Applied Soft Computing Journal 149 (2023) 110941

Contents lists available at ScienceDirect

Applied Soft Computing

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

A multi-objective effort-aware defect prediction approach based on NSGA-II

Xiao Yu [a,b], Liming Liu [a,c], Lin Zhu [d], Jacky Wai Keung [e], Zijian Wang [f], Fuyang Li [a,*]

[a] *School of Computer Science and Artificial Intelligence, Wuhan University of Technology, Wuhan, China*
[b] *Wuhan University of Technology Chongqing Research Institute, Chongqing, China*
[c] *School of Cyber Science and Engineering, Wuhan University, Wuhan, China*
[d] *School of Computer, Wuhan Qingchuan University, Wuhan, China*
[e] *Department of Computer Science, City University of Hong Kong, Hong Kong, China*
[f] *School of Science, Wuhan University of Technology, Wuhan, China*

ARTICLE INFO

Keywords:
Software defect prediction
Effort-aware
Defect density
Multi-objective optimization
Logistic regression

ABSTRACT

Effort-Aware Defect Prediction (EADP) technique sorts software modules by the defect density and aims to find more bugs when testing a certain number of Lines of Code (LOC). The existing EADP methods ignore the number of required inspected modules and thus resulting in more testing cost. Therefore, we propose a multi-objective effort-aware defect prediction approach based on NSGA-II named MOOAC for EADP, which aims to maximize the Proportion of the found Bugs (PofB@20%) and minimize the Proportion of Module Inspected (PMI@20%) when inspecting the top 20% LOC. MOOAC firstly trains a random forest classification model. Then, it builds a logistic regression model, and utilizes the NSGA-II algorithm to generate the coefficient vector of the model by maximizing the PofB@20% value and minimizing the PMI@20% value simultaneously. In the model prediction phase, MOOAC firstly employs the built random forest classifier to decide whether modules are defective. Next, the predicted defective modules are first inspected based on the ratio between the predicted defect probability by the logistic regression model and LOC, which can make testers to find more bugs and test as fewer LOC as possible. The clean modules are then inspected to reduce the Initial False Alarms (IFA), if there is still the testing budget left. The results show that MOOAC exhibits the best overall performance on the PofB@20% and PMI@20%. In other words, MOOAC enables testers to identify more bugs per 1% module.

1. Introduction

With software systems scaling in size and complexity, it poses a challenge to release high-quality software within a limited testing period and resource allocation [1–3]. The Software Defect Prediction (SDP) technique can help software quality assurance teams find software modules that may contain defects more quickly [4–6] and assist fault localization [7–9]. Specifically, the SDP technique firstly extracts some software features of historical software modules, e.g., Lines Of Code (LOC) and code complexity. Then, it uses these features to build a model and predicts the defect-proneness for new ones. Accurate predictions can guide software testers place more emphasis on those predicted defective software modules, and guide the allocation of limited testing resources [10–13].

The existing methods for SDP can be primarily classified into two categories: Classification-Based Defect Prediction (CBDP) and Effort-Aware Defect Prediction (EADP). The CBDP technique treats the prediction problem as a binary classification task and predicts the class label of a new module (i.e., defective or clean). Therefore, software testers may allocate equal testing resources to all predicted defective modules. However, when the testing resources are not enough to inspect all predicted defective modules, the CBDP technique cannot provide the inspection priority. The EADP technique sorts modules by the defect density, and provides guidance to software testers, directing their attention towards prioritizing the inspection of software modules exhibiting a higher density of defects. Consequently, EADP models aim to detect more defects when testing a certain amount of LOC [14–16]. As an illustration, we provide a simple example to explain the benefit of EADP compared with CBDP in Fig. 1.

Example 1. Suppose software testers need to test a newly developed software system of 100 software modules (i.e., $\boldsymbol{M}_1$, $\boldsymbol{M}_2$, $\boldsymbol{M}_3$, ..., $\boldsymbol{M}_{100}$) that contain 100,000 LOC totally. Due to the tight schedule, the testers are only able to test a part of the code (e.g., 20% LOC of the entire system). Therefore, they can build either a CBDP model

* Corresponding author.
E-mail addresses: xiaoyu@whut.edu.cn (X. Yu), liming.liu@whu.edu.cn (L. Liu), linzhu_cs@126.com (L. Zhu), jacky.keung@cityu.edu.hk (J.W. Keung), zijian.w@hotmail.com (Z. Wang), fyli@whut.edu.cn (F. Li).

https://doi.org/10.1016/j.asoc.2023.110941
Received 13 June 2023; Received in revised form 4 October 2023; Accepted 8 October 2023
Available online 17 October 2023
1568-4946/© 2023 Elsevier B.V. All rights reserved.

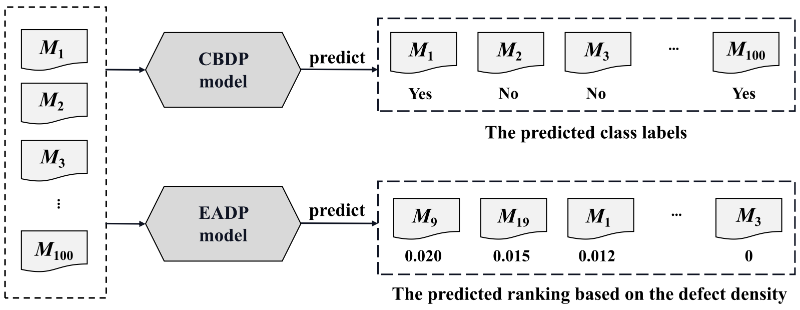


Fig. 1. The difference of the CBDP model and EADP model.

or an EADP model based on historical software data to help them allocate the limited testing resources. Suppose the CBDP model predicts that 30 modules with 25,000 LOC are defective. They cannot test all the predicted defective modules and should determine which ones to inspect first. However, they can test first several modules (i.e., $\boldsymbol{M}_9$, $\boldsymbol{M}_{19}$, $\boldsymbol{M}_1$, …, and so on) in descending order of the density provided by the EADP model, until 20,000 LOC are inspected. Thus, EADP can assist in allocating the limited testing resources more efficiently.

1.1. Motivation

Menzies et al. [17] proposed the ManualUp method that sorted modules with fewer LOC first. Subsequently, Huang et al. [18] argued that more bugs could be found according to the recommended ranking of ManualUp, but the precision value was low and the Initial False Alarms (IFA) value was very high when testing the top-ranked modules. Consequently, Huang et al. [18] proposed CBS+, which suggested to first test those predicted defective modules based on the defect density (i.e., the ratio between the predicted defect probability by the classification model and LOC), and then test the clean ones. Recently, Ni et al. [19,20] showed the superiority of CBS+ for cross-project EADP and just-in-time EADP on JavaScript projects, respectively. However, as software modules with more LOC have a higher probability of being defective, CBS+ tends to rank modules with more LOC at the top of the predicted ranking. Therefore, the Proportion of Module Inspected (PMI@20%) and IFA values are low, but the Proportion of the found Bugs (PofB@20%) value suffers a corresponding decrease while checking the top 20% LOC.

Example 2. Assume there are the 15 software modules in a given test dataset, i.e., $\boldsymbol{M}_1, \boldsymbol{M}_2, \boldsymbol{M}_3, \ldots, \boldsymbol{M}_{15}$. Suppose $\boldsymbol{M}_1$, $\boldsymbol{M}_2$, $\boldsymbol{M}_3$, $\boldsymbol{M}_4$, $\boldsymbol{M}_5$, $\boldsymbol{M}_6$, and $\boldsymbol{M}_7$ are actually defective modules with 4, 2, 3, 2, 2, 2 and 2 bugs respectively, and the remaining modules are actually clean. In addition, the total LOC of these 15 software modules is 3000, among which the LOC of $\boldsymbol{M}_1$, $\boldsymbol{M}_2$, $\boldsymbol{M}_3$, $\boldsymbol{M}_4$, $\boldsymbol{M}_5$, $\boldsymbol{M}_6$, $\boldsymbol{M}_7$, and $\boldsymbol{M}_8$ are 400, 600, 300, 200, 100, 300, 200, and 200 respectively, and the LOC of the rest of the software modules are 100. Due to limited testing resources, software testers can only test the top 20% LOC (i.e., 600 LOC). Assume there are the 5 rankings of the software modules:

Ranking 1: $\boldsymbol{M}_4$, $\boldsymbol{M}_1$, $\boldsymbol{M}_2$, $\boldsymbol{M}_3$, $\boldsymbol{M}_5$, $\boldsymbol{M}_6$, $\boldsymbol{M}_7$, $\boldsymbol{M}_8$, $\boldsymbol{M}_9$, $\boldsymbol{M}_{10}$, $\boldsymbol{M}_{11}$, $\boldsymbol{M}_{12}$, $\boldsymbol{M}_{13}$, $\boldsymbol{M}_{14}$, $\boldsymbol{M}_{15}$

Ranking 2: $\boldsymbol{M}_4$, $\boldsymbol{M}_5$, $\boldsymbol{M}_6$, $\boldsymbol{M}_1$, $\boldsymbol{M}_2$, $\boldsymbol{M}_3$, $\boldsymbol{M}_7$, $\boldsymbol{M}_8$, $\boldsymbol{M}_9$, $\boldsymbol{M}_{10}$, $\boldsymbol{M}_{11}$, $\boldsymbol{M}_{12}$, $\boldsymbol{M}_{13}$, $\boldsymbol{M}_{14}$, $\boldsymbol{M}_{15}$

Ranking 3: $\boldsymbol{M}_6$, $\boldsymbol{M}_3$, $\boldsymbol{M}_1$, $\boldsymbol{M}_2$, $\boldsymbol{M}_4$, $\boldsymbol{M}_5$, $\boldsymbol{M}_7$, $\boldsymbol{M}_8$, $\boldsymbol{M}_9$, $\boldsymbol{M}_{10}$, $\boldsymbol{M}_{11}$, $\boldsymbol{M}_{12}$, $\boldsymbol{M}_{13}$, $\boldsymbol{M}_{14}$, $\boldsymbol{M}_{15}$

Ranking 4: $\boldsymbol{M}_2$, $\boldsymbol{M}_1$, $\boldsymbol{M}_3$, $\boldsymbol{M}_4$, $\boldsymbol{M}_5$, $\boldsymbol{M}_6$, $\boldsymbol{M}_7$, $\boldsymbol{M}_8$, $\boldsymbol{M}_9$, $\boldsymbol{M}_{10}$, $\boldsymbol{M}_{11}$, $\boldsymbol{M}_{12}$, $\boldsymbol{M}_{13}$, $\boldsymbol{M}_{14}$, $\boldsymbol{M}_{15}$

Ranking 5: $\boldsymbol{M}_9$, $\boldsymbol{M}_{10}$, $\boldsymbol{M}_{11}$, $\boldsymbol{M}_{12}$, $\boldsymbol{M}_{13}$, $\boldsymbol{M}_5$, $\boldsymbol{M}_{14}$, $\boldsymbol{M}_{15}$, $\boldsymbol{M}_4$, $\boldsymbol{M}_7$, $\boldsymbol{M}_8$, $\boldsymbol{M}_6$, $\boldsymbol{M}_3$, $\boldsymbol{M}_1$, $\boldsymbol{M}_2$

Fig. 2 illustrates the detection status for each ranking, where the blue box indicates the modules that need to be detected when testing top 20% LOC with the corresponding ranking order, and the number below each module denotes its number of defects.

ManualUp may produce the Ranking 5, which sorts these modules in the ascending order of LOC. Therefore, according to the Ranking 5, software testers require to detect 6 modules (PMI@20% = 6/15) and can find 2 bugs (PofB@20% = 2/17), and the IFA value is high (=5). CBS+ may produce the Ranking 4, which tends to sort these modules with more LOC in the front of the ranking. Therefore, according to the Ranking 4, software testers require to test 1 software module and can find 2 bugs. According to the Ranking 1 and Ranking 3, software testers need to test the same number of modules (=2) but can detect more bugs based on the Ranking 1. According to the Ranking 1 and Ranking 2, software testers can detect the same number of bugs (=6) but need to inspect fewer modules based on the Ranking 1. In a recent study by Huang et al. [18], an investigation was conducted to assess the influence of detecting an excessive number of changes or modules. The participants in the study consisted of professional developers with experience in code review, and a total of 54 replies were collected. The findings revealed that frequent context switches between changes or modules imposed additional effort costs associated with inspecting a wider range of affected files, managing conflicts or dependencies, and localizing bugs. In other words, using the EADP model to detect a higher number of modules requires additional effort and yields inferiors results. Therefore, Ranking 1 is better than Ranking 2. In summary, Ranking 1 is the best ranking of software modules among the five ones, which can guide testers to detect more bugs in fewer software modules while checking the top 20% LOC.

Therefore, **the main objective of EADP is finding more bugs and inspecting as fewer modules as possible, when testing a specific number of lines of code.**

However, current methods treat EADP as either a simple classification or regression problem. For example, CBS+ [18], EATT [21], CBS+(DF) [22], and EASC [19,20] consider EADP as a binary classification task, distinguishing between defective and clean modules, and then combine the classification probabilities with LOC to calculate defect density. The optimization objective is to maximize the accuracy of the classifier. Conversely, EALR [23] considers EADP as a regression task, where it employs a linear regression model to compute the defect density for modules. In this scenario, the main optimization goal shifts towards minimizing the Mean Squared Error (MSE) of the regression model. They do not take into account the number of modules that need to be inspected.

1.2. Our work and contributions

Therefore, we propose a multi-objective effort-aware defect prediction approach called MOOAC (Multi-Objective Optimization After Classification). In the model construction phase, MOOAC firstly trains a random forest classification model based on all training modules. Then, it builds a logistic regression model based on all defective modules and

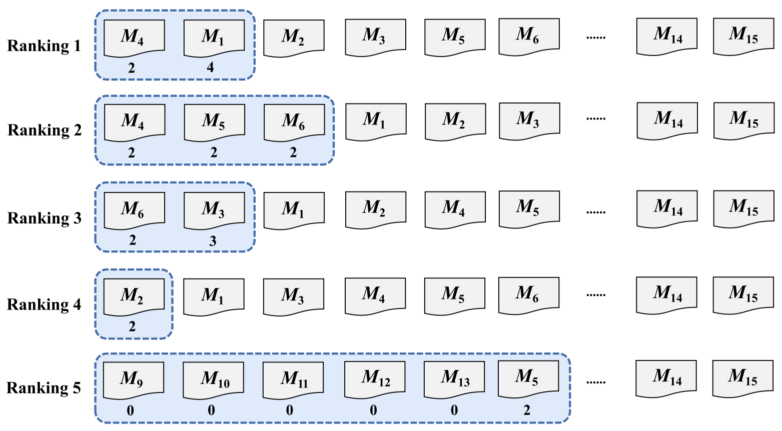


Fig. 2. The explanation of Example 2.

utilizes the multi-objective optimization algorithm (i.e., Non-dominated Sorting Genetic Algorithm-II, NSGA-II [24]) to generate a set of Pareto optimal solutions by maximizing the PofB@20% value and minimizing the PMI@20% value simultaneously while testing a certain number of LOC. Each solution represents a coefficient vector for the model. Finally, MOOAC chooses the coefficient vector that achieves the highest PofB/PMI@20% value among the 10 highest PofB@20% coefficient vectors on the training data as the coefficient vector for the logistic regression model. In the model prediction phase, MOOAC firstly uses the built random forest model to predict the defect-proneness. Then, it ranks those predicted defective modules based on the ratio between the predicted defect probability by the built logistic regression model and LOC, which can help testers inspect fewer modules and find more bugs. Next, it sorts those predicted clean modules based on the ratio between the predicted defect probability by the built random forest model and LOC. Finally, MOOAC incorporates a strategy wherein ranked clean modules are appended to the tail of the ranked defective modules. This approach guides testers to prioritize the inspection of predicted defective modules, aiming to minimize IFA value during the testing process.

Based on the aforementioned method, our research process and experiment can be condensed into the following five Research Questions (RQs):

RQ1: Does MOOAC outperform the state-of-the-art EADP methods?

In order to validate whether MOOAC can detect more bugs and test fewer modules simultaneously, we evaluate MOOAC against the seven existing EADP methods, i.e., CBS+ [18], ManualUp [25], EALR [23], EATT [21], CBS+(DF) [22], EASC [19,20] and NSGA-II [24], which mainly focus on finding more bugs within a certain amount of LOC, and use 11 software projects with 41 releases from the PROMISE repository [26].

RQ2: Does the data imbalance problem affect the performance of MOOAC?

The previous studies [11,27–32] showed that the class imbalance problem degraded the performance of CBDP models, and Tantithamthavorn et al. [33] found Random Under Sampling (RUS) and an auto-tuning parameter version of SMOTE (SMOTEDE) can perform better than four data re-sampling methods on 101 defect datasets. Since SMOTEDE needs to choose an evaluation metric as the optimization objective, but we employ several evaluation metrics to assess EADP models comprehensively. If we solely optimize PofB@20%, it may result in an increase in the PMI@20% and IFA values of the models. As a result, we focus solely on investigating whether RUS could improve the performance of EADP models.

RQ3: Does the underlying classifier of MOOAC affect the performance of MOOAC?

We use random forest as the underlying classifier of MOOAC to distinguish modules into defective and clean ones. The previous studies [34–36] pointed out that different classification algorithms affect the performance of CBDP models. If we select a different classification algorithm embedded in MOOAC, the performance of MOOAC will also be different. Therefore, we explore the effect of the underlying classifier on the performance of MOOAC.

RQ4: Does the defective threshold to distinguish the defective and clean modules affect the performance of MOOAC?

In MOOAC, the testing dataset is partitioned into two sets (i.e, predicted defective set and clean set), with the latter appended after the former in the final ranking. The defective threshold in the random forest model directly affects the composition of above two datasets. A high threshold value may lead to a small number of modules being predicted as defective, and the defect probability predicted by the logistic regression model cannot accurately represent the detection priority between truly defective modules and those falsely predicted as clean due to the high threshold value. Conversely, a low threshold value may result in a large number of truly clean modules with low LOC being ranked at the forefront of the ranking. Therefore, to assess the defective threshold's impact, we compare MOOAC's performance using different thresholds to distinguish between defective and clean modules.

RQ5: Does the model training strategy affect the performance of MOOAC?

On a more detailed level, numerous training strategy factors might influence MOOAC's performance. For example, the type of model employed to compute defect density in defective modules (linear regression or logistic regression), the composition of the training dataset (solely containing truly defective modules or all modules), and the parameter selection approach for the model, all have the potential to impact performance. Therefore, we conduct separate comparative analyses regarding these three aspects of MOOAC's performance to attain the best possible results.

The result shows that (1) Although ManualUp, EATT, and NSGA-II can find more bugs than MOOAC, they produce lots of initial false alarms and are not accepted by software testers [37,38]; (2) MOOAC has an average improvement of CBS+ by 12.82%, EALR by 6.21% and EASC by 31.06% in terms of PofB@20%. Moreover, MOOAC achieves a substantial reduction in the average PMI@20% value compared to the baseline methods except for EASC, ranging from 22.45% to 72.34%. (3) MOOAC exhibits the best overall performance on the PofB@20% and PMI@20%. In other words, MOOAC enables testers to identify more bugs per 1% module.

The contributions of this work are summarized as follows:

• We argue that the main goal of EADP should be not only finding more bugs but also inspecting as fewer modules as possible, while testing a certain number of LOC.

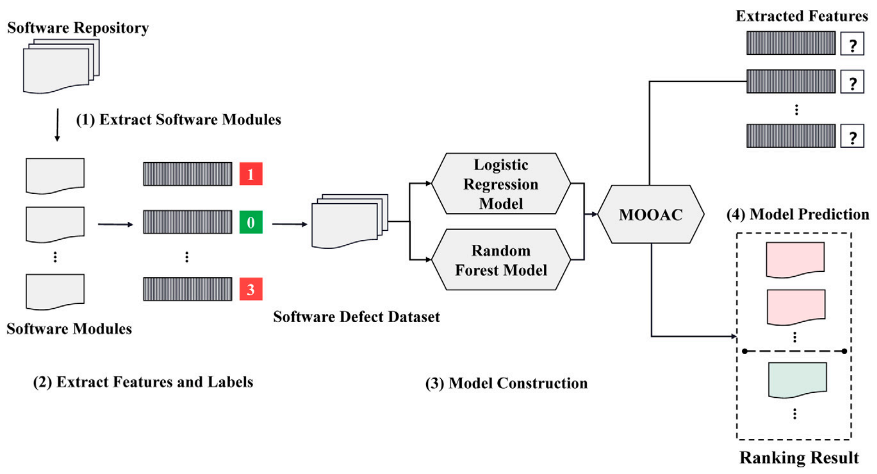


Fig. 3. The process of the MOOAC model.

• We propose a multi-objective effort-aware defect prediction approach based on NSGA-II for EADP, which aims to maximize the found bugs and minimize the required inspected modules simultaneously when testing a certain amount of LOC. To the best of our knowledge, this is the first attempt to introduce multi-objective optimization to address the problem that the existing EADP methods ignore the amount of required inspected modules.

1.3. Organization

The remainder of this paper is organized as follows. The proposed MOOAC method is detailed in Section 2. Sections 3 and 4 introduce the experiment setup and results. Section 5 introduces the threats to validity. The related works about EADP and multi-objective optimization for CBDP are discussed in Section 6. Finally, we address the conclusion in Section 7.

2. Our approach

2.1. Overview

The MOOAC process contains the following four phases as illustrated in Fig. 3. In the first phase, software modules are extracted from the historical software repository. Subsequently, the software features x_i and defect number y_i are extracted in the second phase. Consequently, a software module M_i can be represented as (x_i, y_i). The whole training dataset containing n modules could be denoted as $D = (M_1, M_2, \ldots, M_n)$. Thirdly, the process of model construction is divided into two parts. The first part involves the construction of a random forest model, where the model is trained on D to distinguish between defective and clean modules. The second part involves the construction of a logistic regression model, which is exclusively trained on the defective dataset. The NSGA-II [24] is employed to optimize the model parameters, providing the ranking results for defective modules. The integration of these two models constitutes the MOOAC model. In the last phase, we employ the trained MOOAC model to predict the defect densities of the new software modules based on the extracted features, then sort the modules according to the predicted values. The specific process of model prediction is elaborately presented in Algorithm 2.

2.2. Model construction

In the model construction phase, MOOAC firstly builds a random forest classifier based on the training dataset S containing all software modules to distinguish the testing modules into defective and clean ones. To achieve this binary classification, we adjust the labels of the training set. In particular, those modules that have a number of defects greater than 0 are labeled as defective, whereas those with a number of defects equal to 0 are labeled as clean.

Chen et al. [39] and Huang et al. [18,40] have highlighted that the relationship between defect density and software features may not be linear. Therefore, MOOAC then builds a logistic regression model:

$$y = f(\boldsymbol{w}, \boldsymbol{x}) = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + \cdots + w_d x_d)}}, \quad (1)$$

where $\boldsymbol{x} = (x_1, x_2, \ldots, x_d)$ represents the software feature vector of a software module, $\boldsymbol{w} = (w_1, w_2, \ldots, w_d)$ is the coefficient vector of the logistic regression model, and y denotes the predicted defect probability of the module. Training the model is to learn from the training dataset to obtain the coefficient vector $\boldsymbol{w}$. Once $\boldsymbol{w}$ is fixed, the logistic regression model is learned. In our study, we wish to maximize the PofB@20% and minimize the PMI@20% simultaneously when testing the top 20% LOC. To solve the above-mentioned optimization problem, we use the genetic algorithm to estimate the coefficient vector, more specifically, a multi-objective optimization algorithm (i.e., NSGA-II [24]). We choose the genetic algorithm because it constitutes a widely recognized search technique employed to ascertain precise or approximate solutions for optimization problems and is commonly adopted by researchers in the realm of software engineering and artificial intelligence [41,42]. This algorithm transforms the solution within a search domain into a chromosome format, facilitating the search for optimal solutions. To elaborate further, due to NSGA-II's proven effectiveness in multi-objective optimization, we have chosen the NSGA-II algorithm to achieve our desired objectives for the MOOAC.

Definition 1 (*Pareto Dominance [24]*)**.** Assuming that $\boldsymbol{w}_i$ and $\boldsymbol{w}_j$ are two feasible coefficient vectors, and we call $\boldsymbol{w}_i$ dominates $\boldsymbol{w}_j$ if and only if one of the following criteria is satisfied:

(1) PofB($\boldsymbol{w}_i$) > PofB($\boldsymbol{w}_j$) and PMI($\boldsymbol{w}_i$) ≤ PMI($\boldsymbol{w}_j$),

(2) PofB($\boldsymbol{w}_i$) ≥ PofB($\boldsymbol{w}_j$) and PMI($\boldsymbol{w}_i$) < PMI($\boldsymbol{w}_j$),

where PofB($\boldsymbol{w}_i$) is the PofB value on the given dataset according to the prediction result of the logistic regression model with $\boldsymbol{w}_i$ as the coefficient vector, and PMI($\boldsymbol{w}_i$) is the PMI value on the given dataset according to the prediction result with $\boldsymbol{w}_i$ as the coefficient vector.

Definition 2 (*Pareto Optimal Solution [24]*). A feasible coefficient vector $\boldsymbol{w}$ is a Pareto optimal solution, if and only if no other feasible coefficient vector dominates $\boldsymbol{w}$.

Definition 3 (*Pareto Optimal Set [24]*)**.** The set consisting of all Pareto optimal solutions is called the Pareto optimal set.

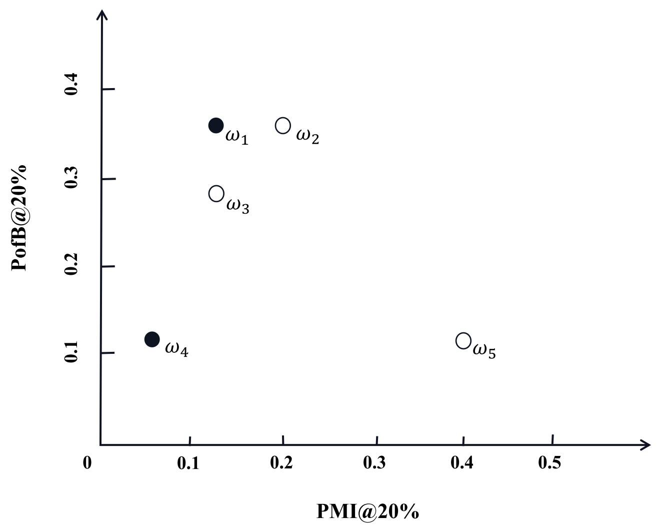


Fig. 4. The PofB@20% and PMI@20% values of the five feasible solutions.

Example 3. Suppose that $\boldsymbol{w}_1$, $\boldsymbol{w}_2$, $\boldsymbol{w}_3$, $\boldsymbol{w}_4$, and $\boldsymbol{w}_5$ are five feasible coefficient vectors of the logistic regression model, and the corresponding five models with the five vectors predict the ranking of the fifteen software modules in Example 2 as Ranking 1, Ranking 2, Ranking 3, Ranking 4, and Ranking 5 as shown in Example 2. Suppose that software testers require to inspect the top 20% LOC, and Fig. 4 shows the PofB@20% and PMI@20% values of the five feasible solutions. According to Definition 1, $\boldsymbol{w}_1$ dominates $\boldsymbol{w}_2$ and $\boldsymbol{w}_3$, and $\boldsymbol{w}_4$ dominates $\boldsymbol{w}_5$. In addition, $\boldsymbol{w}_1$, $\boldsymbol{w}_2$, and $\boldsymbol{w}_3$ dominate $\boldsymbol{w}_5$, while $\boldsymbol{w}_1$ does not dominate $\boldsymbol{w}_4$. In this example, both $\boldsymbol{w}_1$ and $\boldsymbol{w}_4$ are both Pareto optimal solutions, and these two coefficient vectors form the Pareto optimal set together.

In general, the NSGA-II algorithm returns a Pareto optimal set containing multiple Pareto optimal solutions. Therefore, MOOAC needs to select one Pareto optimal solution to build the logistic regression model. Since the primary objective of EADP is to find more bugs, we first choose ten coefficient vectors from the Pareto optimal set that achieve the highest PofB@20% value according to the ranking results predicted by the logistic regression model for inspecting more bugs. In addition, we hope software testers can inspect as fewer modules as possible, so we finally choose the coefficient vector that achieves the highest PofB/PMI@20% value among the 10 coefficient vectors.

Example 4. The Pareto optimal set includes $\boldsymbol{w}_1$ and $\boldsymbol{w}_4$ in Example 3. Because PofB/PMI@20%($\boldsymbol{w}_1$) > PofB/PMI@20%($\boldsymbol{w}_4$), we select $\boldsymbol{w}_1$ as the final coefficient vector.

Algorithm 1 describes the process of using NSGA-II to determine the coefficient vector for a logistic regression model using defective modules as the training set S_d, since the model is used in the prediction stage to predict the defect probability for only predicted defective modules. The algorithm begins by randomly generating an initial population of p solutions, each of which is a feasible coefficient vector (Line 1). Then, the population evolution process generate an offspring population, denoted as P_1 (Line 2). This is followed by a series of iterations, where the current population undergoes selection, crossover, and mutation operations at each iteration to generate a new generation of population. Specifically, the selection operation combines the previous and newly generated populations and selects a subset of p solutions to form the new population (Line 5). The crossover operation is then performed on the selected solutions with a probability of p_c, which results in the modification of the genes of the solutions. The mutation operation randomly changes the genes of the solutions with a small probability, p_m (Line 7). The algorithm ultimately returns the coefficient vector $\boldsymbol{w}$, which achieves the highest PofB/PMI@20% among the 10 highest PofB@20% coefficient vectors to construct the logistic regression model (Lines 9–10). The parameters are as follows: p = 200, t_{max} = 400, coefficients $\boldsymbol{w} \in [-10, 10]$, p_c = 0.7, and p_m = 0.053. Preliminary experimentation suggests that augmenting the values of p and t_{max} does not notably enhance the effectiveness of the MOOAC. Similarly, modifications to the crossover and mutation operator probabilities have been found to have little to no impact on the performance of MOOAC. Therefore, the default settings for these operators are utilized in the experimental study.

Algorithm 1 Estimation of $\boldsymbol{w}$

```
Input: All defective modules, $S_d$
        Number of coefficient vectors in a population, $p$
        Number of maximal generation, $t_{max}$
        Probability of crossover operator, $p_c$
        Probability of mutation operator, $p_m$
Output: $\boldsymbol{w}$
1: $P_0 = p$ generated solutions randomly;
2: Create an offspring population $P_1$ using $P_0$;
3: $t = 2$;
4: while $t < t_{max}$ do
5:     Select $p$ solutions $P_t$ from $P_{t-1}$ and $P_{t-2}$;
6:     $P_t$ = crossover($P_t$) with the probability $p_c$;
7:     $P_t$ = mutation($P_t$) with the probability $p_m$ ;
8:     $t = t + 1$;
9: Select the solution $\boldsymbol{w}$ which achieves the highest
PofB/PMI@20% among the 10 highest PofB@20%
coefficient vectors in $P_t$;
10: return $\boldsymbol{w}$;
```

2.3. Model prediction

MOOAC firstly uses the built random forest model to predict the probability of each software module $\boldsymbol{M}_j$ in the testing dataset (i.e., $Prob\_RF(\boldsymbol{M}_j)$), and divides all software modules into two sets. The *Defective_Set* contains the predicted defective modules whose defect probabilities are larger or equal to 0.5 (e.g., $\boldsymbol{M}_{19}$, $\boldsymbol{M}_{31}$, …, $\boldsymbol{M}_2$), while the *Clean_Set* contains the predicted clean ones (e.g., $\boldsymbol{M}_{92}$, $\boldsymbol{M}_{78}$, …, $\boldsymbol{M}_3$). Then, MOOAC employs the different ranking strategies on the two sets as shown in Fig. 5.

We prioritize predicted defective modules in the $Defective\_Set$ by placing them at the forefront of the ranking. Depending solely on the defect probability $Prob\_RF$ generated by the random forest model to determine the detection priority for the defective modules falls short in meeting the objectives of maximizing the number of detected bugs and minimizing the number of detected modules. To address this, we leverage the constructed logistic regression model to provide a more effective detection priority for defective modules. It sorts each module $\boldsymbol{M}_j$ in the *Defective_Set* based on the ratio between the predicted defect probability by the built logistic regression model (i.e., $Prob\_LR(\boldsymbol{M}_j)$) and its LOC, which can guide testers find more bugs and inspect as fewer modules as possible.

If any budget remains after inspecting all predicted defective modules in $Defective\_Set$, software testers may need to inspect a few predicted clean modules at the beginning of $Clean\_Set$. However, as the logistic regression model is only trained on truly defective data, it cannot prioritize clean modules. Similar to CBS+ [18], we rank each module $\boldsymbol{M}_j$ in the $Clean\_Set$ based on the ratio between $Prob\_RF(\boldsymbol{M}_j)$ and its LOC. Finally, the software modules in the *Clean_Set* is appended at the end of the modules in the *Defective_Set*. In this way, the predicted defective modules will be inspected first to reduce the IFA value.

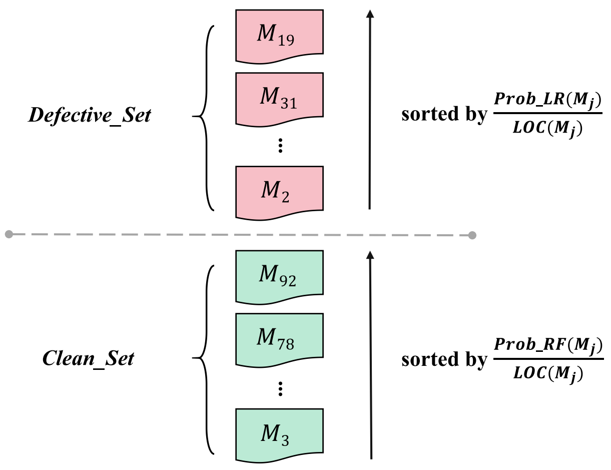

Defective_Set
M_{19}
M_{31}
M_2
sorted by $\frac{Prob\_LR(M_j)}{LOC(M_j)}$
Clean_Set
M_{92}
M_{78}
M_3
sorted by $\frac{Prob\_RF(M_j)}{LOC(M_j)}$

Fig. 5. The ranking strategy of MOOAC.

Algorithm 2 The prediction process of MOOAC

```
Input: Built random forest model, RF
       Built logistic regression model, LR
       Testing dataset, $S_{test}$
Output: Module ranking, RankList
1: $LOCList = extract(S_{test})$;
2: for module $M_j$ in $S_{test}$:
3:     $Prob\_RF(M_j) = RF(M_j)$;
4: $DefectiveSet, CleanSet = separate(S_{test}, Prob\_RF)$;
5: for module $M_j$ in $Defective\_Set$:
6:     $Prob\_LR(M_j) = LR(M_j)$;
7: $DefectiveRank = rank(Prob\_LR, LOCList)$;
8: $CleanRank = rank(Prob\_RF, LOClist)$
9: $RankList.append(DefectiveRank, CleanRank)$;
10: return $RankList$;
```

Algorithm 2 describes the prediction process of MOOAC. It begins by extracting the LOC of each module from the testing dataset (S_{test}) (Line 1). Then, to distinguish between defective and clean modules, all modules in S_{test} are inputted into the built random forest model *RF* to calculate their corresponding defect probability $Prob\_RF$. Subsequently, S_{test} is split into *Defective_Set* and *Clean_Set* according to the $Prob\_RF$ (Lines 2–4). For the predicted defective modules in *Defective_Set*, the built logistic regression model *LR* is employed to calculate their defect probability $Prob\_LR$ and predict the detection priority (Lines 5–6). Finally, MOOAC sorts the predicted defective modules based on the ratio between $Prob\_LR$ and their corresponding LOC, while the predicted clean modules are sorted based on the ratio between $Prob\_RF$ and their corresponding LOC. To ensure the predicted defective modules are detected first, the modules in the *Clean_Set* is appended at the end of the modules in the *Defective_Set* (Lines 7–10).

3. Experimental setup

3.1. Datasets

To align with the goal of EADP models to detect more defects, we chose defect datasets that include the number of defects. Furthermore, to conduct the cross-version validation that is more applicable, we specifically choose 41 versions of 11 publicly available projects from PROMISE. Table 1 presents the details of the datasets, including the number of modules (#Module), the percentage of defective modules (%Defects), the average number of defects per module (AvgDefects), and the average number of lines of code per module (AvgLOC). Each module in the datasets is represented by a 20-dimensional feature vector. For more information on the 20 software features, please refer to [26].

Table 1
The details of the experimental datasets.

Datasets	#Module	%Defects	AvgDefects	AvgLOC
Ant-1.3	125	16%	1.65	301.6
Ant-1.4	178	22.5%	1.18	304.5
Ant-1.5	293	10.9%	1.09	297.1
Ant-1.6	351	26.2%	2.00	322.6
Ant-1.7	745	22.3%	2.04	280.1
Camel-1.0	339	3.8%	1.08	99.5
Camel-1.2	608	35.5%	2.42	109.0
Camel-1.4	872	16.6%	2.31	112.5
Camel-1.6	965	19.5%	2.66	117.2
Ivy-1.1	111	56.8%	3.7	245.9
Ivy-1.4	241	6.6%	1.12	246
Ivy-2.0	352	11.4%	1.4	249.3
Jedit-3.2	272	33.1%	4.24	473.8
Jedit-4.0	306	24.5%	3.01	473.2
Jedit-4.1	312	25.3%	2.75	490.7
Jedit-4.2	367	13.1%	2.21	465.1
Jedit-4.3	492	2.2%	1.09	411.3
Log4j-1.0	135	25.2%	1.79	159.6
Log4j-1.1	109	33.9%	2.32	182.9
Log4j-1.2	205	92.2%	2.63	186.3
Lucene-2.0	195	46.7%	2.95	259.5
Lucene-2.2	247	58.3%	2.88	257.4
Lucene-2.4	340	59.7%	3.11	302.5
Poi-1.5	237	59.5%	2.43	233.9
Poi-2.0	314	11.8%	1.05	296.7
Poi-2.5	385	64.4%	2.0	311.0
Poi-3.0	442	63.6%	1.78	292.6
Synapse-1.0	157	10.2%	1.31	183.5
Synapse-1.1	222	27%	1.65	190.5
Synapse-1.2	256	33.6%	1.69	209.0
Velocity-1.4	196	75%	1.43	263.8
Velocity-1.5	214	66.4%	2.33	248.3
Velocity-1.6	229	34.1%	2.44	249.0
Xalan-2.4	723	15.2%	1.42	311.3
Xalan-2.5	803	48.2%	1.37	379.7
Xalan-2.6	885	46.4%	1.52	465.2
Xalan-2.7	909	98.8%	1.35	471.5
Xerces-init	162	47.5%	2.17	560.0
Xerces-1.2	440	16.1%	1.62	361.9
Xerces-1.3	453	15.2%	2.8	368.9
Xerces-1.4	588	74.3%	3.65	240.1

3.2. Evaluation metrics

To conduct a comprehensive assessment of EADP methods' performance, we utilize the seven effort-aware evaluation metrics, which are also widely used in the fields of artificial intelligence [43–47] and software engineering [48–53]. Referring to these existing EADP studies [18,19,21,23,52,54,55], we set 20% of the total number of LOC as the testing effort. Suppose that the total number of modules in a defect dataset is M, the total number of defects is Q, and the total number of modules with bugs is P. When testing the top 20% LOC based on the ranking result of an EADP model, there are m software modules, and p modules are actually defective and have q bugs among the m ones.

(1) **PofB@20%** is the percentage of bugs in the top 20% LOC to all bugs in the dataset:

$$PofB@20\% = \frac{q}{Q}. \tag{2}$$

(2) **PMI@20%** is the percentage of modules in the top 20% LOC to all modules in the dataset:

$$PMI@20\% = \frac{m}{M}. \tag{3}$$

(3) **PofB/PMI@20%** is the ratio between PofB@20% and PMI@20%, and a higher value indicates testers can find more bugs by

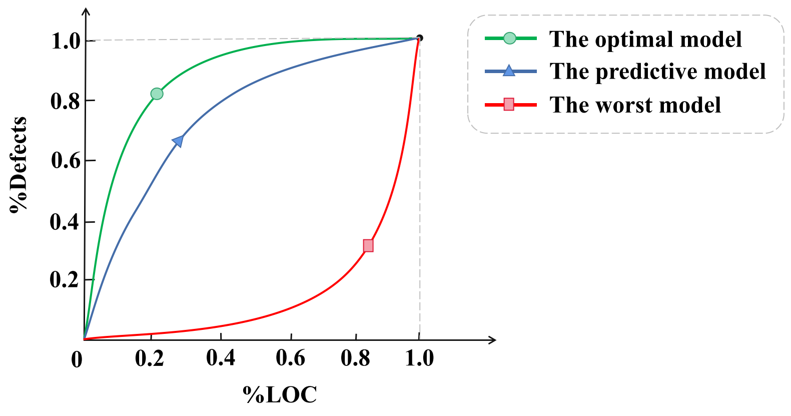


Fig. 6. A cumulative lift chart.

inspecting per 1% module:

$$PofB/PMI@20\% = \frac{PofB@20\%}{PMI@20\%}. \tag{4}$$

(4) **Precision@20%** is the percentage of truly faulty modules to the modules in the top 20% LOC:

$$Precision@20\% = \frac{p}{m}. \tag{5}$$

(5) **Recall@20%** is the percentage of truly defective modules in the top 20% LOC to all defective modules in the dataset:

$$Recall@20\% = \frac{p}{P}. \tag{6}$$

(6) **IFA** is the number of Initial False Alarms encountered by testers, when the first truly defective module is inspected.

(7) ***Popt*** measure the effectiveness of a predictive model which is determined through the use of a cumulative lift chart. As illustrated in Fig. 6, the x-axis represents the cumulative proportion of lines of code (LOC) inspected, while the y-axis represents the cumulative proportion of bugs detected. The chart features three distinct curves, each corresponding to a different model: the predictive model, the optimal model, and the worst model. The predictive model's curve is generated by sorting the software modules in descending order according to the predicted defect density, given by the EADP method. The optimal model's curve, on the other hand, is generated by sorting the modules in descending order according to the actual defect density. Finally, the worst model's curve is generated by sorting the modules in ascending order. The areas enclosed by the curves and the horizontal axis are represented as *Area(m)*, *Area(optimal)*, and *Area(worst)*, respectively. *Popt* is calculated as follows:

$$Popt = 1 - \frac{Area(optimal) - Area(m)}{Area(optimal) - Area(worst)}. \tag{7}$$

3.3. Baseline methods

To assess the efficacy of MOOAC, a comparative analysis is conducted with several state-of-the-art EADP methods.

(1) **CBS+** [18]: Classify Before Sorting (CBS+) firstly uses the logistic regression algorithm to predict the defect probability. Subsequently, the method individually assigns ranks to the predicted defective and clean modules by considering their relative defect density, calculated as the ratio between the predicted defect probability and LOC. Finally, the ranked clean modules are appended to the bottom of the ranking containing the ranked defective modules.

(2) **ManualUp** [17]: It considers smaller modules more likely to have higher defect density, so it provides the ranking of software modules in ascending order of LOC.

(3) **EALR** [23]: Effort-Aware Linear Regression (EALR) builds the linear regression to predict the defect density of a software module based on the software feature values of the module.

(4) **EATT** [21]: Effort-Aware Tri-Training (EATT) is a semi-supervised EADP method, which uses different classifiers and a tri-training strategy to construct the model. It firstly uses labeled data for the initial training of three classifiers. Then, in each round of the triple training, two classifiers are randomly selected, and some unlabeled data is labeled with the same predicted label by the two classifiers. Next, the third classifier is trained using the original labeled data and the new labeled data. Until all three classifiers do not change, the tri-training is stopped, and the integrated classifier construction is completed. In the model prediction process, EATT calculates the ratio between the defect probability provided by the integrated classifier and LOC as the defect density.

(5) **CBS+(DF)** [22]: Li et al. proposed an enhanced approach by incorporating deep forests into CBS+. Consequently, the classifier within CBS+ that distinguishes defective and clean modules was substituted with a deep forest classifier, leading to the nomenclature CBS+(DF).

(6) **EASC** [19,20]: Recently, Ni et al. showed the superiority of CBS+ for cross-project EADP and just-in-time EADP on JavaScript projects, respectively. Its execution process is similar to CBS+, with the distinction lying in the replacement of the classifier with a Bayesian classifier.

(7) **NSGA-II** : It is a modified version of the MOOAC by eliminating the classification process and utilizing solely a logistic regression model trained by NSGA-II [24] to calculate the defect density of software modules. In the prediction phase, the modules are ranked based on the ratio of the predicted defect probability determined by the logistic regression model and the Lines of Code (LOC) of the respective module. The aim of this comparison is to evaluate the effect of incorporating both classification and multi-objective optimization in the MOOAC.

3.4. Experimental design summary

In this experimental setup, a cross-version validation methodology is employed to evaluate the performance of the EADP model. Specifically, the model is built using data from a previous version of a software project and then applied to predict defects in the subsequent adjacent version of the same project. For instance, the EADP model is constructed using the Ant-1.3 dataset, and its predictive capability is assessed by predicting the detection priority of modules in the Ant-1.4 dataset (the adjacent new version dataset within the same "Ant" project). Due to the randomness of MOOAC, EATT, and NSGA-II, the entire procedure is repeated 50 times, and the median value of the 50 results is considered as the final outcome for each dataset.

Similar to previous study [55–59], we conduct the Wilcoxon signed-rank test [60] to examine the statistical significance of the differences between MOOAC and the compared method across all testing datasets. Due to the comparisons of MOOAC with multiple methods, we employ the Benjamini-Hochberg (BH) procedure [61] to correct the obtained

Table 2
The PofB@20% values of the eight methods in RQ1.

Testing dataset	MOOAC	CBS+	ManualUp	EALR	EATT	CBS+(DF)	EASC	NSGA-II
Ant-1.4	0.149	0.128	0.468	0.043	**0.489**	0.128	0.234	0.436
Ant-1.5	0.229	0.314	0.286	**0.343**	0.229	0.229	0.229	0.243
Ant-1.6	0.228	0.185	0.152	0.141	0.185	0.207	**0.234**	0.196
Ant-1.7	0.268	0.254	0.175	**0.281**	0.228	0.257	0.195	0.170
Camel-1.2	0.262	0.272	**0.410**	0.255	0.282	0.236	0.186	0.259
Camel-1.4	0.369	0.236	0.331	0.301	0.349	**0.448**	0.263	0.304
Camel-1.6	0.378	0.316	0.380	0.306	0.392	**0.406**	0.288	0.304
Ivy-1.4	0.333	0.222	0.278	0.222	0.222	**0.532**	0.222	0.278
Ivy-2.0	0.232	0.107	0.196	0.250	0.196	**0.405**	0.286	0.205
Jedit-4.0	**0.381**	0.257	0.199	0.292	0.288	0.301	0.261	0.248
Jedit-4.1	0.380	0.309	0.235	0.313	0.263	**0.382**	0.341	0.235
Jedit-4.2	**0.297**	0.198	0.160	0.226	0.292	0.245	0.245	0.269
Jedit-4.3	0.417	0.250	0.417	0.167	**0.583**	0.417	0.167	0.250
Log4j-1.1	0.372	0.360	0.163	**0.453**	0.221	0.372	0.407	0.256
Log4j-1.2	0.161	0.175	**0.547**	0.319	0.520	0.181	0.205	0.295
Lucene-2.2	0.332	0.254	**0.386**	0.309	0.377	0.232	0.215	0.295
Lucene-2.4	0.357	0.381	0.473	0.373	**0.483**	0.413	0.304	0.428
Poi-2.0	0.308	0.231	**0.462**	0.359	0.333	0.308	0.205	0.256
Poi-2.5	0.161	0.101	**0.494**	0.143	0.260	0.113	0.107	0.447
Poi-3.0	0.335	0.392	**0.422**	0.306	0.398	0.354	0.174	0.359
Synapse-1.1	0.263	0.253	0.273	0.202	0.283	**0.343**	0.232	0.222
Synapse-1.2	0.262	0.228	0.310	0.303	**0.345**	0.186	0.186	0.214
Velocity-1.5	0.378	**0.535**	0.480	0.181	0.532	0.532	0.492	0.517
Velocity-1.6	0.426	0.426	**0.458**	0.284	0.400	0.405	0.237	0.447
Xalan-2.5	0.168	0.171	0.539	0.405	**0.556**	0.271	0.147	0.433
Xalan-2.6	0.363	0.317	0.474	0.307	**0.478**	0.397	0.269	0.446
Xalan-2.7	0.271	0.261	**0.635**	0.378	0.632	0.339	0.203	0.575
Xerces-1.2	0.678	0.652	0.730	0.591	**0.791**	0.661	0.148	0.730
Xerces-1.3	0.259	0.176	0.466	0.326	0.456	**0.472**	0.166	0.451
Xerces-1.4	0.228	0.215	**0.451**	0.327	0.449	0.219	0.216	0.326
Average	0.308	0.273	0.382	0.290	0.384	0.317	0.235	0.336
W/D/L	–	22/1/7	9/1/20	18/0/12	10/1/19	12/4/14	23/1/6	16/0/14
p-value	–	**0.017**	0.067	0.419	**0.035**	0.770	**0.005**	0.717
$\|\delta\|$	–	0.257	0.359	0.107	0.323	0.054	0.457	0.084

p-value. MOOAC outperforms significantly than the compared method, if the corrected *p*-value after the BH procedure is less than 0.05. In addition, we also calculate the effect size (i.e., Cliff's δ [62]) to measure the magnitude of the difference obtained by two methods across all testing datasets. If $0 < |\delta| < 0.147$, the magnitude between the two methods is negligible. If $0.147 < |\delta| < 0.33$, the magnitude between the two methods is small. If $0.33 < |\delta| < 0.474$, the magnitude between the two methods is moderate. If $|\delta| \geq 0.474$, the magnitude between the two methods is large.

4. Experimental results

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

We conducted an analysis of the evaluation results for the eight methods based on seven performance metrics. The detailed values of PofB@20%, PMI@20%, and PofB/PMI@20% on each testing dataset are presented in Tables 2, 3, and 4, respectively. Fig. 7 illustrates the distribution of Precision@20%, Recall@20%, IFA, and *Popt* values across all testing datasets for the eight methods. Additionally, Table 5 provides the average values of these four evaluation metrics across all testing datasets.

The average IFA values for ManualUp, EATT, and NSGA-II exceed 10, and it is deemed unacceptable that all of the top 10 software modules recommended by EADP models are clean, as stated in prior studies [37,38]. Except for the above three methods, MOOAC exhibits the best overall performance on the two core metrics (i.e., PofB@20% and PMI@20%) and achieves the best performance in terms of Precision@20%, and IFA. The detailed results are as follows.

(1) As shown in Fig. 7(c) and Table 5, MOOAC achieves the lowest average **IFA** value (i.e., 3.283), and there is a statistically significant difference between MOOAC and ManualUp, EATT, and NSGA-II. The average IFA values of ManualUp, EATT, and NSGA-II are greater than 10, since the three methods tend to rank modules with fewer LOC first and the modules are more likely to be clean. The previous studies [37, 38] have pointed out that testers would not use the EADP model, if the first 10 software modules returned by the model are all false alarms. Therefore, the performance of the three methods on other metrics has no practical significance. Compared with NSGA-II, MOOAC introduces the classification process, so its IFA value is significantly lower than that of NSGA-II.

(2) As shown in Table 2, except for the above three methods, MOOAC achieved a average value on **PofB@20%** slightly lower than CBS+(DF), with no noticeable difference between the two. The row Win/Draw/Loss (W/D/L) provides a summary of the number of instances where MOOAC outperforms, achieves equal performance, or performs worse than other methods. MOOAC wins CBS+, EALR and EASC on 22, 18 and 23 datasets in terms of PofB@20%, respectively. MOOAC can improve the average PofB@20% values of CBS+, EALR and EASC by 12.82%, 6.21% and 31.06%, respectively. MOOAC has a significant difference with CBS+ and EASC with a moderate improvement according to the *p*-value and Cliff's δ.

(3) As shown in Table 3, MOOAC achieved a average value on **PMI@20%** slightly lower than EASC, with no noticeable difference between the two and performs the best on most datasets. MOOAC wins CBS+, ManualUp, EALR, EATT, CBS+(DF), EASC and NSGA-II on 20, 30, 28, 30, 26, 17 and 29 datasets in terms of PMI@20%, respectively. Except for EASC, MOOAC can reduce the average PMI@20% values of the baseline methods by 22.45%, 72.34%, 45.71%, 70.63%, 28.03% and 60.33%, respectively. MOOAC significantly outperforms CBS+ and CBS+(DF) with a small improvement, and ManualUp, EALR, EATT, and NSGA-II with a large improvement according to the *p*-value and Cliff's δ.

(4) As shown in Table 4, MOOAC obtains the highest **PofB/PMI@20 %** value on 14 datasets. It wins CBS+, ManualUp, EALR, EATT, CBS+(DF), EASC and NSGA-II on 22, 30, 30, 30, 25, 19 and 28 datasets in

Table 3
The PMI@20% values of the eight methods in RQ1.

Testing dataset	MOOAC	CBS+	ManualUp	EALR	EATT	CBS+(DF)	EASC	NSGA-II
Ant-1.4	0.084	0.073	0.624	0.084	0.506	**0.010**	0.112	0.573
Ant-1.5	**0.263**	0.352	0.662	0.410	0.625	0.410	0.485	0.544
Ant-1.6	**0.068**	**0.068**	0.652	0.177	0.573	0.105	0.199	0.323
Ant-1.7	**0.079**	0.098	0.643	0.086	0.609	0.117	0.128	0.240
Camel-1.2	0.176	0.373	0.627	0.191	0.423	0.225	**0.109**	0.244
Camel-1.4	0.171	**0.094**	0.635	0.446	0.653	0.205	0.119	0.614
Camel-1.6	**0.092**	0.331	0.639	0.382	0.617	0.102	0.107	0.604
Ivy-1.4	0.261	0.282	0.743	0.390	0.714	0.373	**0.162**	0.635
Ivy-2.0	0.384	0.571	0.736	0.531	0.716	0.429	**0.074**	0.236
Jedit-4.0	**0.096**	0.199	0.745	0.376	0.680	0.235	0.206	0.691
Jedit-4.1	**0.109**	0.125	0.734	0.391	0.718	0.128	0.138	0.694
Jedit-4.2	**0.105**	0.139	0.695	0.346	0.657	0.150	0.131	0.326
Jedit-4.3	0.056	**0.041**	0.703	0.181	0.600	0.069	0.081	0.186
Log4j-1.1	**0.142**	0.156	0.569	0.193	0.495	0.156	0.156	0.339
Log4j-1.2	**0.122**	0.146	0.605	0.351	0.580	0.141	0.171	0.280
Lucene-2.2	**0.119**	0.239	0.664	0.348	0.640	0.227	0.178	0.421
Lucene-2.4	0.263	0.468	0.700	0.482	0.641	0.415	**0.194**	0.468
Poi-2.0	0.193	0.382	0.634	0.347	0.602	0.411	**0.146**	0.463
Poi-2.5	0.092	**0.029**	0.636	0.278	0.519	0.039	0.065	0.445
Poi-3.0	0.394	0.466	0.658	0.450	0.586	0.414	**0.136**	0.467
Synapse-1.1	0.171	**0.072**	0.577	0.252	0.545	0.288	0.221	0.320
Synapse-1.2	**0.105**	0.117	0.594	0.305	0.570	0.109	0.141	0.311
Velocity-1.5	**0.530**	0.729	0.738	0.332	0.748	0.724	0.729	0.689
Velocity-1.6	0.380	0.450	0.746	0.498	0.747	0.493	**0.188**	0.624
Xalan-2.5	0.085	**0.059**	0.724	0.478	0.707	0.245	0.103	0.457
Xalan-2.6	0.296	0.320	0.727	0.508	0.711	0.379	**0.168**	0.668
Xalan-2.7	0.249	0.226	0.717	0.415	0.704	0.309	**0.158**	0.639
Xerces-1.2	0.434	0.664	0.841	0.489	0.861	0.432	**0.116**	0.708
Xerces-1.3	0.099	**0.018**	0.839	0.340	0.857	0.417	0.077	0.762
Xerces-1.4	0.071	**0.070**	0.808	0.449	0.818	0.071	0.094	0.398
Average	0.190	0.245	0.687	0.350	0.647	0.264	0.170	0.479
W/D/L	–	20/1/9	30/0/0	28/1/1	30/0/0	26/1/3	17/0/13	29/0/1
p-value	–	**0.011**	**0.000**	**0.000**	**0.000**	**0.000**	0.673	**0.000**
$\|\delta\|$	–	0.088	1.000	0.621	0.993	0.280	0.022	0.820

terms of PofB/PMI@20%, respectively. In addition, MOOAC achieves the highest average PofB/PMI@20% value (i.e., 2.222), and improves the average PofB/PMI@20% values of CBS+ by 9.19%, of ManualUp by 301.08%, of EALR by 140.22%, of EATT by 274.70%, of CBS+(DF) by 33.37%, of EASC by 33.21% and of NSGA-II by 201.90%. MOOAC significantly performs better than CBS+, CBS+(DF) and EASC with a small improvement, and ManualUp, EALR, EATT, and NSGA-II with a large improvement according to the p-values and Cliff's δ. Therefore, MOOAC exhibits the best overall performance on the two metrics (i.e., PofB@20% and PMI@20%). In other words, MOOAC enables testers to identify more bugs per 1% module.

(5) As shown in Fig. 7(a), (b), (d) and Table 5, MOOAC achieves the highest average **Precision@20%** value (i.e., 0.522), and significantly outperforms ManualUp, EALR, EATT, CBS+(DF) and NSGA-II. Except for ManualUp, EATT, and NSGA-II, CBS+(DF) achieves the best average value on **Recall@20%** (i.e., 0.322) and ***Popt*** (i.e., 0.624).

Answer to RQ1: Except for ManualUp, EATT, and NSGA-II whose IFA values exceed 10, MOOAC exhibits the best overall performance on the two metrics, PofB@20% and PMI@20%. In other words, MOOAC enables testers to identify more bugs per 1% module.

4.2. RQ2: Does the data imbalance problem affect the performance of MOOAC?

We first apply RUS to the target training datasets whose defective ratios are less than 0.5. The desired defective ratio is set as 0.5, similar to [33]'s study. Then, MOOAC, CBS+, EALR, EATT, CBS+(DF), EASC and NSGA-II are trained on the balanced datasets, and we denote them as MOOAC*, CBS+*, EALR*, EATT*, CBS+(DF)*, EASC* and NSGA-II*. We also compare the performance of the MOOAC method trained on the original datasets with MOOAC*. We do not investigate the performance of ManualUp, since it is an unsupervised learning method. Tables 6, 7, and 8 show the detailed PofB@20%, PMI@20%, and PofB/PMI@20% values on each testing dataset of the methods. Fig. 8 shows the distribution of the Precision@20%, Recall@20%, IFA, and *Popt* values of the methods across 24 testing datasets, and Table 9 presents the average value of the four evaluation metrics of the methods on 24 testing datasets.

The average IFA values of EATT* and NSGA-II* remain higher than 10, which is considered unacceptable as it implies that the top 10 software modules recommended by the EADP models are all clean [37,38]. Except for the above two methods, MOOAC* achieves the best performance in terms of PofB@20%, PofB/PMI@20%, and Precision@20%on the balanced datasets. However, MOOAC* does not outperform MOOAC on most of the metrics. The detailed results are as follows.

(1) As shown in Fig. 8(c) and Table 9, MOOAC* achieves the second lowest average **IFA** value (i.e., 7.875) among the seven methods on the balanced datasets, and there is a statistically significant difference between MOOAC* and EATT. MOOAC achieves the lower average IFA value than MOOAC*, and there is a statistically difference between these two methods.

(2) As shown in Table 6, except for EATT* and NSGA-II*, MOOAC* achieves the best average value on **PofB@20%** on the balanced datasets. MOOAC* wins CBS+*, EALR*, CBS+(DF)* and EASC* on 18, 13, 11 and 19 datasets in terms of PofB@20%, respectively. MOOAC* can improve the average PofB@20% values of CBS+*, EALR*, CBS+(DF)* and EASC* by 11.32%, 3.15%, 2.43% and 27.71% respectively. MOOAC* has a significant difference with CBS+* with a small improvement, with EASC* and NSGA-II* with a moderate improvement according to the p-value and Cliff's δ. MOOAC and MOOAC* achieve almost equal average PofB@20% values.

Table 4
The PofB/PMI@20% values of the eight methods in RQ1(It is worth noting that the data in this table is the median value of 50 cross-version results instead of obtained by dividing the corresponding data in Tables 2 and 3).

Testing dataset	MOOAC	CBS+	ManualUp	EALR	EATT	CBS+(DF)	EASC	NSGA-II
Ant-1.4	1.767	1.748	0.751	0.505	0.968	1.337	**2.083**	0.776
Ant-1.5	0.870	0.894	0.432	0.837	0.366	0.558	0.472	0.433
Ant-1.6	**3.338**	2.702	0.233	0.800	0.323	1.959	1.172	0.608
Ant-1.7	**3.344**	2.597	0.271	3.272	0.374	2.204	1.531	0.710
Camel-1.2	**1.491**	0.729	0.654	1.335	0.666	1.046	1.712	1.034
Camel-1.4	2.121	**2.508**	0.522	0.676	0.535	2.181	2.203	0.496
Camel-1.6	**4.133**	0.956	0.594	0.800	0.636	3.998	2.698	0.504
Ivy-1.4	**1.275**	0.788	0.374	0.569	0.311	0.744	1.373	0.438
Ivy-2.0	0.605	0.188	0.267	0.471	0.274	0.416	3.868	0.842
Jedit-4.0	**4.061**	1.287	0.267	0.777	0.423	1.279	1.268	0.364
Jedit-4.1	**3.592**	2.470	0.320	0.801	0.366	2.983	2.474	0.337
Jedit-4.2	**2.831**	1.426	0.231	0.654	0.445	1.637	1.875	0.811
Jedit-4.3	**7.321**	6.150	0.592	0.921	0.973	6.029	2.050	1.345
Log4j-1.1	**2.535**	2.311	0.286	2.354	0.446	2.386	2.609	0.770
Log4j-1.2	1.261	1.194	0.903	0.909	0.896	1.278	1.200	0.965
Lucene-2.2	**2.892**	1.062	0.582	0.888	0.589	1.023	1.207	0.695
Lucene-2.4	1.409	0.815	0.676	0.774	0.753	0.996	1.565	0.922
Poi-2.0	1.677	0.604	0.728	1.034	0.554	0.749	1.400	0.547
Poi-2.5	1.750	3.528	0.776	0.515	0.501	2.898	1.646	0.998
Poi-3.0	0.856	0.841	0.641	0.680	0.679	0.855	1.282	0.770
Synapse-1.1	1.534	**3.504**	0.473	0.801	0.519	1.191	1.053	0.725
Synapse-1.2	**2.485**	1.942	0.523	0.996	0.605	1.702	1.324	0.657
Velocity-1.5	0.716	0.734	0.651	0.546	0.711	0.734	0.676	0.746
Velocity-1.6	1.108	0.948	0.613	0.571	0.536	0.821	1.261	0.709
Xalan-2.5	1.979	**2.928**	0.744	0.847	0.785	1.105	1.421	0.969
Xalan-2.6	1.231	0.991	0.652	0.604	0.673	1.048	1.597	0.665
Xalan-2.7	1.095	1.159	0.885	0.910	0.898	1.096	1.280	0.897
Xerces-1.2	**1.567**	0.983	0.869	1.210	0.919	1.530	1.275	0.989
Xerces-1.3	2.608	**9.975**	0.556	0.960	0.532	1.130	2.146	0.583
Xerces-1.4	**3.196**	3.082	0.558	0.728	0.549	3.061	2.304	0.768
Average	2.222	2.035	0.554	0.925	0.593	1.666	1.668	0.736
W/D/L	–	22/0/8	30/0/0	30/0/0	30/0/0	25/0/5	19/0/11	28/0/2
p-value	–	**0.026**	**0.000**	**0.000**	**0.000**	**0.000**	**0.022**	**0.000**
$\|\delta\|$	–	0.227	0.944	0.764	0.940	0.316	0.209	0.869

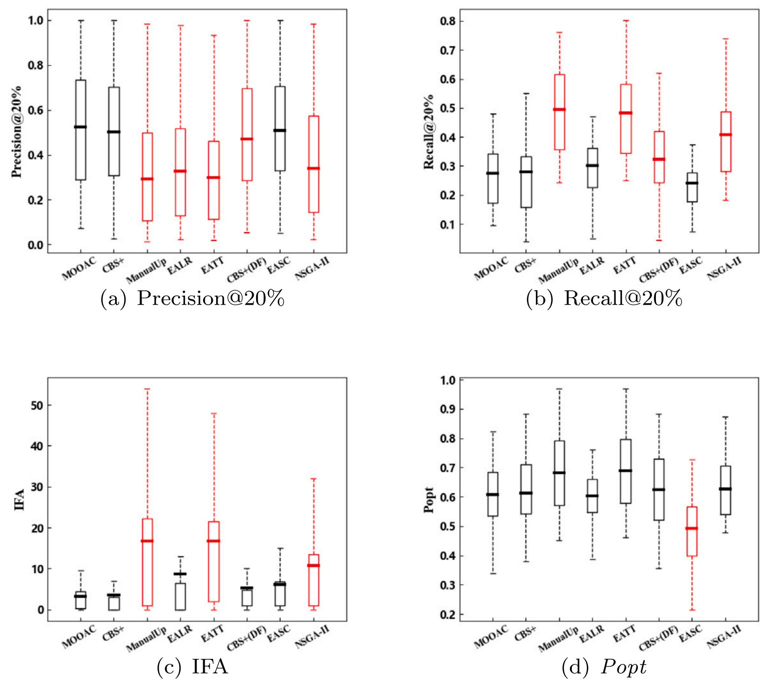


Fig. 7. The boxplots of Precision@20%, Recall@20%, IFA and *Popt* of the eight methods in RQ1. (The red boxplot indicates that the difference between MOOAC and the corresponding method is significant).

(3) As shown in Table 7, MOOAC* achieves the second best average **PMI@20%** value (i.e., 0.199) on the balanced datasets. MOOAC* wins CBS+*, EALR*, EATT*, CBS+(DF)* and NSGA-II* on 21, 22, 24, 19 and 23 datasets in terms of PMI@20%, respectively. Except for EASC*, MOOAC* can reduce the average PMI@20% values of the five methods by 23.75%, 42.98%, 69.15%, 23.46% and 52.51%, respectively. MOOAC* significantly outperforms CBS+* and CBS+(DF)* with a moderate improvement, and EALR*, EATT* and NSGA-II* with

Table 5
The average Precision@20%, Recall@20%, IFA and *Popt* values of the eight methods in RQ1.

Metrics	MOOAC	CBS+	ManualUp	EALR	EATT	CBS+(DF)	EASC	NSGA-II
Precision@20%	**0.522**	0.502	0.292	0.326	0.298	0.470	0.507	0.338
Recall@20%	0.274	0.280	**0.494**	0.301	0.483	0.322	0.241	0.409
IFA	**3.283**	3.567	16.767	8.733	16.683	5.300	6.100	10.783
Popt	0.608	0.613	0.682	0.603	**0.690**	0.624	0.493	0.626

a large improvement according to the *p*-values and Cliff's δ. MOOAC significantly outperforms MOOAC* and wins it on 18 datasets.

(4) As shown in Table 8, MOOAC* obtains the highest **PofB/PMI @20%** value on 12 datasets among the seven methods trained on the balanced datasets. It wins CBS+*, EALR*, EATT*, CBS+(DF)*, EASC* and NSGA-II* on 21, 21, 24, 20, 15 and 22 datasets in terms of PofB/PMI@20%, respectively. In addition, MOOAC* improves the average PofB/PMI@20% values of CBS+* by 57.55%, of EALR* by 0.24%, of EATT* by 187.61%, of CBS+(DF)* by 40.26%, of EASC* by 5.78% and of NSGA-II* by 64.96%. MOOAC* significantly performs better than CBS+(DF)* with a moderate improvement, and CBS+*, EALR*, EATT*, and NSGA-II* with a large improvement according to the *p*-value and Cliff's δ. MOOAC significantly outperforms MOOAC* and wins it on 19 datasets.

(5) As shown in Fig. 8(a), (b), (d) and Table 9, MOOAC* significantly outperforms CBS+*, EALR*, EATT*, CBS+(DF)* and NSGA-II*, and achieves the highest average **Precision@20%** value (i.e., 0.473). Except for EATT* and NSGA-II*, CBS+(DF)* achieves the best average value on **Recall@20%** (i.e., 0.323), and EALR* achieves the best average value on ***Popt*** (i.e., 0.603). MOOAC outperforms MOOAC* in terms of the average Precision@20% and *Popt*. Although MOOAC* achieves the higher average Recall@20% value than MOOAC, there is no statistically significant difference between these two methods.

(6) In summary, RUS cannot significantly improve the performance of MOOAC. Instead, the performance on PMI@20%, PofB/PMI@20%, Precision@20%, IFA, and *Popt* is degraded. The potential reasons are as followings. In addition to the class imbalance problem, there is the imbalanced distribution of the defect densities of defective modules. RUS can only solve the class imbalance problem. After applying RUS to the class imbalanced datasets, the percentages between defective modules and clean ones can be equal. However, the distribution of the defect densities of defective modules is still imbalanced.

Answer to RQ2: Except for EATT and NSGA-II whose IFA values are greater than 10, MOOAC* outperforms the state-of-the-art methods trained on the balanced datasets in terms of Precision@20%, PofB@20% and PofB@20%/PMI@20%. However, applying RUS to the defect datasets does not improve the performance of MOOAC.

4.3. RQ3: Does the underlying classifier of MOOAC affect the performance of MOOAC?

To assess the impact, we analyze the results of MOOAC embedded with the five classifiers, including Random Forest (RF), Naive Bayes (NB), Logistic Regression (LR), Decision Tree (DT), and K-Nearest Neighbor (KNN). The five classifiers are widely used in previous SDP studies. Fig. 9 shows the performance distribution of MOOAC with the five classifiers across all testing datasets, and Table 10 presents the average value across all testing datasets.

RF achieves the best performance in terms of PofB@20%, Recall@20%, and *Popt*. NB achieves the best performance in terms of PMI@20% and Precision@20%. LR achieves the best performance in terms of PofB/PMI@20% and IFA. We finally choose RF as the underlying classifier for MOOAC, since the primary objective of EADP is to find more defects and RF performs the best in terms of PofB@20%. The detailed results are as follows.

(1) As shown in Fig. 9(a), (b), and (c) and Table 10, RF achieves the highest average **PofB@20%** value (i.e., 0.308) among the five classifiers. It significantly outperforms NB, LR, and KNN in terms of PofB@20%. NB achieves the lowest average **PMI@20%** and PofB@20% values (i.e., 0.125 and 0.280, respectively) among the five classifiers, but there is no statistically significant difference between NB and other classifiers in terms of PMI@20%. LR achieves the highest average **PofB/PMI@20%** value (i.e., 3.161) among the five classifiers, but there is no statistically significant difference between RF and LR on the evaluation metric.

(2) As shown in Fig. 9(d), (e), (g) and Table 10, NB achieves the best average **Precision@20%** value (i.e., 0.560) among five classifiers, but there is no statistically significant difference between RF and NB in terms of Precision@20%. RF achieves the highest average **Recall@20%** and ***Popt*** values (i.e., 0.274 and 0.608, respectively) among five classifiers. RF significantly outperforms LR in terms of Recall@20%, and significantly performs better than NB, DT, and KNN in terms of *Popt*.

(3) As shown in Fig. 9(f) and Table 10, LR achieves the lowest average **IFA** value (i.e., 3.083) among the five classifiers, but there is no statistically significant difference between LR and other classifiers.

Answer to RQ3: The different underlying classifiers affect the performance of MOOAC. RF performs the best in terms of PofB@20%, Recall@20%, and *Popt*, so we recommend to use RF as the underlying classifier of MOOAC.

4.4. RQ4: Does the defective threshold to distinguish the defective and clean modules affect the performance of MOOAC?

We analyze the evaluation results of MOOAC using different defective threshold to distinguish the defective and clean modules in terms of the seven performance metrics. Fig. 10 shows the performance distribution of MOOAC using the different defective thresholds.

Setting the defective threshold as 0.5 yields optimal performance for the PofB@20% metric, and good results for other metrics as well.

(1) As shown in Fig. 10(b), as the threshold varies, **IFA** shows significant fluctuations, but remains below a level of 4 in the range of 0.2 to 0.8, whereas excessively high or low thresholds may result in abnormal IFA values or even loss of practical significance (greater than 10).

(2) As shown in Fig. 10(a), the **PofB@20%** remains stable as the threshold varied, which may be due to the fact that the top-ranked modules in the ranking almost do not change when inspecting the top 20% of the LOC. Within the threshold range of 0.2 to 0.8, both **PMI@20%** and **PofB/PMI@20%** remain stable and achieve good performances. Within this range, MOOAC has demonstrated consistently excellent performance across various metrics, thereby making it an ideal range for threshold selection. A low threshold would cause many truly clean modules with low LOC values to be ranked at the top of the ranking, resulting in the poor performance of the PMI@20%. Conversely, a high threshold would cause the large inspection budget to be allocated to inspect modules predicted to be clean, whose ranking results are largely determined by corresponding LOC values, resulting

Table 6
The PofB@20% values of the eight methods in RQ2.

Testing dataset	MOOAC*	CBS+*	EALR*	EATT*	CBS+(DF)*	EASC*	NSGA-II*	MOOAC
Ant1.4	0.234	0.149	0.255	**0.468**	0.319	0.128	0.298	0.149
Ant1.5	**0.314**	0.229	0.286	0.229	0.229	0.257	**0.314**	0.229
Ant1.6	0.277	0.255	0.109	0.196	**0.293**	0.223	0.264	0.228
Ant1.7	0.219	**0.225**	0.222	0.210	0.178	0.195	0.189	0.268
Camel1.2	0.195	0.215	0.190	**0.395**	0.230	0.167	0.326	0.262
Camel1.4	0.324	0.304	0.251	0.310	0.373	0.322	**0.348**	0.369
Camel1.6	0.318	0.208	0.340	**0.380**	0.318	0.304	0.350	0.378
Ivy2.0	**0.268**	0.196	0.179	0.196	0.143	0.250	0.223	0.232
Jedit4.0	**0.365**	0.283	0.288	0.190	0.341	0.283	0.327	0.381
Jedit4.1	**0.378**	0.276	0.318	0.309	0.318	0.336	0.359	0.380
Jedit4.2	0.302	0.311	0.208	0.236	**0.330**	0.236	0.307	0.297
Jedit4.3	0.250	0.417	0.083	**0.583**	0.333	0.250	0.333	0.417
Log4j1.1	0.372	0.186	0.267	0.186	0.256	**0.384**	0.244	0.372
Log4j1.2	0.152	0.229	0.337	0.458	0.231	0.217	**0.494**	0.161
Lucene2.2	0.339	0.254	0.302	**0.362**	0.239	0.196	0.312	0.332
Poi2.5	0.214	0.188	0.232	0.286	0.202	0.131	**0.474**	0.161
Synapse1.1	0.202	0.111	0.202	0.273	0.172	0.212	**0.283**	0.263
Synapse1.2	0.248	0.221	**0.276**	0.269	0.193	0.207	0.269	0.262
Xalan2.5	0.271	0.264	0.414	**0.529**	0.256	0.147	0.426	0.168
Xalan2.6	0.373	0.323	0.307	**0.462**	0.398	0.269	0.452	0.363
Xalan2.7	0.285	0.304	0.378	**0.625**	0.341	0.211	0.575	0.271
Xerces1.2	0.661	0.670	0.600	**0.765**	0.670	0.148	0.735	0.678
Xerces1.3	0.259	0.249	0.435	**0.487**	0.254	0.264	0.456	0.259
Xerces1.4	0.270	0.293	0.382	**0.466**	0.300	0.214	0.405	0.228
Average	0.295	0.265	0.286	0.370	0.288	0.231	0.365	0.296
W/D/L	–	16/0/8	13/1/10	9/0/15	11/1/12	19/1/4	7/1/16	11/2/11
p-value	–	**0.045**	0.705	**0.047**	0.808	**0.003**	**0.034**	0.987
$\|\delta\|$	–	0.267	0.030	0.246	0.061	0.444	0.387	0.007

Table 7
The PMI@20% values of the eight methods in RQ2.

Testing dataset	MOOAC*	CBS+*	EALR*	EATT*	CBS+(DF)*	EASC*	NSGA-II*	MOOAC
Ant1.4	0.123	0.174	0.376	0.478	0.242	**0.101**	0.258	0.084
Ant1.5	**0.176**	0.273	0.386	0.625	0.396	0.423	0.401	0.263
Ant1.6	0.108	0.191	0.242	0.570	**0.103**	0.225	0.207	0.068
Ant1.7	**0.105**	0.187	0.235	0.495	0.213	0.117	0.205	0.079
Camel1.2	0.183	0.186	**0.063**	0.628	0.212	0.132	0.401	0.176
Camel1.4	0.249	0.233	0.487	0.640	0.345	**0.131**	0.573	0.171
Camel1.6	0.144	0.218	0.451	0.547	0.237	**0.120**	0.590	0.092
Ivy2.0	0.290	0.219	0.429	0.733	**0.205**	0.071	0.389	0.384
Jedit4.0	0.176	0.284	0.235	0.735	0.297	**0.163**	0.198	0.096
Jedit4.1	0.173	0.260	0.494	0.686	0.244	0.186	**0.133**	0.109
Jedit4.2	0.277	0.327	**0.011**	0.662	0.232	0.177	0.410	0.105
Jedit4.3	**0.106**	0.175	0.148	0.630	0.185	0.126	0.266	0.056
Log4j1.1	**0.128**	0.229	0.239	0.550	0.193	0.156	0.252	0.142
Log4j1.2	**0.122**	0.224	0.376	0.541	0.224	0.185	0.515	0.122
Lucene2.2	**0.138**	0.259	0.348	0.623	0.267	0.178	0.350	0.119
Poi2.5	0.273	0.319	0.423	0.535	0.281	**0.125**	0.517	0.092
Synapse1.1	**0.090**	0.144	0.284	0.568	0.140	0.207	0.178	0.171
Synapse1.2	0.189	0.266	0.320	0.566	0.195	**0.172**	0.396	0.105
Xalan2.5	0.204	0.224	0.506	0.699	0.202	**0.121**	0.458	0.085
Xalan2.6	0.326	0.316	0.507	0.721	0.385	**0.165**	0.672	0.296
Xalan2.7	0.271	0.274	0.416	0.711	0.308	**0.164**	0.643	0.249
Xerces1.2	0.426	0.670	0.441	0.857	0.479	**0.152**	0.752	0.434
Xerces1.3	0.288	0.371	0.404	0.839	0.373	**0.152**	0.765	0.099
Xerces1.4	0.151	0.247	0.566	0.832	0.286	**0.146**	0.531	0.071
Average	0.199	0.261	0.349	0.645	0.260	0.163	0.419	0.153
W/D/L	–	21/0/3	22/0/2	24/0/0	19/0/5	9/0/16	23/0/1	5/1/18
p-value	–	**0.000**	**0.000**	**0.000**	**0.000**	0.152	**0.000**	0.007
$\|\delta\|$	–	0.389	0.625	1.000	0.403	0.229	0.708	0.443

in poor performance of the PMI@20%. The 0.5 threshold is commonly used in practice, and it guarantees the best performance for PofB@20%.

(3) As shown in Fig. 10(b), the **Precision@20%**, **Recall@20%**, and ***Popt*** remain stable with small fluctuations in the range of 0.2 to 0.8.

Answer to RQ4: We recommend setting the defective threshold to 0.5, as it demonstrates the best performance on PofB@20% and satisfactory performance on other metrics.

4.5. RQ5: Does the model training strategy affect the performance of MOOAC?

(1) The type of model for predicted defective modules.

We analyze the evaluation results of MOOAC using the logistic regression model and the linear regression model in terms of the seven performance metrics. Fig. 11 shows the performance distribution of MOOAC using the two models across all testing datasets, and Table 11 presents the average values across all testing datasets.

Table 8
The PofB/PMI@20% values of the eight methods in RQ2.

Testing dataset	MOOAC*	CBS+*	EALR*	EATT*	CBS+(DF)*	EASC	NSGA-II*	MOOAC
Ant1.4	**1.894**	0.855	0.678	0.980	1.321	1.262	1.262	1.767
Ant1.5	**1.346**	0.837	0.741	0.366	0.577	0.608	0.800	0.870
Ant1.6	2.585	1.338	0.449	0.343	**2.861**	0.990	1.263	3.338
Ant1.7	**2.090**	1.205	0.945	0.424	0.832	1.672	0.929	3.344
Camel1.2	1.070	1.154	**3.034**	0.634	1.083	1.267	0.823	1.491
Camel1.4	1.335	1.308	0.514	0.485	1.081	**2.466**	0.611	2.121
Camel1.6	2.222	0.956	0.754	0.695	1.340	**2.529**	0.595	4.133
Ivy2.0	0.929	0.898	0.416	0.268	0.698	**3.520**	0.570	0.605
Jedit4.0	**1.981**	0.996	1.222	0.259	1.146	1.733	1.645	4.061
Jedit4.1	2.129	1.065	0.644	0.450	1.305	1.721	**2.719**	3.592
Jedit4.2	1.092	0.952	**19.042**	0.356	1.426	1.332	0.747	2.831
Jedit4.3	2.301	**2.384**	0.562	0.926	1.802	1.984	1.308	7.321
Log4j1.1	**2.806**	0.811	1.121	0.338	1.328	2.460	0.784	2.535
Log4j1.2	**1.227**	1.020	0.898	0.846	1.029	1.170	0.951	1.261
Lucene2.2	**2.474**	0.979	0.867	0.581	0.895	1.098	0.870	2.892
Poi2.5	0.784	0.587	0.548	0.535	0.729	**1.051**	0.912	1.750
Synapse1.1	**2.198**	0.771	0.712	0.481	1.230	1.024	1.581	1.534
Synapse1.2	**1.292**	0.831	0.861	0.475	0.989	1.204	0.730	2.485
Xalan2.5	**1.328**	1.176	0.819	0.757	1.270	1.216	0.974	1.979
Xalan2.6	1.144	1.022	0.606	0.641	1.034	**1.629**	0.671	1.231
Xalan2.7	1.057	1.111	0.908	0.879	1.108	**1.288**	0.896	1.095
Xerces1.2	**1.565**	0.999	1.361	0.893	1.396	0.971	0.983	1.567
Xerces1.3	0.967	0.671	1.078	0.581	0.681	**1.735**	0.603	2.608
Xerces1.4	**1.746**	1.189	0.674	0.560	1.050	1.465	0.759	3.196
Average	1.648	1.046	1.644	0.573	1.175	1.558	0.999	2.484
W/D/L	–	21/0/3	21/0/3	24/0/0	20/0/4	15/0/9	22/0/2	5/0/19
p-value	–	**0.000**	**0.000**	**0.000**	**0.000**	0.152	**0.000**	**0.007**
$\lvert\delta\rvert$	–	0.642	0.705	0.976	0.403	0.229	0.691	0.399

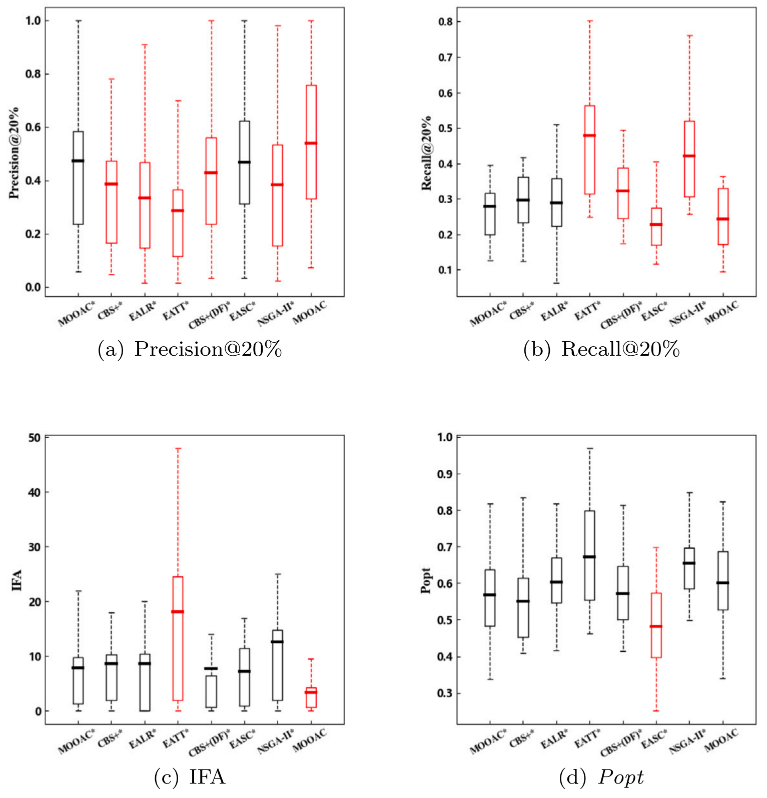
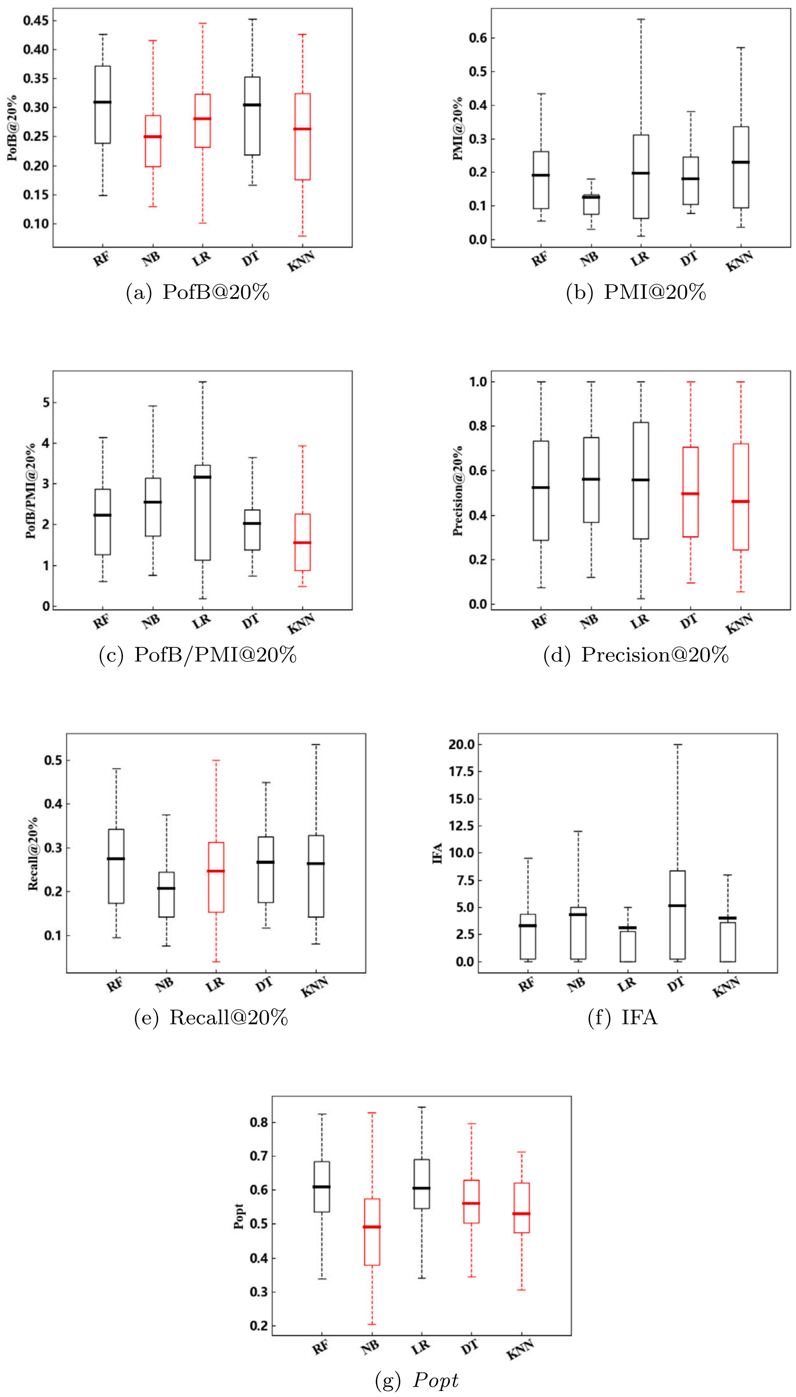


Fig. 8. The boxplots of Precision@20%, Recall@20%, IFA, and *Popt* of the eight methods in RQ2. (The red boxplot indicates that the difference between MOOAC* and the corresponding method is significant).

As shown in Fig. 11 and Table 11, MOOAC using the logistic regression model achieves the best average PofB@20%, Recall@20%, and *Popt* values (i.e., 0.308, 0.274, and 0.608, respectively). And there is a significant difference between these two models in terms of PofB@20% and Recall@20%. Although MOOAC using the linear regression model achieves the best average PMI@20%, PofB/PMI@20%,

Table 9
The average Precision@20%, Recall@20%, IFA, and *Popt* values of the eight methods in RQ2.

Metrics	MOOAC*	CBS+*	EALR*	EATT*	CBS+(DF)*	EASC*	NSGA-II*	MOOAC
Precision@20%	**0.473**	0.387	0.335	0.280	0.428	0.468	0.383	0.538
Recall@20%	0.279	0.297	0.289	**0.471**	0.323	0.227	0.422	0.244
IFA	7.875	8.625	8.625	19.625	7.708	**7.208**	12.583	3.354
Popt	0.569	0.550	0.603	**0.671**	0.572	0.482	0.656	0.601



(a) PofB@20%
(b) PMI@20%
(c) PofB/PMI@20%
(d) Precision@20%
(e) Recall@20%
(f) IFA
(g) *Popt*

Fig. 9. The boxplot of Precision@20%, Recall@20%, PofB@20%, PMI@20%, PofB/PMI@20%, IFA, and *Popt* of MOOAC embedding the five classifiers. (The red boxplot indicates that the difference between RF and the corresponding classifier is significant).

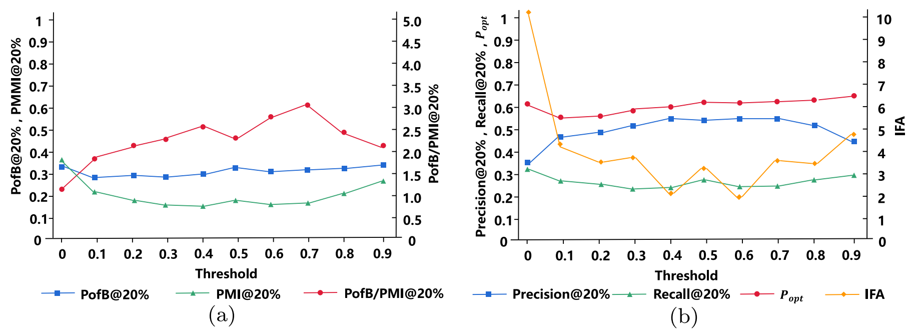


Fig. 10. The line chart of PofB@20%, PMI@20%, PofB/PMI@20%, Precision@20%, Recall@20%, IFA, and *Popt* of MOOAC applied different threshold on distinguishing defective and clean modules.

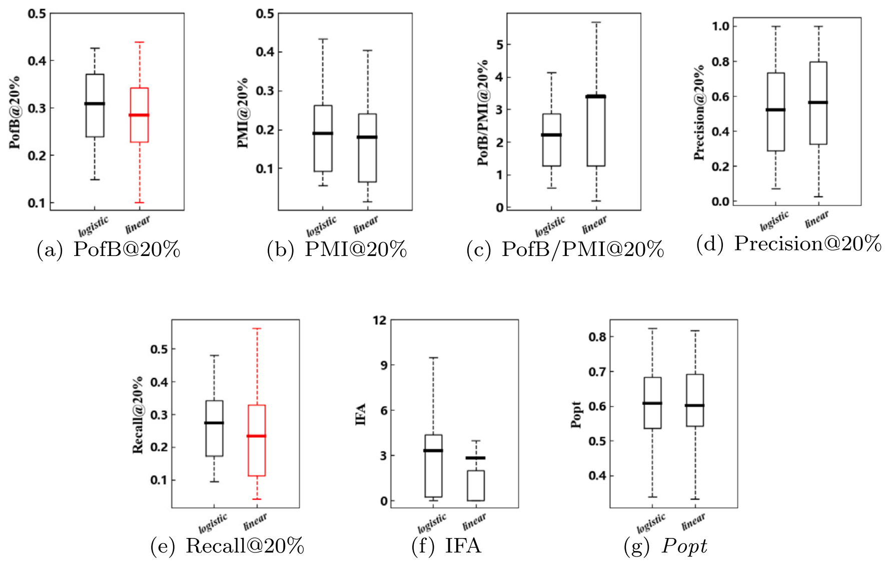


Fig. 11. The boxplots of PofB@20%, PMI@20%, PofB/PMI@20%, Precision@20%, Recall@20%, IFA, and *Popt* of MOOAC trained on logistic regression model and linear regression model. (The red boxplot indicates that the difference between logistic regression model and linear regression model is significant).

Table 10
The average PofB@20%, PMI@20%, PofB/PMI@20%, Precision@20%, Recall@20%, IFA, and *Popt* values of MOOAC embedding the five classifiers.

Metrics	RF	NB	LR	DT	KNN
PofB@20%	**0.308**	0.249	0.280	0.304	0.262
PMI@20%	0.190	**0.125**	0.197	0.181	0.229
PofB/ PMI@20%	2.222	0.254	**3.161**	2.032	1.551
Precision@20%	0.522	**0.560**	0.557	0.495	0.459
Recall@20%	**0.274**	0.207	0.246	0.266	0.264
IFA	3.283	4.317	**3.083**	5.133	4.000
Popt	**0.608**	0.491	0.605	0.559	0.530

Table 11
The average PofB@20%, PMI@20%, PofB/PMI@20%, Precision@20%, Recall@20%, IFA, and *Popt* values of MOOAC using the logistic regression model and the linear regression model.

Metrics	Logistic	Linear
PofB@20%	**0.308**	0.284
PMI@20%	0.190	**0.180**
PofB/ PMI@20%	2.222	**3.381**
Precision@20%	0.522	**0.561**
Recall@20%	**0.274**	0.234
IFA	3.283	**2.282**
Popt	**0.608**	0.602

Precision@20%, and IFA values, there is no significant difference between these two models. Since the primary objective of EADP is to find more bugs, we choose the logistic regression model for MOOAC.

(2) The training dataset of logistic regression model.

We compare two datasets in terms of the seven performance metrics to investigate the better one (i.e., dataset S_d only containing defective modules and dataset S containing all modules). Fig. 12 shows the performance distribution of MOOAC trained on the two training datasets across all testing datasets, and Table 12 presents the average values across all testing datasets.

As shown in Fig. 12 and Table 12, MOOAC trained on S_d achieves the best average values in terms of all seven metrics. But there is no significant difference on all metrics. On the other hand, considering that the NSGA-II algorithm employed by MOOAC is time-consuming in obtaining the optimal solution of the logistic regression model, utilizing a smaller training dataset S_d with fewer modules can effectively reduce the training time. Therefore, we choose S_d as the training dataset to build the logistic regression model for MOOAC.

(3) The parameter selection strategy for logistic regression model using the NSGA-II algorithm.

Since we aim to find more bugs and inspect as fewer modules as possible, we choose the coefficient vector in the Pareto optimal set that achieves the best PofB/PMI@20% value among the top 10 coefficient vectors that achieve the best PofB@20% value in MOOAC. To verify the effectiveness of the strategy, we compare it with the two selection strategies that choose the coefficient vector in the Pareto optimal set that achieves the best PofB@20% and PMI@20% values in the top 10

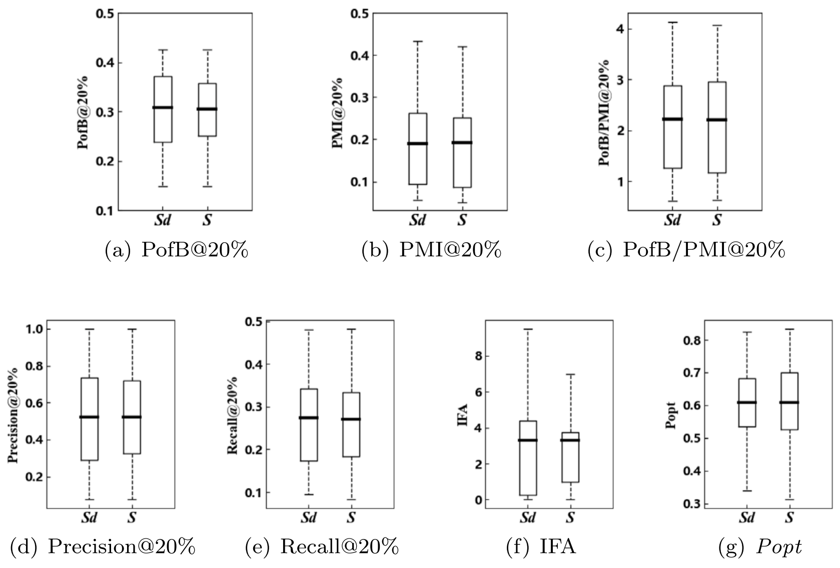


Fig. 12. The boxplots of PofB@20%, PMI@20%, PofB/PMI@20%, Precision@20%, Recall@20%, IFA, and *Popt* of MOOAC trained on S_d and S.

Table 12
The average PofB@20%, PMI@20%, PofB/PMI@20%, Precision@20%, Recall@20%, IFA, and *Popt* values of MOOAC trained on S_d and S (We keep four decimals of Precision@20% and *Popt* values to show the difference).

Metrics	S_d	S
PofB@20%	**0.308**	0.304
PMI@20%	**0.190**	0.192
PofB/ PMI@20%	**2.222**	2.205
Precision@20%	**0.5225**	0.5224
Recall@20%	**0.274**	0.271
IFA	**3.283**	3.317
Popt	**0.6084**	0.6080

Table 13
The average PofB@20%, PMI@20%, PofB/PMI@20%, Precision@20%, Recall@20%, IFA, and *Popt* values of MOOAC using the three selection strategies.

Metrics	Strategy 1	Strategy 2	Strategy 3
PofB@20%	**0.308**	0.300	0.301
PMI@20%	**0.190**	0.195	0.197
PofB/ PMI@20%	**2.222**	2.120	2.082
Precision@20%	**0.522**	0.508	0.513
Recall@20%	**0.274**	0.269	0.262
IFA	3.283	3.233	**3.067**
Popt	0.608	0.605	**0.609**

coefficient vectors that achieve the best PofB@20% value in terms of seven performance metrics. We call the above three selection strategies Strategy 1, Strategy 2, and Strategy 3, respectively. Fig. 13 shows the performance distribution of MOOAC with the three strategies across all testing datasets, and Table 13 presents the average value across all testing datasets.

As shown in Fig. 13 and Table 13, Strategy 1 achieves the best average value on PofB@20%, PMI@20%, PofB/PMI@20%, Precision@20%, and Recall@20%. Strategy 3 achieves the best average value on IFA and *Popt*. In addition, there is no significant difference between the three selection strategies on all metrics. The potential reason is the top 10 coefficient vectors that achieve the best PofB@20% value are already the Pareto optimal solutions, so picking one from them does not significantly affect the performance of MOOAC. Since the main objective is to find more bugs and inspect as fewer modules as possible, we choose Strategy 1 as the selection strategy for MOOAC.

Answer to RQ5: The strategy of using logistic regression model to calculate the defect probability for defective modules, training MOOAC on S_d containing only defective modules, and selecting the coefficient vector in Parteo optimal set that achieves the highest PofB/PMI@20% value among the 10 highest PofB@20% coefficient vectors help MOOAC obtain better performance.

5. Threats of validity

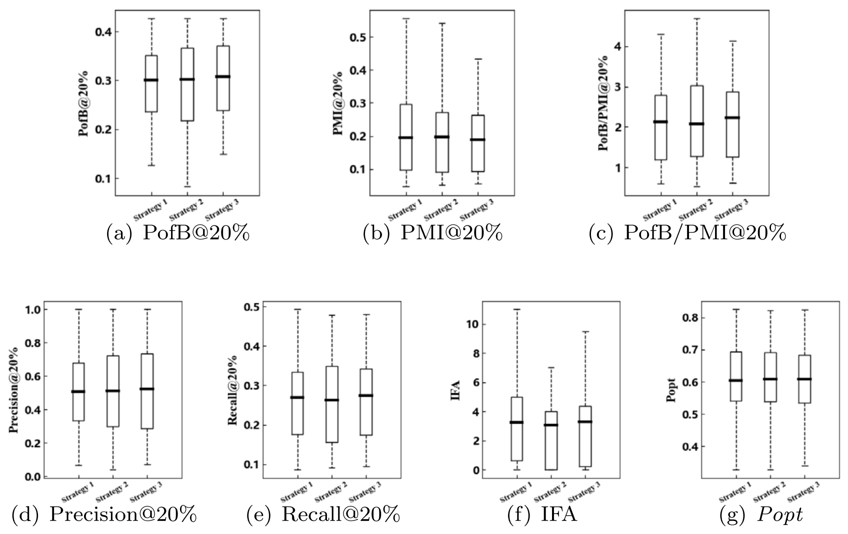
In this section, we discuss the threats of validity of our study.

(1) For our experimental analysis, we select the PROMISE repository as the dataset source due to its inclusion of multiple software releases and associated bug information. This choice allows us to conduct the cross-version validation that is more practical for the evaluation of EADP methods. Though the PROMISE datasets are commonly utilized in EADP studies [19,52], we could not affirm that MOOAC still performs the best on other defect datasets. Therefore, we will verify the generality of our method on more datasets in the future.

(2) In this study, all 19 software features (excluding LOC), are utilized to construct the MOOAC model. Previous empirical works [63–66] have demonstrated that the application of feature selection methods can improve the performance of CBDP models. However, studies investigating feature selection for EADP models are scarce. The potential improvement of the performance of EADP models through feature selection remains an open question. As a result, we decide against implementing feature selection, like the study conducted by [19].

(3) The parameters for the baseline methods are primarily determined based on previous literature or default settings. In our study, we set the desired defective ratio of RUS as 0.5, and the classification threshold for distinguishing defective and clean modules to 0.5 although we discuss it in *RQ4*, as they are the common practice. In addition, the NSGA-II algorithm in MOOAC is implemented using the "geatpy"[1] package, and the default algorithm parameters of the package are used. Nevertheless, it is crucial to acknowledge that the optimal parameters for a specific method and dataset may differ. Therefore, exploring the influence of parameters on the performance of EADP methods represents a potential avenue for future investigation.

[1] https://github.com/geatpy-dev/geatpy.



(a) PofB@20% (b) PMI@20% (c) PofB/PMI@20%

(d) Precision@20% (e) Recall@20% (f) IFA (g) *Popt*

Fig. 13. The boxplots of PofB@20%, PMI@20%, PofB/PMI@20%, Precision@20%, Recall@20%, IFA, and *Popt* of MOOAC using the three strategies.

(4) Randomness in experiments may affect the performance of EADP methods. Due to the NSGA-II algorithm, there is a certain randomness in the results of MOOAC. In the *RQ2*, we apply RUS for each EADP method. We cannot guarantee the same result in each run, because RUS deletes clean modules randomly. Therefore, the entire procedure is repeated 50 times, and the median value of the 50 results is considered as the final outcome for each dataset.

(5) Indeed, the definition of "effort" in SDP remains a continuous controversy. Several software developers have raised the observation that estimating the actual effort required to review a software module may not be adequately captured solely by the count of LOC [18]. It is essential to consider additional factors, such as the complexity and nature of software modules, when assessing the actual effort involved in inspecting them. Therefore, we recommend that future researches undertake empirical investigations into the factors influencing inspection effort and how to assign weights to different factors to establish a standardized effort for all instances. Furthermore, in practical situations, more factors than just the frequency of module switching may contribute to the additional cost of software testing, such as the number of dependencies of a module and the distance between modules in the file system. Some software testers may be more concerned about the additional cost caused by the above factors rather than the frequent switch between detected modules. In other words, there may be different goals in software testing in different projects. In this case, one of the optimization objective in our work can be changed from the number of modules to other factors. MOOAC provides a universal software module ranking method that satisfies multiple objectives for detection priority in software testing. Therefore, we recommend that developers select the multiple optimization goals to build the most suitable model based on the specific requirements, schedule, and resources of their project.

6. Related work

6.1. EADP

Mende et al. [16] and Kamei et al. [14] first introduced the concept of "effort-aware" into the field of SDP and pointed out that the objective of the EADP technique is to detect more bugs within the same LOC compared with the CBDP technique. Kamei et al. [23] predicted the defect density of code changes using the linear regression algorithm and ranked changes based on the predicted values. Yang et al. [25] observed that the unsupervised method ManualUp can achieve a higher Recall@20% value compared with the supervised method EALR for change-level EADP. However, Yan et al. [59] observed that the findings of Yang et al. [25] are inconsistent for cross-project file-level EADP. Huang et al. [18] pointed out that software testers would encounter lots of initial false alarms (i.e., the high IFA value) and must inspect many software modules (i.e., the high PMI@20% value) according to the rankings of ManualUp. Therefore, they proposed the CBS+ method, and the results showed that CBS+ could find 15%–26% more defective modules than EALR and reduce the PMI@20% and IFA values compared with ManualUp. Yang et al. [54] proposed an EADP method based on differential evolution algorithm. Li et al. [21] proposed the EATT semi-supervised method, which adopted a greedy strategy to prioritize code changes. Ni et al. [19,20] indicated the superiority of CBS+ for cross-project EADP and just-in-time EADP on JavaScript projects, respectively. The above-mentioned studies regarded EADP as a single objective optimization problem, and focused on finding more bugs when testing a certain number of LOC. Zhao et al. [53] proposed a compositional model for EADP on android apps. Yu et al. [67] improved EADP by directly learning to rank modules. Bennin et al. [68] and Yu et al. [55] studied the best EADP modeling algorithms. Bennin et al. [68] and Li et al. [22] investigated the effects of data imbalance and feature selection for EADP model respectively. Chen et al. [39] proposed MULTI, an EADP method based on multi-objective optimization. The approach utilized logistic regression to build a model and employed the NSGA-II algorithm to generate optimal solutions for model parameters. The optimization objectives include maximizing the number of detected defects (i.e., PofB) while minimizing the inspected LOC. However, considering that software testers need to inspect a specific amount of LOC (e.g., 20% LOC), the primary objective of the method remains focused on identifying more bugs.

In summary, the existing EADP methods ignore the number of required inspected modules, and the frequent switches between different modules also increase the testing cost. Therefore, we propose the MOOAC method with the multi-objectives of finding more bugs and inspecting as fewer modules as possible.

6.2. Multi-objective optimization for CBDP

Recently, some researchers considered CBDP as a multi-objective optimization problem. Ryu et al. [69] proposed a multi-objective naive Bayes method for cross-project CBDP, and there are three optimization objectives, i.e., the Probability of Detection (PD), Probability of False alarm (PF), and Balance. PD and PF are two common metrics in multi-objective CBDP. Chen et al. [70,71] proposed a feature selection method for CBDP with the optimization objectives of minimizing the number of selected features and maximizing the CBDP model performance. Niu et al. [72] proposed a multi-objective oriented cuckoo search to optimize several objects simultaneously to improve the SDP model accuracy. Cao et al. [73] set the PD and PF as the optimization objectives, and proposed a CBDP model based on twin support vector machines. Ni et al. [71] proposed a feature selection method based on multi-objective optimization for CBDP. The primary objective is to minimize the number of selected features in order to reduce detection costs. The secondary objective is to maximize the HyperVolume (HV), which is a measure of the volume covered by the Pareto front in the objective space. This objective aims to improve the performance of the constructed models. Cai et al. [74] utilized a hybrid multi-objective cuckoo search algorithm to develop a CBDP model using Support Vector Machine (SVM). Their approach addresses the challenges of class imbalance in datasets and the selection of optimal parameters for SVM. The objectives of the cuckoo search algorithm in this context are to optimize the PF and PD of the CBDP model. Zhang et al. [75] utilized the NSGA-II algorithm to choose the fewest features and minimize the classification error of CBDP models. Kanwar et al. [76] and Ye et al. [77] proposed a CBDP model based on multi-objective optimization to minimize the PF and to maximize the PD.

However, the above studies only focused on optimizing the classification performance of SDP models. In our study, we propose a multi-objective optimization-based method for EADP, which aims to maximize the found bugs and minimize the required inspected modules simultaneously when testing a certain amount of LOC.

7. Conclusion

The EADP technique can help software testers to assign testing resources more efficiently by suggesting testers inspect software modules with high defect density first. The previous studies [14,17] point out that the main objective of EADP is finding more bugs when testing a certain number of LOC. However, the existing EADP methods ignore that inspecting more modules and frequently switching between different modules also increase the testing cost. Therefore, we argue that the main objective of EADP should be not only finding more bugs but also inspecting as fewer modules as possible, when testing a certain amount of LOC.

In this paper, we propose a multi-objective effort-aware defect prediction approach named MOOAC for EADP, which aims to maximize the PofB@20% value and minimize the PMI@20% value while checking the top 20% LOC. In addition, MOOAC employs the random forest classifier to distinguish defective modules and clean ones to reduce the IFA value in the prediction phase. We perform the more practical cross-version validation in the experiments and evaluate MOOAC against the five existing EADP methods. In addition, we discuss the internal factors that affect the performance of MOOAC. The experimental results in our study include the following. (1) ManualUp, EATT, and NSGA-II tend to rank modules with fewer LOC first, so their IFA values are greater than 10. Except for the three methods, MOOAC achieves the best overall performance in terms of PofB@20%, PMI@20%, PofB/PMI@20%, and IFA. (2) Applying RUS to the defect datasets cannot improve the performance of MOOAC. (3) Using random forest as the underlying classifier of MOOAC can achieve the highest PofB@20%, Recall@20%, and *Popt* values among all the selected classifiers. (4) MOOAC suggest setting the defective threshold as 0.5 due to its best performance on PofB@20% and good performance on other metrics, which is also widely used in practice. Furthermore, to fit to the diverse needs of different situations across different performance metrics, we recommend setting the threshold value in the range of 0.2 to 0.8. (5) Using the logistic regression model to build the relationship between the defect density and software features, training MOOAC on only defective modules, and choosing the coefficient vector in the Pareto optimal set that achieves the best PofB/PMI@20% value can help MOOAC achieve better performance.

As a whole, our MOOAC method can ensure to find more bugs and inspect as fewer modules as possible simultaneously. In addition, the initial false alarm of the ranking provided by MOOAC is low, thus guaranteeing to detect actual defective modules as soon as possible. Therefore, MOOAC is recommended as an effective method for EADP.

CRediT authorship contribution statement

Xiao Yu: Data curation, Formal analysis, Methodology, Writing – original draft. **Liming Liu:** Data curation, Methodology, Software, Writing – original draft. **Lin Zhu:** Formal analysis, Software, Writing – review & editing. **Jacky Wai Keung:** Project administration, Supervision, Validation. **Zijian Wang:** Resources, Software, Writing – original draft. **Fuyang Li:** Formal analysis, 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.

Data availability

Data will be made available on request.

Acknowledgments

This work is supported in part by the National Natural Science Foundation of China (62272356), the National Natural Science Foundation of China of Chongqing City (cstc2021jcyj-msxmX1115), and the Start-up Grant from Wuhan University of Technology (104-40120693).

References

[1] 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.
[2] 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.
[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. 152 (2022) 107057.
[4] H. Chen, X.-Y. Jing, Y. Zhou, B. Li, B. Xu, Aligned metric representation based balanced multiset ensemble learning for heterogeneous defect prediction, Inf. Softw. Technol. 147 (2022) 106892.
[5] Z. Sun, J. Li, H. Sun, L. He, CFPS: Collaborative filtering based source projects selection for cross-project defect prediction, Appl. Soft Comput. 99 (2021) 106940.
[6] 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.
[7] 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.
[8] 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.
[9] Z. Zhang, Y. Lei, T. Su, M. Yan, X. Mao, Y. Yu, Influential global and local contexts guided trace representation for fault localization, ACM Trans. Softw. Eng. Methodol. 32 (3) (2023) 78:1–78:27.
[10] 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.
[11] Y. Gao, Y. Zhu, Y. Zhao, Dealing with imbalanced data for interpretable defect prediction, Inf. Softw. Technol. 151 (2022) 107016.
[12] S. Stradowski, L. Madeyski, Industrial applications of software defect prediction using machine learning: A business-driven systematic literature review, Inf. Softw. Technol. 159 (2023) 107192, http://dx.doi.org/10.1016/j.infsof.2023.107192.
[13] Z. Sun, J. Zhang, H. Sun, X. Zhu, Collaborative filtering based recommendation of sampling methods for software defect prediction, Appl. Soft Comput. 90 (2020) 106163.

[14] 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.

[15] F. Li, P. Yang, J.W. Keung, W. Hu, H. Luo, X. Yu, Revisiting 'revisiting supervised methods for effort-aware cross-project defect prediction', IET Softw. 17 (4) (2023) 472–495.

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

[17] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, A. Bener, Defect prediction from static code features: current results, limitations, new approaches, Autom. Softw. Eng. 17 (4) (2010) 375–407.

[18] Q. Huang, X. Xia, D. Lo, Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction, Empir. Softw. Eng. 24 (5) (2019) 2823–2862.

[19] 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. 48 (3) (2022) 786–802.

[20] 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. 31 (4) (2022) 1–38.

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

[22] 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. 17 (2) (2023) 168–193.

[23] 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 (6) (2012) 757–773.

[24] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput. 6 (2) (2002) 182–197.

[25] Y. Yang, Y. Zhou, H. Lu, L. Chen, Z. Chen, B. Xu, H. Leung, Z. Zhang, Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction? An empirical study, IEEE Trans. Softw. Eng. 41 (4) (2014) 331–357.

[26] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, B. Turhan, The Promise Repository of Empirical Software Engineering Data, West Virginia University, Department of Computer Science, 2012, URL: http://promisedata.org/repository.

[27] 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 (2) (2022) 185–199.

[28] S. Feng, J. Keung, P. Zhang, Y. Xiao, M. Zhang, The impact of the distance metric and measure on SMOTE-based techniques in software defect prediction, Inf. Softw. Technol. 142 (2022) 106742.

[29] S.K. Pandey, A.K. Tripathi, An empirical study toward dealing with noise and class imbalance issues in software defect prediction, Soft Comput. 25 (21) (2021) 13465–13492.

[30] 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 (1) (2019) 139–153.

[31] 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.

[32] W. Zheng, T. Shen, X. Chen, P. Deng, Interpretability application of the Just-in-Time software defect prediction model, J. Syst. Softw. 188 (2022) 111245.

[33] C. Tantithamthavorn, A.E. Hassan, K. Matsumoto, The impact of class rebalancing techniques on the performance and interpretation of defect prediction models, IEEE Trans. Softw. Eng. 46 (11) (2020) 1200–1219.

[34] D. Bowes, T. Hall, J. Petrić, Software defect prediction: do different classifiers find the same defects? Softw. Qual. J. 26 (2) (2018) 525–552.

[35] B. Ghotra, S. McIntosh, A.E. Hassan, Revisiting the impact of classification techniques on the performance of defect prediction models, in: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1, IEEE Computer Society, 2015, pp. 789–800.

[36] C. Tantithamthavorn, S. McIntosh, A.E. Hassan, K. Matsumoto, The impact of automated parameter optimization on defect prediction models, IEEE Trans. Softw. Eng. 45 (7) (2018) 683–711.

[37] P.S. Kochhar, X. Xia, D. Lo, S. Li, Practitioners' expectations on automated fault localization, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, 2016, pp. 165–176.

[38] C. Parnin, A. Orso, Are automated debugging techniques actually helping programmers? in: Proceedings of the 2011 International Symposium on Software Testing and Analysis, 2011, pp. 199–209.

[39] 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.

[40] 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.

[41] H.Q. Awla, S.W. Kareem, A.S. Mohammed, A comparative evaluation of Bayesian networks structure learning using falcon optimization algorithm, Int. J. Interact. Multimedia Artif. Intell. 8 (2) (2023) 81.

[42] Q. Ding, X. Xu, Improved GWO algorithm for UAV path planning on crop pest monitoring, Int. J. Interact. Multimed. Artif. Intell. 7 (5) (2022) 30.

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

[44] Y. Chen, X. Lu, S. Wang, Deep cross-modal image–voice retrieval in remote sensing, IEEE Trans. Geosci. Remote Sens. 58 (10) (2020) 7049–7061.

[45] 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.

[46] C. He, J. Wu, Q. Zhang, Characterizing research leadership on geographically weighted collaboration network, Scientometrics 126 (5) (2021) 4005–4037.

[47] C. He, J. Wu, Q. Zhang, Proximity-aware research leadership recommendation in research collaboration via deep neural networks, J. Assoc. Inf. Sci. Technol. 73 (1) (2022) 70–89.

[48] 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 (19) (2021) 6417.

[49] F. Li, K. Zou, J.W. Keung, X. Yu, S. Feng, Y. Xiao, On the relative value of imbalanced learning for code smell detection, Softw. - Pract. Exp. 53 (10) (2023) 1902–1927.

[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] Z. Yang, J.W. Keung, X. Yu, Y. Xiao, Z. Jin, J. Zhang, On the significance of category prediction for code-comment synchronization, ACM Trans. Softw. Eng. Methodol. 32 (2) (2023) 1–41.

[52] 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.

[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 (3) (2022) 259–278.

[54] 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 (03) (2021) 289–310.

[55] 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. 157 (2023) 107165.

[56] T.-D.B. Le, D. Lo, Beyond support and confidence: Exploring interestingness measures for rule-based specification mining, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, 2015, pp. 331–340.

[57] F. Rahman, C. Bird, P. Devanbu, Clones: What is that smell? Empir. Softw. Eng. 17 (2012) 503–530.

[58] 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.

[59] 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.

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

[61] J. Ferreira, A. Zwinderman, On the benjamini–hochberg method, Ann. Statist. 34 (4) (2006) 1827–1849.

[62] V.B. Kampenes, T. Dybå, J.E. Hannay, D.I. Sjøberg, A systematic review of effect size in software engineering experiments, Inf. Softw. Technol. 49 (11–12) (2007) 1073–1086.

[63] A.O. Balogun, S. Basri, S. Mahamad, S.J. Abdulkadir, M.A. Almomani, V.E. Adeyemo, Q. Al-Tashi, H.A. Mojeed, A.A. Imam, A.O. Bajeh, Impact of feature selection methods on the predictive performance of software defect prediction models: an extensive empirical study, Symmetry 12 (7) (2020) 1147.

[64] B. Ghotra, S. McIntosh, A.E. Hassan, A large-scale study of the impact of feature selection techniques on defect classification models, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), IEEE, 2017, pp. 146–157.

[65] C. Ni, X. Chen, F. Wu, Y. Shen, Q. Gu, An empirical study on pareto based multi-objective feature selection for software defect prediction, J. Syst. Softw. 152 (JUN.) (2019) 215–238.

[66] K. Thirumoorthy, et al., A feature selection model for software defect prediction using binary Rao optimization algorithm, Appl. Soft Comput. 131 (2022) 109737.

[67] X. Yu, J. Rao, W. Hu, J. Keung, J. Zhou, J. Xiang, Improving effort-aware defect prediction by directly learning to rank software modules, Inf. Softw. Technol. (2023) http://dx.doi.org/10.1016/j.infsof.2023.107250.

[68] 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.

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

[70] X. Chen, Y. Shen, Z. Cui, X. Ju, Applying feature selection to software defect prediction using multi-objective optimization, in: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), Vol. 2, IEEE, 2017, pp. 54–59.

[71] C. Ni, X. Chen, F. Wu, Y. Shen, Q. Gu, An empirical study on pareto based multi-objective feature selection for software defect prediction, J. Syst. Softw. 152 (2019) 215–238.

[72] Y. Niu, Z. Tian, M. Zhang, X. Cai, J. Li, Adaptive two-SVM multi-objective cuckoo search algorithm for software defect prediction, Int. J. Comput. Sci. Math. 9 (6) (2018) 547–554.

[73] Y. Cao, Z. Ding, F. Xue, X. Rong, An improved twin support vector machine based on multi-objective cuckoo search for software defect prediction, Int. J. Bio-Inspired Comput. 11 (4) (2018) 282–291.

[74] X. Cai, Y. Niu, S. Geng, J. Zhang, Z. Cui, J. Li, J. Chen, An under-sampled software defect prediction method based on hybrid multi-objective cuckoo search, Concurr. Comput.: Pract. Exper. 32 (5) (2020) e5478.

[75] N. Zhang, S. Ying, W. Ding, K. Zhu, D. Zhu, WGNCS: A robust hybrid cross-version defect model via multi-objective optimization and deep enhanced feature representation, Inform. Sci. 570 (2021) 545–576.

[76] S. Kanwar, L.K. Awasthi, V. Shrivastava, Efficient random forest algorithm for multi-objective optimization in software defect prediction, IETE J. Res. (2023) 1–13.

[77] T. Ye, W. Li, J. Zhang, Z. Cui, A novel multi-objective immune optimization algorithm for under sampling software defect prediction problem, Concurr. Comput.: Pract. Exper. 35 (4) (2023) e7525.