The Impact of the bug number on Effort-Aware Defect Prediction: An Empirical Study

Peixin Yang
School of Computer Science and Artificial Intelligence, Wuhan University of Technology
Wuhan, China
Sanya Science and Education Innovation Park of Wuhan University of Technology
Sanya, China
peixinyang@whut.edu.cn

Lin Zhu
School of Computer, Wuhan Qingchuan University
Wuhan, China
linzhu_cs@126.com

Wenhua Hu*
School of Computer Science and Artificial Intelligence, Wuhan University of Technology
Wuhan, China
whu10@whut.edu.cn

Jacky Wai Keung
Department of Computer Science, City University of Hong Kong
Hong Kong, China
jacky.keung@cityu.edu.hk

Liping Lu
School of Computer Science and Artificial Intelligence, Wuhan University of Technology
Wuhan, China
luliping@whut.edu.cn

Jianwen Xiang
School of Computer Science and Artificial Intelligence, Wuhan University of Technology
Wuhan, China
Wuhan University of Technology Chongqing Research Institute
Chongqing, China
jwxiang@whut.edu.cn

ABSTRACT

Previous research have utilized public software defect datasets such as NASA, RELINK, and SOFTLAB, which only contain class label information. Almost all Effort-Aware Defect Prediction (EADP) studies are carried out around these datasets. However, EADP studies typically relying on bug density (i.e., the ratio between bug numbers and the lines of code) for ranking software modules. In order to investigate the impact of neglecting bug number information in software defect datasets on the performance of EADP models, we examine the performance degradation of the best-performing learning to rank methods when class labels are utilized instead of bug numbers. The experimental results show that neglecting bug number information in building EADP models results in an increase in the detected bugs. However, it also leads to a significant increase in the initial false alarms, ranging from 45.5% to 90.9% of the datasets, and an significant increase in the modules that need to be inspected, ranging from 5.2% to 70.4%. Therefore, we recommend not only the class labels but also the bug number information should be disclosed when publishing software defect datasets, in order to construct more accurate EADP models.

CCS CONCEPTS

• **Software and its engineering** → **Software defect analysis**; • **Computing methodologies** → *Cross-validation*; *Machine learning approaches*.

KEYWORDS

Software Defect Prediction, Effort-Aware, Bug Number, Learning to Rank

ACM Reference Format:
Peixin Yang, Lin Zhu, Wenhua Hu, Jacky Wai Keung, Liping Lu, and Jianwen Xiang. 2023. The Impact of the bug number on Effort-Aware Defect Prediction: An Empirical Study . In *14th Asia-Pacific Symposium on Internetware (Internetware 2023), August 04–06, 2023, Hangzhou, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3609437.3609458

1 INTRODUCTION

Software bugs have an obvious influence on the software quality and even the economy of software. Early detection of buggy modules in software development is an urgent problem that needs to be solved. Software Defect Prediction (SDP) is one of the feasible methods, which employs machine learning techniques to predict whether there exist software bugs in a software project relying on historical development database and discovered bugs [58, 55, 44, 59]. The conventional approach for modeling SDP involves training a binary classification model on existing software database to predict the presence of bugs in new modules [24]. Accurately predicting

*Corresponding author

Internetware 2023, August 04–06, 2023, Hangzhou, China

ACM ISBN 979-8-4007-0894-7/23/08…$15.00
https://doi.org/10.1145/3609437.3609458

the distribution of bugs in software projects can help testers allocate scarce resources to the predicted buggy modules that require inspection [56, 15, 45, 23].

However, the traditional binary classification-based SDP model does not take into account the density of various software modules, resulting in equal allocation of testing resources among software modules with different bug densities. To solve the problem, Mende et al. [28] and Kamei et al. [20] put forward a concept of Effort-Aware Defect Prediction (EADP), which sorts software instances based on the bug density. To represent effort, both researchers made use of the number of detected Lines Of Code (LOC).

To advance the research on SDP, many researchers have published numerous software defect datasets, such as NASA [36], Eclipse [62], AEEEM [10], SOFTLAB [37], RELINK [41], Change-level datasets [21], Android datasets [6], and PROMISE [5]. The open-source nature of these datasets greatly facilitates subsequent researchers' in-depth exploration of SDP methods. Almost all EADP studies are carried out around these open-source datasets. But EADP studies typically rely on bug density for ranking software modules, thus requiring bug number information in the training set. However, our investigation found some EADP studies [27, 28, 20, 21, 29, 33, 26, 43, 53, 11, 7, 46, 52, 14, 17, 34, 60, 42, 61, 9] employed class labels (buggy or non-buggy) instead of bug numbers for modeling, which can impact the prediction performance. Therefore, we aim to investigate the impact of using class labels instead of bug numbers on the performance of EADP.

Our investigation reveal that six open-source datasets are commonly utilized in EADP studies, including NASA [36], Eclipse [62], PROMISE [5], AEEEM [10], Change-level datasets [21], and Android datasets [6]. Some datasets of them (e.g., PROMISE) contain bug numbers, while some of them lack this information. Bug numbers can affect the performance of the models, as EADP models usually rely on bug density to rank software modules. Therefore, we seek to investigate the performance degradation of the best-performing learning to rank methods when class labels are used instead of bug numbers. Recently, Yang et al. [48] proposed the differential evolution based supervised method DEJIT to build the Just-In-Time (JIT) EADP model. Their experimental results have demonstrated DEJIT's superiority over existing models in defect prediction across both cross-release and cross-project setups. In addition, a recent research by Yu et al. [54] showed that LTR-linear (Learning-To-Rank with the linear model) demonstrated outstanding performance in the cross-release setup, while both LTR-linear and LTR-logistic exhibit excellent overall performance in the cross-project setup. Therefore, we employ these three learning to rank algorithms to construct the EADP models on the PROMISE dataset. Our experiment results indicate that:

(1) The models established using class labels instead of bug numbers on average increase the number of Initial False Alarms (IFA) ranging from 7 to 9 modules under the cross-project setting and from 3 to 7 modules under the cross-release setting. Moreover, these three learning-to-rank algorithms increase IFA ranging from 45.5% to 90.1% across the 11 cross-project experiments, and from 50% to 70% across the 30 cross-release experiments. However, IFA is a critical metric, and it needs to be controlled within 10. According to Kochhar et al.'s research [22] almost all respondents (close to 98%) agreed that examining more than ten actual clean modules was beyond their acceptable level. If the first few recommendations are all false alarms, testers are not willing to use the prediction models.

(2) When inspecting the top 20% LOC, LTR-linear established using class labels instead of bug numbers exhibit significant increases in PMI@20% (Proportion of Module Inspected when the top 20% LOC are inspected) by 15.4% under the cross-project setting and 10.4% under the cross-release setting. LTR-logistic significantly increases PMI@20% by 46.3% under the cross-project setting and 11.4% under the cross-release setting. LTR-linear and LTR-logistic show a significant reduction of Precision@20% by 2.5% and 21%, respectively. DEJIT increases PMI@20% by 1% and 5.2% under the cross-project and cross-release setups, respectively. This implies that the testing team need expend a considerable amount of time on the overhead of switching between modules in most situation, as high PMI@20% indicates that they need to inspect a greater number of modules.

(3) However, in terms of the average values of PofB@20% (Proportion of the found Bugs when the top 20% LOC are inspected), LTR-linear, LTR-logistic, and DEJIT under the cross-project setting exhibit significant increases of 7.2%, 17.6%, and 9.1%, respectively. Under the cross-release setting, these three methods show significant increases of 7.8%, 7.6%, and 5.4%, respectively. This indicates that models established using class labels can lead to the detection of more bugs. Using class labels to construct models instead of relying on bug numbers tends to prioritize modules with less LOC, as they have a higher defect density. As a result, during the inspection of the same amount of LOC (i.e., the top 20% LOC), software testers are compelled to examine a greater number of modules, leading to a higher PMI@20% value. Consequently, they are presented with increased opportunities to identify buggy modules and detect defects, yielding higher PofB@20% values. However, the experiment results (1) and (2) suggest that the increase in PofB@20% comes at the expense of significant increases in IFA and PMI@20%.

In summary, using only the class labels to construct EADP models tends to degrade the performance. Therefore, we recommend not only the class labels but also the bug numbers should be disclosed when publishing software defect datasets, in order to facilitate the establishment of more accurate EADP models by future researchers.

2 BACKGROUND

The objective of EADP is to develop a forecasting model from the training dataset that can sort new modules relying on the predicted bug density. We have selected three EADP algorithms, including DEJIT, LTR-linear, and LTR-logistic. These algorithms have shown excellent performance in the module-level EADP algorithms, according to recent research [48, 54, 57].

According to Yu et al.'s study [54, 57], LTR-linear exhibits superior performance under cross-release setup. LTR-linear [46] is one of the listwise algorithms. LTR-linear trains a linear model that is identical to the one used in linear regression, i.e.,

$$y = f(\boldsymbol{b}, \boldsymbol{x}) = b_0 + b_1 x_1 + b_2 x_2 + \ldots + b_m x_m, \tag{1}$$

but it employs the composite differential evolution approach to optimize the PofB@20% metric directly and determine the optimal value for $\boldsymbol{b}$.

In addition, some studies [17, 8] used the logistic regression model to describe the relationship between software features and the number of defects or defect density:

$$y = \frac{1}{1 + e^{-(b_0 + b_1 x_1 + b_2 x_2 + \ldots + b_m x_m)}}. \quad (2)$$

According to Yu et al.'s study [54], they also investigate the performance of LTR when using the logistic regression model to build EADP models. They denote LTR with linear model as LTR-linear, and denote LTR with logistic regression model as LTR-logistic.

Furthermore, as presented in Yang et al.'s research [48], DEJIT has been proven to possess superior performance. DEJIT uses the Density-Percentile-Average (DPA) as optimization objective on the training set and employs logistic regression to construct the prediction model. To maximize DPA on the training set, DEJIT uses the differential evolution algorithm to determine the coefficients of the logistic regression model.

3 RELATED WORK

Mende et al. [27] proposed a simple model and compared it with other classification methods. The results using NASA datasets indicated that the simple model performed well when considering LOC as effort. Subsequently, Mende et al. [28] further incorporated effort into the SDP procedure and evaluated the predictive performance of Random Forest (RF) on 12 publicly available NASA and three Eclipse datasets. Kamei et al. [20] further explored the performance of three EADP classification algorithms, (i.e., Linear Regression (LR), CART, and RF) on three Eclipse datasets. Yang et al. [51] assessed the practical applications of slice-based cohesion metrics in EADP. Yang et al. [49] and Bennin et al. [4] revisited prior research and delved further into machine learning methods and impacting factors for enhancing the performance of EADP. Subsequently, Bennin et al. [3] and Yu et al. [53] investigated the data resampling techniques to further enhance the performance of EADP. Yang et al. [50] proposed an unsupervised method called ManualUp and Liu et al. [25] put forward an unsupervised model based on code churn called CCUM. The results on change-level datasets showed that their unsupervised models outperformed supervised models in JIT EADP. Yan et al. [43] built upon the research of Yang et al. [50] and further compared unsupervised and supervised EADP models on the PROMISE dataset. Miletić et al. [29] explored the use of cross-release code churn to predict software defects in evolution. The results on Eclipse indicated that the model obtained a better performance while cross-release code churn was employed. Qu et al. [35] and Du et al. [11] adopted the k-core decomposition on class dependency networks to investigate the distribution of software bugs and enhance the efficacy of EADP models. They leveraged LR and RF techniques to establish EADP models and validated the proposed top-core equation on 18 Java software datasets, demonstrating an improvement in the performance of EADP. Kamei et al. [21] constructed a model using LR based on various factors of software change characteristics, and conducted a comprehensive study on commercial and change-level projects in multiple domains. Fu et al. [14] put forward a simple supervised model called OneWay for JIT EADP. The experiments showed that OneWay outperformed most unsupervised models. Huang et al. [17] re-examined previous work on EADP and introduced a supervised model called CBS+ that has demonstrated superior performance compared to both unsupervised and supervised models. Chen et al. [8] put forward a supervised method for building JIT EADP models based on multi-objective optimization. The approach surpassed the performance of advanced supervised and unsupervised techniques. Qiao et al. [34] put forward an approach for JIT EADP based on deep learning, named NNR. The approach used a neural network to select relevant features for prediction and has shown promising results while comparing to other methods in the literature. Zhang et al. [60] put forward an semi-supervised model named EATT for JIT EADP, which outperformed existing supervised and unsupervised models in terms of various labeled rates on change-level datasets. Guo et al. [16] analyzed the impact of classification accuracy on capability and interpretation of JIT EADP, while Fan et al. [12] specifically studied the impact of mislabeled changes. Xu et al. [42], Cheng et al. [9], and Zhao et al. [61] proposed three JIT EADP models for Android applications. Panichella et al. [33] and Yang et al. [48] employed Genetic Algorithms (GA) to train their predictive models. The results obtained on the PROMISE dataset and other six open-source projects indicate that regression models trained by GA were significantly superior to traditional regression models. Ma et al. [26] explored the correlation between defect-proneness and single network measures in EADP by constructing a software source code network in POMISE dataset. Jiang et al. [19] reviewed past model evaluation techniques and introduced and discussed the cost curve. Based on the publicly available NASA database, they indicated that the optimal model cannot be selected without considering project cost features. Ni et al. [7] proposed an improved supervised method called EASC through comparing existing EADP methods. Further comparisons were made with unsupervised methods. Based on experiments on four datasets (NASA, AEEEM, RELINK, PROMISE), the results showed that EASC has better prediction performance in cross-project scenarios. Nguyen et al. [31] and Muthukumaran et al. [30] investigated whether algorithms for defect prediction based on effort performed better traditional methods for defect prediction. Wang et al. [39] put forward a top-k learning ranking method and presented a novel data resampling approach named SMOTE-PENN to address the imbalance problem. The results on PROMISE datasets indicated that this approach outperformed other algorithms. Yang et al. [47] delved into the examination of the efficacy of two new models in cross-version defect prediction and compared their performance with conventional regression models. The study conducted on the PROMISE dataset revealed that the two models displayed superior performance. Ulan et al. [38] put forward an automated method relying on unsupervised learning for weighted metrics aggregation in EADP. The results on the change-level dataset showed that the unsupervised method can function as an agnostic predictor when ground truth is not available.

4 EXPERIMENT SETTING

4.1 Datasets

Numerous publicly-accessible module-level bug datasets, including NASA [36], RELINK [41], and SOFTLAB [37], which hold only class labels. However, EADP usually rely on bug density for ranking software modules. Therefore, we conduct experiments based on PORMISE dataset as it provide bug number information. The

PROMISE dataset comprises of 41 releases from 11 open-source software projects, and every module in this dataset comprises 20 distinct code features. The experimental projects originate from diverse application domains, exhibiting varying sizes, ranging from 205 to 965 modules, and differing proportions of buggy modules, ranging from 2.2% to 98.8%. This diversity proves advantageous in enhancing the generalization of our experiments. Furthermore, the PROMISE dataset are publicly accessible, allowing other researchers to replicate the findings of this paper to a significant extent.

4.2 Validation Setup

We employ the following two evaluative configurations.

Cross-Project Setup: We employ eleven software projects as the experimental datasets. It is difficult to determine which project to use as a training set to build a better model, so we choose one dataset as a within-project for testing and the remaining ten cross-projects for training at each time. The cross-project experimental setting is the same as Huang et al.'s [18] study. From the 41 versions of 11 projects available in the PROMISE dataset, we select only the original release to be used as either the training or testing dataset.

Cross-Release Setup: In this scenario, the prediction model is assessed within the same project, while the chronological order of modules is taken into consideration. For datasets at the module level, we employ the preceding and current releases for training and testing, respectively. For example, we train on Ant1.3 and test on Ant1.4. Finally, we attain 30 sets of paired training and test datasets.

To mitigate potential biases during hyperparameter tuning, we repeat the training and testing process for each pair ten times under both cross-project and cross-release setups.

4.3 Evaluation Measures

In our study, we generally constrain the effort to 20% of the overall LOC. Fenton et al. [13] and Andersson et al. [2] have noted that roughly 20% LOC are responsible for 80% or more of the bugs, leading to the use of 20% as a commonly accepted cutoff value for determining the effort required for bug inspection, as noted in studies by [32, 50, 45]. For some specific metrics such as PofB, PMI, and PofB/PMI, we also consider absolute LOC, specifically 3000 LOC and 5000 LOC.

Given a dataset with N modules, Q LOC, and M bugs, there are P buggy modules out of the N modules. Under the assumption of 20% LOC, n modules are inspected, yielding m bugs and p buggy modules found. Additionally, upon discovering the first buggy module, n' modules have been inspected.

Precision@20% measures the proportion of actual buggy modules among all inspected modules. A lower Precision@20% could affect developer confidence in the EADP model [17].

$$\text{Precision@20\%} = p/n \tag{3}$$

Recall@20% represents the proportion of actual buggy modules among all the actual buggy modules that have been inspected. More buggy modules could be discovered with a higher value of Recall@20%.

$$\text{Recall@20\%} = p/P \tag{4}$$

F1@20% takes into account both the Precision@20% and Recall@20% simultaneously, which is a harmonic average of the two.

$$F1@20\% = \frac{2 \times Precision@20\% \times Recall@20\%}{Precision@20\% + Recall@20\%} \tag{5}$$

PofB@effort is the Proportion of the inspected Bugs among all bugs, where a higher value means a greater number of bugs can be detected. This metric is equivalent to "Recall@effort" when each buggy module only has one bug. Previous research [17], [8] have referred to "PofB" as "Recall" as the class labels of modules was treated as bug numbers. Due to the varying sizes of different instances in the dataset, some instances may have a hundred LOC, while others may have a thousand lines of code. Inspired by the work of Ni et al. [7], we consider two types of PofB@effort to comprehensively investigate the performance. These are the relative LOC of PofB and the absolute LOC of PofB.

$$\text{PofB@20\%} = m/M, \text{ where effort equals to 20\% of total LOC} \tag{6}$$

$$\text{PofB@3000} = m/M, \text{ where effort equals to 3000 LOC} \tag{7}$$

$$\text{PofB@5000} = m/M, \text{ where effort equals to 5000 LOC} \tag{8}$$

PMI@effort refers to the Proportion of Modules Inspected when examining the top LOC. A high PMI@effort implies that a larger number of modules need to be inspected for the same amount of LOC. Similar to PofB@effort, we also use the relative LOC of PMI and the absolute LOC of PMI to evaluate the performance.

$$\text{PMI@20\%} = n/N, \text{ where effort equals to 20\% of total LOC} \tag{9}$$

$$\text{PMI@3000} = n/N, \text{ where effort equals to 3000 LOC} \tag{10}$$

$$\text{PMI@5000} = n/N, \text{ where effort equals to 5000 LOC} \tag{11}$$

PofB/PMI@effort is employed to assess the efficiency of the EADP models. A higher value of PofB/PMI@effort implies that more bugs will be discovered for per unit of LOC inspected.

$$\text{PofB/PMI@20\%} = \text{PofB@20\%/PMI@20\%} \tag{12}$$

$$\text{PofB/PMI@3000} = \text{PofB@3000/PMI@3000} \tag{13}$$

$$\text{PofB/PMI@5000} = \text{PofB@5000/PMI@5000} \tag{14}$$

P_{opt} is a metric predicated on the cumulative lift chart depicted in Figure 1, devised by Kamei et al. [20] to assess the predictive model. The chart shows the cumulative percentage of LOC on the x-axis and the cumulative percentage of bugs found given that LOC on the y-axis. The chart displays three curves, representing the prediction model, the optimal model, and the worst model. The modules are sorted based on their predicted bug numbers or densities from the prediction model, actual bug numbers or bug densities from the optimal model, and increasing actual bug numbers or bug densities from the worst model. The measure evaluates the effectiveness of the prediction EADP model in finding bugs.

P_{opt} is calculated as follows:

$$P_{opt} = \frac{\text{Area(prediction)} - \text{Area(worst)}}{\text{Area(optimal)} - \text{Area(worst)}} \tag{15}$$

where Area() is the area under the corresponding curve.

A greater P_{opt} value implies a more insignificant disparity between the prediction and optimal models. P_{opt} differs from previous evaluation metrics, as it assesses the global ranking of predicted occurrences.

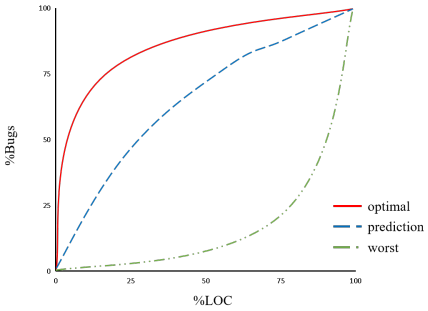


Figure 1: A cumulative lift chart.

IFA is the number of Initial False Alarms before the testing team could discover the first buggy module. A higher value of IFA suggests that a greater amount of effort, in terms of module inspection, would be invested before discovering the first bug.

$$\text{IFA} = n' \tag{16}$$

4.4 Procedure

We organize the experiment by answering the following research question.

> **What is the effect of bug numbers in building module-level EADP models?**

The main procedure of the experiment is designed in Figure 2. Specifically, we first generate a training dataset and a testing dataset under two different setups, as described in Subsection 4.2. We then construct two EADP models using the algorithms described in Section 2: one with bug numbers and the other without bug numbers (i.e., using class labels). Finally, we compare the performance of these two models and analyze the experimental results.

4.5 Statistical Comparison Test

Wilcoxon signed-rank test [40] is a type of statistical hypothesis test for evaluating performance metrics. The Bonferroni correction [1] is employed to address issues with multiple comparisons. The original hypothesis is that the predicted models after training have no significant difference, with a significance level set at 0.05. If the Bonferroni-corrected p-value is less than 0.05, we reject the original hypothesis; otherwise, we accept the original hypothesis.

5 RESULTS

Approach. We compute the performance reduction of the best learning to rank algorithms (including LTR-linear, LTR-logistic and DEJIT) in both cross-project and cross-release scenarios that are trained using class labels instead of bug numbers. We then present the distribution of performance reduction for each evaluation measure using boxplots. We employ the Wilcoxon signed-rank test [40] with a Bonferroni correction [1] to assess if the performance reduction is statistically significant. Different colors of the boxplots indicate different magnitudes of the performance reduction. Red represents the magnitudes of the performance reduction are significant, while the black represents there is no significant difference.

Table 1: The IFA value of the three methods under each cross-project experiment.

Test Dataset	LTR-linear	LTR-log	DEJIT
Ant1.3	-4.4	-29.1	-24.3
Camel1.0	-2	-28.8	-26.9
Ivy1.1	-0.3	-3.5	-3.8
Jedit3.2	1.7	-3	-4.8
Log4j1.0	0.1	0.6	-10.2
Lucene2.0	3.4	-2.1	-3.4
Poi1.5	1.8	-1.2	-1.8
Synapse1.0	-0.1	-1.4	-12.5
Velocity1.4	0	0.2	0.7
Xalan2.4	4.4	-2.3	-3.3
Xerces1.1	-1.6	-1.1	-0.8

Table 2: The IFA value of the three methods under each cross-release experiment.

Dataset pair	LTR-linear	LTR-log	DEJIT
Ant1.3_Ant1.4	-1.4	3.3	0.5
Ant1.4_Ant1.5	5	-14.4	-3.8
Ant1.5_Ant1.6	-0.7	-0.7	-0.1
Ant1.6_Ant1.7	-8.4	-6.4	-1.8
Camel1.0_Camel1.2	-11	-11	0
Camel1.2_Camel1.4	-9.3	-12	-28.8
Camel1.4_Camel1.6	-20	-20	0
Ivy1.1_Ivy1.4	9.9	15.1	-0.4
Ivy1.4_Ivy2.0	15.4	-68.6	-0.4
Jedit3.2_Jedit4.0	-0.8	-0.8	-0.7
Jedit4.0_Jedit4.1	-8.9	-5	-8.2
Jedit4.1_Jedit4.2	-31.5	-1.6	1
Jedit4.2_Jedit4.3	-0.1	-0.1	1.6
Log4j1.0_Log4j1.1	-1	-1	-0.1
Log4j1.1_Log4j1.2	0	0	0
Lucene2.0_Lucene2.2	0.1	-3	-0.2
Lucene2.2_Lucene2.4	-1	0	0
Poi1.5_Poi2.0	-11.1	-14.7	-20
Poi2.0_Poi2.5	-0.7	-0.7	0
Poi2.5_Poi3.0	-4.1	3.8	1.1
Synapse1.0_Synapse1.1	-1	0.6	0.1
Synapse1.1_Synapse1.2	-0.8	-1	0
Velocity1.4_Velocity1.5	-2.9	-4	0
Velocity1.5_Velocity1.6	4.2	4.1	0
Xalan2.4_Xalan2.5	-3.6	-4.8	0
Xalan2.5_Xalan2.6	2.4	-22.7	-3
Xalan2.6_Xalan2.7	0	0	0
Xerces1.1_Xerces1.2	8.2	1.4	-4.1
Xerces1.2_Xerces1.3	-22.2	-43	-2.8
Xerces1.3_Xerces1.4	-1	2.6	-0.1

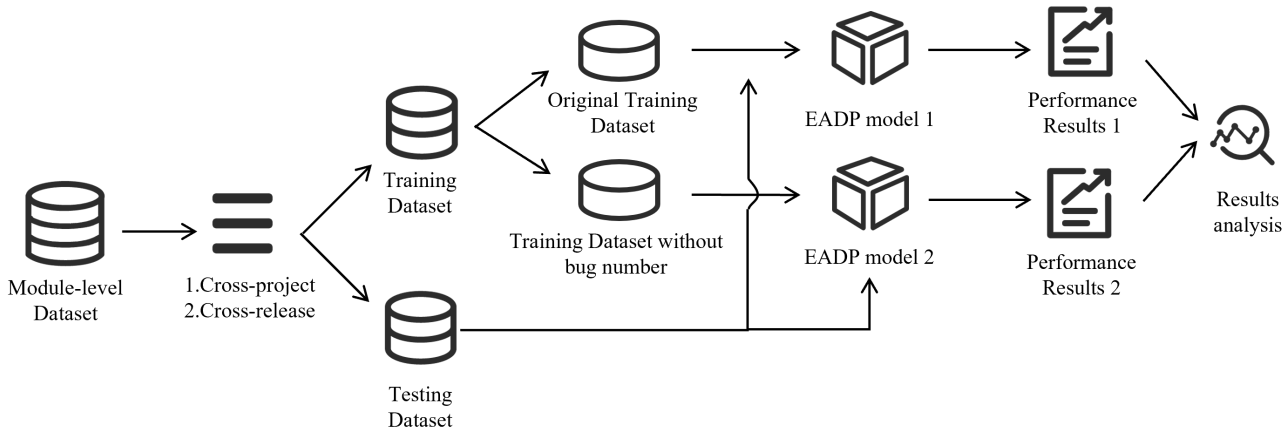


Figure 2: An overview of the procedure.

Table 3: The ten metric values of DEJIT under each cross-project experiment when inspecting the top 20% LOC or 3000/5000 LOC.

Test Dataset	PofB@20%	PMI@20%	PofB/PMI @20%	Popt	PofB@3000	PMI@3000	PofB/PMI @3000	PofB@5000	PMI@5000	PofB/PMI @5000
Ant1.3	-0.038	-0.012	-0.063	-0.007	0.039	-0.009	0.118	-0.035	-0.011	-0.067
Camel1.0	-0.022	-0.013	-0.018	-0.010	0.024	-0.007	0.076	-0.028	-0.015	-0.033
Ivy1.1	-0.280	-0.003	-0.407	-0.278	-0.201	-0.005	-0.367	-0.270	-0.006	-0.402
Jedit3.2	-0.221	-0.007	-0.322	-0.200	-0.060	-0.004	-0.207	-0.071	-0.005	-0.196
Log4j1.0	-0.035	-0.002	-0.063	-0.072	-0.075	-0.001	-0.158	-0.028	-0.004	-0.042
Lucene2.0	-0.102	0.000	-0.175	-0.174	-0.056	0.011	-0.135	-0.102	-0.006	-0.231
Poi1.5	-0.089	-0.010	-0.147	-0.166	-0.009	-0.003	-0.023	-0.007	0.002	-0.021
Synapse1.0	-0.005	-0.014	-0.006	-0.054	-0.015	-0.005	-0.040	-0.010	-0.004	-0.020
Velocity1.4	-0.101	-0.040	-0.095	-0.048	-0.083	-0.023	-0.134	-0.089	-0.031	-0.113
Xalan2.4	-0.025	0.003	-0.041	-0.034	-0.004	0.001	-0.020	-0.004	0.002	-0.016
Xerces1.1	-0.074	-0.001	-0.091	-0.067	-0.095	-0.001	-0.163	-0.100	0.002	-0.159

Table 4: The ten metric values of LTR-linear under cross-project setting when inspecting the top 20% LOC or 3000/5000 LOC.

Test Dataset	PofB@20%	PMI@20%	PofB/PMI @20%	Popt	PofB@3000	PMI@3000	PofB/PMI @3000	PofB@5000	PMI@5000	PofB/PMI @5000
Ant1.3	0.067	-0.226	0.565	0.072	0.018	-0.218	0.450	0.109	-0.238	0.758
Camel1.0	0.036	-0.119	0.231	-0.021	0.087	-0.112	0.501	0.069	-0.120	0.341
Ivy1.1	-0.293	-0.285	-0.111	-0.255	-0.286	-0.340	-0.221	-0.275	-0.282	-0.085
Jedit3.2	-0.152	-0.209	-0.064	-0.161	-0.053	-0.195	0.432	-0.049	-0.216	0.227
Log4j1.0	-0.032	-0.056	0.001	0.003	-0.085	-0.053	-0.134	0.003	-0.055	0.063
Lucene2.0	-0.051	-0.119	0.077	-0.042	-0.080	-0.069	-0.132	-0.066	-0.090	-0.019
Poi1.5	-0.110	-0.135	0.031	-0.144	-0.117	-0.258	3.445	-0.092	-0.173	0.181
Synapse1.0	0.047	-0.099	0.173	0.071	0.014	-0.086	0.142	0.041	-0.097	0.171
Velocity1.4	-0.190	-0.160	-0.036	-0.094	-0.290	-0.236	-0.163	-0.237	-0.198	-0.075
Xalan2.4	-0.031	-0.206	0.206	-0.013	-0.019	-0.030	0.014	-0.013	-0.066	1.971
Xerces1.1	-0.085	-0.088	0.000	-0.054	-0.162	-0.090	-0.123	-0.106	-0.057	-0.075

Visualizations: Figure 3 presents the distribution of performance reduction of DEJIT, LTR-linear, and LTR-logistic trained using the class labels for 15 effort-aware evaluation measures under the cross-project setting. Figure 4 presents the distribution of the performance reduction of the three methods that trained using the class labels

Table 5: The ten metric values of LTR-logistic under cross-project setting when inspecting the top 20% LOC or 3000/5000 LOC.

Test Dataset	PofB@20%	PMI@20%	PofB/PMI @20%	Popt	PofB@3000	PMI@3000	PofB/PMI @3000	PofB@5000	PMI@5000	PofB/PMI @5000
Ant1.3	0.061	-0.162	0.355	0.085	0.006	-0.230	0.414	0.074	-0.185	0.501
Camel1.0	0.012	-0.480	4.326	-0.001	-0.071	-0.347	1.972	0.001	-0.428	3.889
Ivy1.1	-0.384	-0.673	15.018	-0.377	-0.387	-0.559	-0.692	-0.352	-0.652	15.041
Jedit3.2	-0.222	-0.556	4.644	-0.171	-0.075	-0.285	0.729	-0.108	-0.344	0.723
Log4j1.0	0.084	-0.455	3.694	0.112	0.051	-0.416	3.730	0.136	-0.458	2.865
Lucene2.0	-0.090	-0.299	0.893	-0.061	-0.107	-0.236	4.501	-0.105	-0.270	4.214
Poi1.5	-0.286	-0.481	1.698	-0.347	-0.138	-0.281	2.623	-0.174	-0.345	2.426
Synapse1.0	0.246	-0.365	2.877	0.155	0.171	-0.322	5.796	0.236	-0.352	3.436
Velocity1.4	-0.707	-0.657	-0.229	-0.577	-0.471	-0.429	-0.129	-0.542	-0.506	-0.128
Xalan2.4	-0.035	-0.268	0.359	-0.021	-0.012	-0.021	0.368	-0.020	-0.040	0.899
Xerces1.1	-0.613	-0.704	1.086	-0.444	-0.576	-0.550	0.884	-0.609	-0.615	1.074

Table 6: The mean of ten metric values of the DEJIT, LTR-linear, and LTR-logistic under cross-release setting when inspecting the top 20% LOC or 3000/5000 LOC.

Method	PofB@20%	PMI@20%	PofB/PMI @20%	Popt	PofB@3000	PMI@3000	PofB/PMI @3000	PofB@5000	PMI@5000	PofB/PMI @5000
DEJIT	-0.078	-0.052	0.442	-0.061	-0.023	-0.029	-0.139	-0.029	-0.037	0.159
LTR-linear	-0.076	-0.104	0.477	0.251	-0.042	-0.087	0.882	-0.056	-0.104	1.036
LTR-logistic	-0.054	-0.114	0.460	0.270	-0.048	-0.099	1.642	-0.053	-0.112	1.024

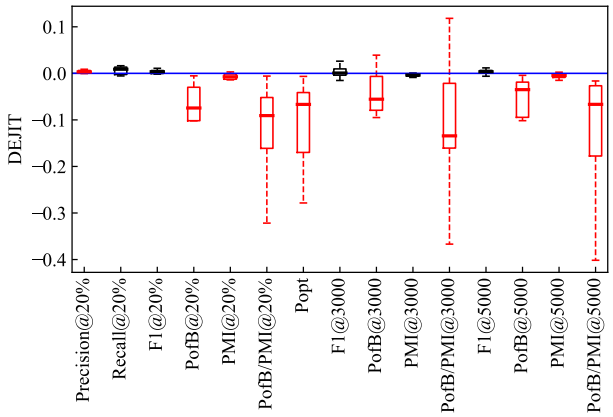
under the cross-release setting. Since the scales between IFA and other performance measures are different, we use a single figure (i.e., Figure 5) to show the distribution of the performance reduction on IFA. Table 1 and Table 2 present the IFA values for the DEJIT, LTR-linear, and LTR-logistic methods under each cross-project and cross-release experiment. Table 3, Table 4, and Table 5 display the 10 metric values for the three methods in each cross-project experiment. We use Table 6 to present the average results of these three methods under cross-release setup.

Results on IFA metric: From Figure 5, it is evident that the models established using class labels show a significant increase in IFA except for DEJIT under the cross-project setting. The models established using class labels on average increase IFA by from 7 to 9 modules under the cross-project setting and by from 3 to 7 modules under the cross-release setting. This implies that testing personnel would need to inspect more modules before detecting the first buggy module. Moreover, according to Table 1 and Table 2, these three methods established using class labels instead of bug numbers increase IFA on 10, 5, and 9 experiments under 11 cross-project experiments, and on 15, 21, and 20 experiments under 30 cross-release experiments. In particular, the LTR-logistic method increases IFA by a maximum of 68.6 under the cross-release experiment setting and by a maximum of 29.1 under the cross-project experiment setting. Moreover, for the 11 cross-project experiments, these three learning-to-rank algorithms increase IFA in 45.5% to 90.1% of the datasets, and for the 30 cross-release experiments, they increase IFA in 50% to 70% of the experiments.

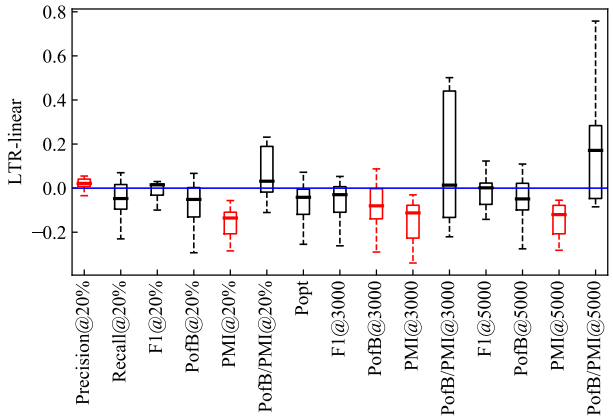
DEJIT increases IFA value by more than 10 in 36.4% of the datasets under the cross-project setting and in 6.7% of the datasets under the cross-release setting. LTR-linear increases IFA by more than 10 in 16.7% of the datasets under the cross-release setting. LTR-logistic increases IFA value by more than 10 in 18.2% and 26.7% of the datasets under the cross-project and cross-release settings, respectively. However, it is crucial to control IFA within 10, as emphasized in Kochhar et al.'s research [22]. Such a significant increase in IFA values can render other metrics of the model statistically insignificant, as testers are not willing to use the prediction models if the first few recommendations are all false alarms. Clearly, established using class labels would lead to more resource consumption for the testing personnel.

Results under cross-project setting: (1) When inspecting the top 20% LOC, DEJIT established using class labels instead of bug numbers under the cross-project setting shows significant increases of 9.1%, 12.9%, and 10.1% in PofB@20%, PofB/PMI@20%, and P_{opt}, respectively. This indicates that the model established using class labels can identify more bugs. However, on the other hand, using class labels to build the model can significantly increases PMI@20%, implying that testing personnel will need to inspect more modules, resulting in increased time overhead due to module switching. When checking a fixed number of LOC (i.e., inspecting 3000 LOC and 5000 LOC), DEJIT established using class labels can increases the average value of PofB@3000 and PofB@5000 by 4.9% and 6.8%, respectively.

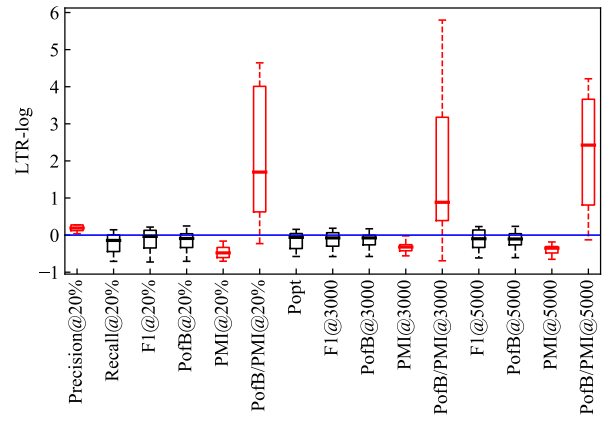
(2) When inspecting the top 20% LOC, LTR-linear established using class labels shows a significant increase of 15.4% in PMI@20% and a decrease of 2.5% in Precision@20%, while the other metrics do not exhibit significant differences. When checking 3000 LOC, LTR-linear established using class labels significantly increases PMI@3000 by 15.3% and PofB@3000 by 8.8%. However, when checking 5000 LOC, only PMI@5000 shows a significant increase of 14.5%.



(a) DEJIT under cross-project setting

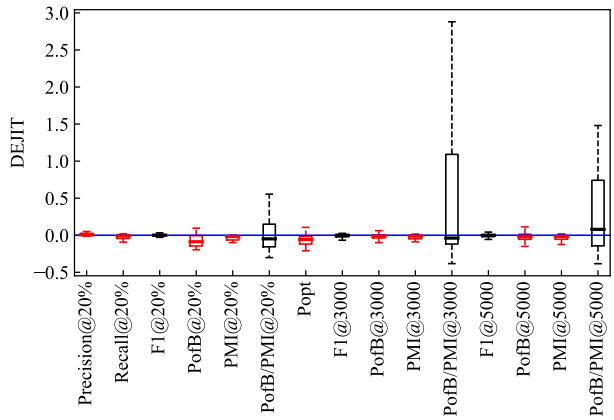


(b) LTR-linear under cross-project setting

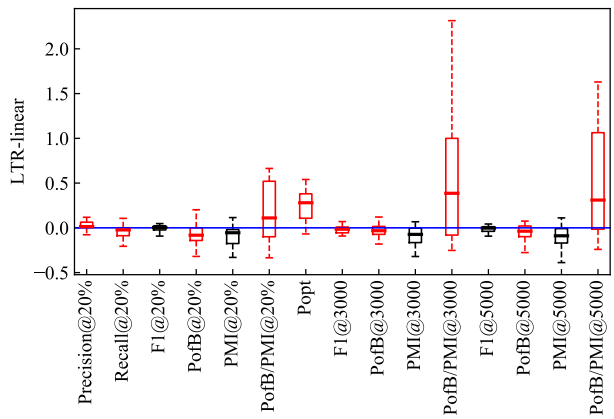


(c) LTR-logistic under cross-project setting

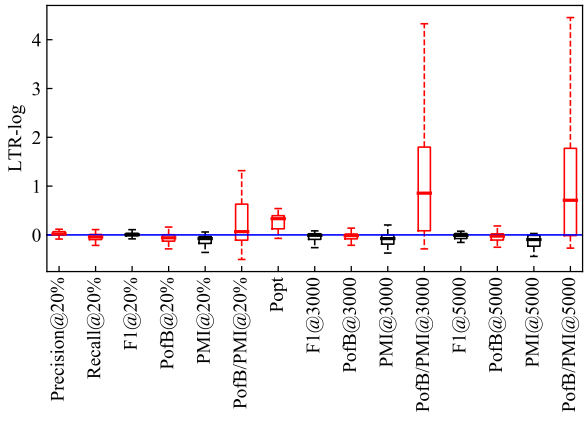
Figure 3: The performance reduction of the learning to rank algorithms that trained using the class labels for 6 effort-aware performance measures under cross-project setting. A blue line indicates a performance difference of zero (i.e., no reduction). (Different colors of the boxplots indicate different magnitudes of the difference. Red represents that the magnitudes of the difference are significant, while black represents there are no significant difference.)



(a) DEJIT under cross-release setting



(b) LTR-linear under cross-release setting



(c) LTR-logistic under cross-release setting

Figure 4: The performance reduction of the learning to rank algorithms that trained using the class labels for 6 effort-aware performance measures under cross-release setting. A blue line indicates a performance difference of zero (i.e., no reduction). (Different colors of the boxplots indicate different magnitudes of the difference. Red represents that the magnitudes of the difference are significant, while black represents there are no significant difference.)

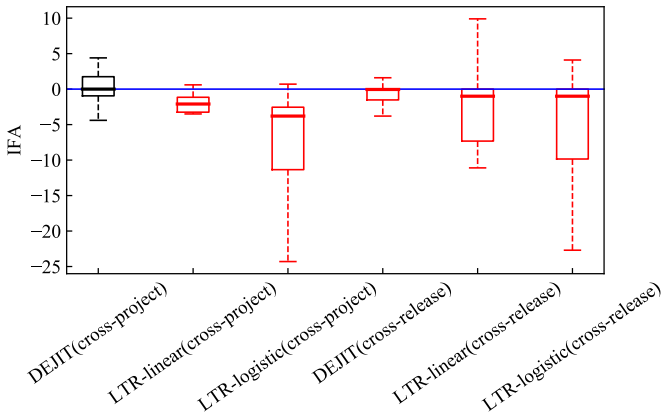


Figure 5: The performance reduction of the learning to rank algorithms that trained using the class labels for IFA. A blue line indicates a performance difference of zero (i.e., no reduction). (Different colors of the boxplot indicate different magnitudes of the difference. Red represents that the magnitudes of the difference are significant, while black represents there are no significant difference.)

This indicates that using class labels to build the model can significantly increases the number of modules that need to be inspected.

(3) When inspecting the top 20% LOC, LTR-logistic established using class labels shows a significant increase of 46.3% in PMI@20% and a decrease of 21% in Precision@20%. It is worth noting that the models established using class labels significantly reduces PofB/PMI@20% by 316%, indicating that the increase in PMI@20% is much larger than the increase in PofB@20%. When checking 3000 LOC and 5000 LOC, it shows a significant increase of 33.4% and 38.1% in PMI@3000 and PMI@5000, respectively, and a significant increase of 183% and 317% in PofB/PMI@3000 and PofB/PMI@5000, respectively.

(4) Thus, in the cross-project setting, using class labels to build the model will increase PofB@effort to some extent, increase PMI@effort and decrease PofB/PMI@effort. This means that testing personnel will need to spend a considerable amount of time on the overhead of switching between modules and they can find more bugs, as high PMI@effort indicates that they need to inspect a greater number of modules when checking a certain LOC.

Results under cross-release setting: (1) When inspecting the top 20% LOC, DEJIT established using class labels significantly increases the average PofB@20% by 7.8% and P_{opt} by 6.1%, but at the same time, it also significantly increases PMI@20% by 5.2%. When checking the top 3000 LOC and top 5000 LOC, PofB@3000 and PofB@5000 both show a significant increase of 2.9%, PMI@3000 and PMI@5000 show a significant increase of 2.9% and 3.7%, respectively. This indicates that in this setting, the DEJIT established using class labels can detect more bugs, but meanwhile it also need inspect more modules.

(2) When inspecting the top 20% LOC, LTR-linear established using class labels significantly increases PofB@20% by 7.6%. However, both LTR-linear and LTR-logistic established using class labels significantly increase the average PofB/PMI@20% by 47.7% and 46%, respectively, and significantly decrease the average P_{opt} by 25.1% and 27%. When checking the top 3000 LOC and top 5000 LOC, LTR-linear established using class labels significantly increases the average PofB/PMI@3000 and PofB/PMI@5000 by 88.2% and 103.6%, respectively. Similarly, LTR-logistic established using class labels significantly increases the average PofB/PMI@3000 and PofB/PMI@5000 by 164.2% and 102.4% for the two settings. Both methods show an average increase of 10.4% and 11.4% in PMI, respectively.

(3) Therefore, in the cross-release setting, we can conclude that these methods established using class labels can increase the number of modules that need to be inspected. It can also increase the number of detected bugs, but at the cost of sacrificing a amount of Precision@20%, PMI@20%, and P_{opt}. Taking into consideration the significant decrease in the PofB/PMI@20% metric, we believe that the extent of sacrificing PMI@20% is greater than the improvement in PofB@20%.

Summary: Under cross-project and cross-release settings, methods using label-based will significantly increases IFA on a substantial proportion of the datasets. If the first few recommendations are all false alarms, testers are not willing to use the prediction models. Additionally, using class labels to build the model also leads to a significant increase in PMI@effort and PofB@effort, indicating that testing personnel will need to inspect a larger number of modules. While it can result in detecting more bugs, it comes at the cost of sacrificing a considerable amount of IFA, PMI@effort, and P_{opt}.

6 DISCUSSION

6.1 Threats of validity

(1) We solely investigate the decline in performance of the top-performing learning to rank techniques when class labels are leveraged instead of bug numbers, but only on the module-level datasets. This is due to the published change-level and Android datasets do not encompass information regarding the count of bugs. Furthermore, all software projects are conceived by the open-source community, and it remains ambiguous whether our conclusions can be extrapolated to business projects. We will curtail this threat by scrutinizing additional modules from other defect repositories in the future work.

(2) We employ the LTR-linear, LTR-logistic, and DEJIT methods as approaches to build EADP models in cross-project and cross-release setups, since some research [54, 48] found that these algorithms performed optimally in their respective setups through a large-scale experiment. In particular, Yu et al. [54] compared 24 pointwise L2R algorithms, 4 pairwise algorithms, and 3 listwise algorithms. The results showed that LTR-linear showed the most excellent performance in the IFA indicator.

(3) We utilize the Precision@20%, Recall@20%, F1@20%, PofB@effort, PMI@effort, PofB/PMI@effort, IFA, and P_{opt} to evaluate our models in the effort-aware scenario [18, 7]. We employ PofB@effort, Recall@20%, and P_{opt} to evaluate whether the models meet the goal of EADP. Precision@20% is also used as it is often paired

with Recall@20%. A higher value of PMI@effort requires software testers to check more modules. Therefore, to correct any biases that can arise from using PofB@effort and PMI@effort, we employ PofB/PMI@effort. Additionally, previous study [18] present that high IFA value would significantly reduce the confidence of software testers, so we also use IFA measure. Furthermore, we use the non-parametric statistical Wilcoxon test to compare different models and to guarantee that any disparities observed are statistically meaningful.

6.2 Implications

We provide recommendations based on experiments for researchers in EADP studies. **Both class labels and bug numbers should be indicated when publishing software defect datasets.** Many publicly-available module-level defect datasets hold only class labels. Our experimental results show that using class labels to construct module-level EADP models causes a increase of the found bugs, but meanwhile it will lead to a significant increase of IFA and PMI@20%. The change-level datasets that are generated by SZZ approaches do not hold bug numbers either. However, many studies [21], [50], [49], [14], [25], [8], [16] considered the proportion of class labels of a change and the responding LOC as the actual defect density and then used it to construct EADP models. Since we cannot obtain the change-level datasets that hold bug number information, we do not investigate whether not using the bug numbers to construct change-level EADP models impacts the performance. But we still encourage researchers to extract the number of defects of changes and publish the datasets.

7 CONCLUSION

This paper explores the effects of disregarding bug numbers on the effectiveness of EADP models in software defect datasets. We examine the decline in the efficacy of the best learning to rank methods when class labels are utilized instead of bug numbers, and constructing an EADP model using LTR-linear, LTR-logistic, and DEJIT under both cross-project and cross-release setups. To assess the performance reduction of the best algorithms trained using the class labels, we compute the distribution of performance reduction for each evaluation measure (i.e., Precision@20%, Recall@20%, F1@20%, PofB@effort, PMI@effort, PofB/PMI@effort, IFA, and P_{opt}) using boxplots and employ the Wilcoxon test to present the statistical significance of the reduction. The experimental results indicate that incorporating the numbers of bug in building EADP models can lead to an decrease in the detected bugs. But it can lead to software testers inspecting fewer modules and more efficiently identifying the first buggy module in the majority of cases. Therefore, we suggest that not only the class labels but also the number of bugs should be indicated when publishing software defect datasets to construct more accurate EADP models.

ACKNOWLEDGMENTS

This work is partially supported by the National Key Research and Development Program (2022YFB3104001), the National Natural Science Foundation of China (61806151, 62202350), the Key Research and Development Program of Hubei Province (2022BAA050, 2020AAA001), the Key Research and Development Program of Hainan Province (ZDYF2021GXJS014), and the Natural Science Foundation of Chongqing (cstc2021jcyj-msxmX1146, cstc2021jcyj-msxmX1115).

REFERENCES

[1] Hervé Abdi. 2007. Bonferroni and Šidák corrections for multiple comparisons. *Encyclopedia of measurement and statistics* 3 (2007), 103–107.

[2] Carina Andersson and Per Runeson. 2007. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE transactions on software engineering* 33, 5 (2007), 273–286.

[3] Kwabena Ebo Bennin, Jacky Keung, Akito Monden, Yasutaka Kamei, and Naoyasu Ubayashi. 2016. 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 (COMPSAC)*, Vol. 1. IEEE, 154–163.

[4] Kwabena Ebo Bennin, Koji Toda, Yasutaka Kamei, Jacky Keung, Akito Monden, and Naoyasu Ubayashi. 2016. Empirical evaluation of cross-release effort-aware defect prediction models. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 214–221.

[5] G Boetticher. 2007. The PROMISE repository of empirical software engineering data. *http://promisedata. org/repository* (2007).

[6] Gemma Catolino, Dario Di Nucci, and Filomena Ferrucci. 2019. Cross-project just-in-time bug prediction for mobile apps: An empirical assessment. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 99–110.

[7] Ni Chao, Xin Xia, Lo David, Chen Xiang, and Gu Qing. 2022. Revisiting Supervised and Unsupervised Methods for Effort-Aware Cross-Project Defect Prediction. *IEEE Transactions on Software Engineering* 48, 3 (2022), 786–802.

[8] Xiang Chen, Yingquan Zhao, Qiuping Wang, and Zhidan Yuan. 2018. MULTI: Multi-objective effort-aware just-in-time software defect prediction. *Information and Software Technology* 93 (2018), 1–13.

[9] Tian Cheng, Kunsong Zhao, Song Sun, Muhammad Mateen, and Junhao Wen. 2022. Effort-aware cross-project just-in-time defect prediction framework for mobile apps. *Frontiers of Computer Science* 16, 6 (2022), 1–15.

[10] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An extensive comparison of bug prediction approaches. In *2010 7th IEEE working conference on mining software repositories*. IEEE, 31–41.

[11] Xin Du, Tian Wang, Liuhai Wang, Weifeng Pan, Chunlai Chai, Xinxin Xu, Bo Jiang, and Jiale Wang. 2022. CoreBug: improving effort-aware bug prediction in software systems using generalized k-core decomposition in class dependency networks. *Axioms* 11, 5 (2022), 205.

[12] Yuanrui FAN, Xin XIA, Daniel A COSTA, David LO, Ahmed E HASSAN, and Shanping LI. [n. d.]. The impact of changes mislabeled by SZZ on just-in-time defect prediction.(2019). *IEEE Transactions on Software Engineering* ([n. d.]), 1–26.

[13] Norman E. Fenton and Niclas Ohlsson. 2000. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software engineering* 26, 8 (2000), 797–814.

[14] Wei Fu and Tim Menzies. 2017. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 72–83.

[15] Lina Gong, Haoxiang Zhang, Jingxuan Zhang, Mingqiang Wei, and Zhiqiu Huang. 2022. A Comprehensive Investigation of the Impact of Class Overlap on Software Defect Prediction. *IEEE Transactions on Software Engineering* (2022).

[16] Yuchen Guo, Martin Shepperd, and Ning Li. 2018. Bridging effort-aware prediction and strong classification: a just-in-time software defect prediction study. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 325–326.

[17] Qiao Huang, Xin Xia, and David Lo. 2018. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. *Empirical Software Engineering* (2018), 1–40.

[18] Qiao Huang, Xin Xia, and David Lo. 2019. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. *Empirical Software Engineering* 24, 5 (2019), 2823–2862.

[19] Yue Jiang, Bojan Cukic, and Yan Ma. 2008. Techniques for evaluating fault prediction models. *Empirical Software Engineering* 13 (2008), 561–595.

[20] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Kenichi Matsumoto, Bram Adams, and Ahmed E Hassan. 2010. Revisiting common bug prediction findings using effort-aware models. In *2010 IEEE International Conference on Software Maintenance*. IEEE, 1–10.

[21] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773.

[22] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 165–176.

[23] Fuyang Li, Wanpeng Lu, Jacky Wai Keung, Xiao Yu, Lina Gong, and Juan Li. 2023. The impact of feature selection techniques on effort-aware defect prediction: An empirical study. *IET Software* 17, 2 (2023), 168–193.

[24] Fuyang Li, Peixin Yang, Jacky Wai Keung, Wenhua Hu, Haoyu Luo, and Xiao Yu. 2023. Revisiting 'revisiting supervised methods for effort-aware cross-project defect prediction'. *IET Software* (2023). https://doi.org/10.1049/sfw2.12133

[25] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. 2017. Code churn: A neglected metric in effort-aware just-in-time defect prediction. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 11–19.

[26] Wanwangying Ma, Lin Chen, Yibiao Yang, Yuming Zhou, and Baowen Xu. 2016. Empirical analysis of network measures for effort-aware fault-proneness prediction. *Information and Software Technology* 69 (2016), 50–70.

[27] Thilo Mende and Rainer Koschke. 2009. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*. 1–10.

[28] Thilo Mende and Rainer Koschke. 2010. Effort-aware defect prediction models. In *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 107–116.

[29] M Miletić, M Vukušić, G Mauša, and T Galinac Grbac. 2018. Cross-release code churn impact on effort-aware software defect prediction. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 1460–1466.

[30] K Muthukumaran, NL Bhanu Murthy, G Karthik Reddy, and Prateek Talishetti. 2016. Testing and Code Review Based Effort-Aware Bug Prediction Model. *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing* (2016), 17–30.

[31] Tung Thanh Nguyen, Tran Quang An, Vu Thanh Hai, and Tu Minh Phuong. 2014. Similarity-based and rank-based defect prediction. In *2014 International Conference on Advanced Technologies for Communications (ATC 2014)*. IEEE, 321–325.

[32] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. 2005. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering* 31, 4 (2005), 340–355.

[33] Annibale Panichella, Carol V Alexandru, Sebastiano Panichella, Alberto Bacchelli, and Harald C Gall. 2016. A search-based training algorithm for cost-aware defect prediction. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. 1077–1084.

[34] Lei Qiao and Yan Wang. 2019. Effort-aware and just-in-time defect prediction with neural network. *PloS one* 14, 2 (2019), e0211359.

[35] Yu Qu, Qinghua Zheng, Jianlei Chi, Yangxu Jin, Ancheng He, Di Cui, Hengshan Zhang, and Ting Liu. 2019. Using k-core decomposition on class dependency networks to improve bug prediction model's practical performance. *IEEE Transactions on Software Engineering* 47, 2 (2019), 348–366.

[36] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. 2013. Data quality: Some comments on the nasa software defect datasets. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1208–1215.

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

[38] Maria Ulan, Welf Löwe, Morgan Ericsson, and Anna Wingkvist. 2021. Weighted software metrics aggregation and its application to defect prediction. *Empirical Software Engineering* 26, 5 (2021), 1–34.

[39] Feng Wang, Jinxiao Huang, and Yutao Ma. 2018. A top-k learning to rank approach to cross-project software defect prediction. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 335–344.

[40] Frank Wilcoxon. [n. d.]. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 ([n. d.]), 80–83.

[41] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. 2011. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 15–25.

[42] Zhou Xu, Kunsong Zhao, Tao Zhang, Chunlei Fu, Meng Yan, Zhiwen Xie, Xiaohong Zhang, and Gemma Catolino. 2021. Effort-aware just-in-time bug prediction for mobile apps via cross-triplet deep feature embedding. *IEEE Transactions on Reliability* 71, 1 (2021), 204–220.

[43] Meng Yan, Yicheng Fang, David Lo, Xin Xia, and Xiaohong Zhang. 2017. File-level defect prediction: Unsupervised vs. supervised models. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 344–353.

[44] Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E Hassan, David Lo, and Shanping Li. 2020. Just-in-time defect identification and localization: A two-phase framework. *IEEE Transactions on Software Engineering* 48, 1 (2020), 82–101.

[45] Meng Yan, Xin Xia, Yuanrui Fan, David Lo, Ahmed E Hassan, and Xindong Zhang. 2020. Effort-aware just-in-time defect identification in practice: a case study at Alibaba. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1308–1319.

[46] Xiaoxing Yang, Ke Tang, and Xin Yao. 2014. A learning-to-rank approach to software defect prediction. *IEEE Transactions on Reliability* 64, 1 (2014), 234–246.

[47] Xiaoxing Yang and Wushao Wen. 2018. Ridge and lasso regression models for cross-version defect prediction. *IEEE Transactions on Reliability* 67, 3 (2018), 885–896.

[48] Xingguang Yang, Huiqun Yu, Guisheng Fan, and Kang Yang. 2021. DEJIT: a differential evolution algorithm for effort-aware just-in-time software defect prediction. *International Journal of Software Engineering and Knowledge Engineering* 31, 03 (2021), 289–310.

[49] Yibiao Yang, Mark Harman, Jens Krinke, Syed Islam, David Binkley, Yuming Zhou, and Baowen Xu. 2016. An empirical study on dependence clusters for effort-aware fault-proneness prediction. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 296–307.

[50] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. 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*. ACM, 157–168.

[51] Yibiao Yang, Yuming Zhou, Hongmin Lu, Lin Chen, Zhenyu Chen, Baowen Xu, Hareton Leung, and Zhenyu Zhang. 2014. Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction? An empirical study. *IEEE Transactions on Software Engineering* 41, 4 (2014), 331–357.

[52] Guoan You, Feng Wang, and Yutao Ma. 2016. An empirical study of ranking-oriented cross-project software defect prediction. *International Journal of Software Engineering and Knowledge Engineering* 26, 09n10 (2016), 1511–1538.

[53] Xiao Yu, Kwabena Ebo Bennin, Jin Liu, Jacky Wai Keung, Xiaofei Yin, and Zhou Xu. 2019. 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*. IEEE, 298–309.

[54] Xiao Yu, Heng Dai, Li Li, Xiaodong Gu, Jacky Wai Keung, Kwabena Ebo Bennin, Fuyang Li, and Jin Liu. 2023. Finding the best learning to rank algorithms for effort-aware defect prediction. *Information and Software Technology* (2023), 107165. https://doi.org/10.1016/j.infsof.2023.107165

[55] Xiao Yu, Jacky Keung, Yan Xiao, Shuo Feng, Fuyang Li, and Heng Dai. 2022. Predicting the precise number of software defects: Are we there yet? *Information and Software Technology* 146 (2022), 106847.

[56] Xiao Yu, Jin Liu, Jacky Wai Keung, Qing Li, Kwabena Ebo Bennin, Zhou Xu, Junping Wang, and Xiaohui Cui. 2019. Improving ranking-oriented defect prediction using a cost-sensitive ranking SVM. *IEEE Transactions on Reliability* 69, 1 (2019), 139–153.

[57] Xiao Yu, Jiqing Rao, Wenhua Hu, Jacky Keung, Junwei Zhou, and Jianwen Xiang. 2023. Improving effort-aware defect prediction by directly learning to rank software modules. *Information and Software Technology* (2023), 107250.

[58] Xiao Yu, Man Wu, Yiheng Jian, Kwabena Ebo Bennin, Mandi Fu, and Chuanxiang Ma. 2018. Cross-company defect prediction via semi-supervised clustering-based data filtering and MSTrA-based transfer learning. *Soft Computing* 22 (2018), 3461–3472.

[59] Tao Zhang, Jiachi Chen, Geunseok Yang, Byungjeong Lee, and Xiapu Luo. 2016. Towards more accurate severity prediction and fixer recommendation of software bugs. *Journal of Systems and Software* 117 (2016), 166–184.

[60] Wenzhou Zhang, Weiwei Li, and Xiuyi Jia. 2019. Effort-aware tri-training for semi-supervised just-in-time defect prediction. In *Advances in Knowledge Discovery and Data Mining: 23rd Pacific-Asia Conference, PAKDD 2019, Macau, China, April 14-17, 2019, Proceedings, Part II 23*. Springer, 293–304.

[61] Kunsong Zhao, Zhou Xu, Meng Yan, Lei Xue, Wei Li, and Gemma Catolino. 2022. A compositional model for effort-aware Just-In-Time defect prediction on android apps. *IET Software* 16, 3 (2022), 259–278.

[62] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE, 9–9.