] J. Ferreira, A. Zwinderman, On the benjamini–hochberg method, Ann. Statist. 34 (2006) 1827–1849.

[60] X. Yu, M. Wu, Y. Jian, K.E. Bennin, M. Fu, C. Ma, Cross-company defect prediction via semi-supervised clustering-based data filtering and MSTrA-based transfer learning, Soft Comput. 22 (2018) 3461–3472.

[61] Z. Xu, S. Pang, T. Zhang, X.-P. Luo, J. Liu, Y.-T. Tang, X. Yu, L. Xue, Cross project defect prediction via balanced distribution adaptation based transfer learning, J. Comput. Sci. Tech. 34 (2019) 1039–1062.

[62] D. Li, M. Liang, B. Xu, X. Yu, J. Zhou, J. Xiang, A cross-project aging-related bug prediction approach based on joint probability domain adaptation and k-means SMOTE, in: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C), IEEE, 2021, pp. 350–358.

[63] X. Yu, J. Liu, W. Peng, X. Peng, Improving cross-company defect prediction with data filtering, Int. J. Softw. Eng. Knowl. Eng. 27 (2017) 1427–1438.

[64] Q. Huang, L. Ma, S. Jiang, G. Wu, H. Song, L. Jiang, C. Zheng, A cross-project defect prediction method based on multi-adaptation and nuclear norm, IET Softw. 16 (2022) 200–213.

[65] S. Feng, J. Keung, X. Yu, Y. Xiao, K.E. Bennin, M.A. Kabir, M. Zhang, COSTE: Complexity-based OverSampling technique to alleviate the class imbalance problem in software defect prediction, Inf. Softw. Technol. 129 (2021) 106432.

[66] X. Yu, J. Liu, Z. Yang, X. Jia, Q. Ling, S. Ye, Learning from imbalanced data for predicting the number of software defects, in: 2017 IEEE 28th International Symposium on Software Reliability Engineering, ISSRE, IEEE, 2017, pp. 78–89.

[67] H. Tong, W. Lu, W. Xing, B. Liu, S. Wang, SHSE: A subspace hybrid sampling ensemble method for software defect number prediction, Inf. Softw. Technol. 142 (2022) 106747.

[68] K.E. Bennin, A. Tahir, S.G. MacDonell, J. Börstler, An empirical study on the effectiveness of data resampling approaches for cross-project software defect prediction, IET Softw. 16 (2022) 185–199.

[69] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, A.E. Hassan, Revisiting common bug prediction findings using effort-aware models, in: 2010 IEEE International Conference on Software Maintenance, IEEE, 2010, pp. 1–10.

[70] K.E. Bennin, K. Toda, Y. Kamei, J. Keung, A. Monden, N. Ubayashi, Empirical evaluation of cross-release effort-aware defect prediction models, in: 2016 IEEE International Conference on Software Quality, Reliability and Security, QRS, IEEE, 2016, pp. 214–221.

[71] K.E. Bennin, J. Keung, A. Monden, Y. Kamei, N. Ubayashi, Investigating the effects of balanced training and testing datasets on effort-aware fault prediction models, in: 2016 IEEE 40th Annual Computer Software and Applications Conference, vol. 1, COMPSAC, IEEE, 2016, pp. 154–163.

[72] M. Yan, Y. Fang, D. Lo, X. Xia, X. Zhang, File-level defect prediction: Unsupervised vs. supervised models, in: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM, IEEE, 2017, pp. 344–353.

[73] Q. Huang, X. Xia, D. Lo, Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction, in: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2017, pp. 159–170.

[74] C. NI, X. XIA, D. LO, X. YANG, A.E. HASSAN, Just-in-time defect prediction on JavaScript projects: A replication study, ACM Trans. Softw. Eng. Methodol. (2022).

[75] L. Qiao, Y. Wang, Effort-aware and just-in-time defect prediction with neural network, PLoS One 14 (2019) e0211359.

[76] Y. Qu, Q. Zheng, J. Chi, Y. Jin, A. He, D. Cui, H. Zhang, T. Liu, Using K-core decomposition on class dependency networks to improve bug prediction model's practical performance, IEEE Trans. Softw. Eng. 47 (2019) 348–366.

[77] X. Du, T. Wang, L. Wang, W. Pan, C. Chai, X. Xu, B. Jiang, J. Wang, CoreBug: improving effort-aware bug prediction in software systems using generalized k-core decomposition in class dependency networks, Axioms 11 (2022) 205.

[78] Y. Qu, J. Chi, H. Yin, Leveraging developer information for efficient effort-aware bug prediction, Inf. Softw. Technol. 137 (2021) 106605.

[79] B. Shuai, H. Li, M. Li, Q. Zhang, C. Tang, Software defect prediction using dynamic support vector machine, in: 2013 Ninth International Conference on Computational Intelligence and Security, IEEE, 2013, pp. 260–263.

[80] C. Arun, C. Lakshmi, Genetic algorithm-based oversampling approach to prune the class imbalance issue in software defect prediction, Soft Comput. (2021) 1–17.

[81] R.S. Wahono, N.S. Herman, Genetic feature selection for software defect prediction, Adv. Sci. Lett. 20 (2014) 239–244.

[82] R.S. Wahono, N. Suryana, S. Ahmad, Metaheuristic optimization based feature selection for software defect prediction, J. Softw. 9 (2014) 1324–1333.

[83] D. Ryu, J. Baik, Effective multi-objective naïve Bayes learning for cross-project defect prediction, Appl. Soft Comput. 49 (2016) 1062–1077.

[84] S. Hosseini, B. Turhan, M. Mäntylä, Search based training data selection for cross project defect prediction, in: Proceedings of the the 12th International Conference on Predictive Models and Data Analytics in Software Engineering, 2016, pp. 1–10.