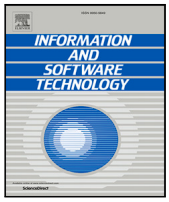
Information and Software Technology 157 (2023) 107165

Contents lists available at ScienceDirect

Information and Software Technology

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



Finding the best learning to rank algorithms for effort-aware defect prediction

Xiao Yu [a,b], Heng Dai [c], Li Li [d], Xiaodong Gu [e], Jacky Wai Keung [f], Kwabena Ebo Bennin [g], Fuyang Li [a], Jin Liu [h,*]

[a] School of Computer Science and Artificial Intelligence, Wuhan University of Technology, Wuhan, China
[b] Sanya Science and Education Innovation Park of Wuhan University of Technology, Sanya, China
[c] School of Mechanical and Electrical Engineering, Wuhan Qingchuan University, Wuhan, China
[d] School of Software, Beihang University, Beijing, China
[e] School of Software, Shanghai Jiao Tong University, Shanghai, China
[f] Department of Computer Science, City University of Hong Kong, Hong Kong, China
[g] Information Technology Group, Wageningen University and Research, Wageningen, Netherlands
[h] School of Computer Science, Wuhan University, Wuhan, China

ARTICLE INFO

Keywords:
Software defect prediction
Empirical study
Learning to rank
Ranking instability

ABSTRACT

Context: Effort-Aware Defect Prediction (EADP) ranks software modules or changes based on their predicted number of defects (i.e., considering modules or changes as effort) or defect density (i.e., considering LOC as effort) by using learning to rank algorithms. Ranking instability refers to the inconsistent conclusions produced by existing empirical studies of EADP. The major reason is the poor experimental design, such as comparison of few *learning to rank* algorithms, the use of small number of datasets or datasets without indicating numbers of defects, and evaluation with inappropriate or few metrics.
Objective: To find a stable ranking of *learning to rank* algorithms to investigate the best ones for EADP,
Method: We examine the practical effects of 34 algorithms on 49 datasets for EADP. We measure the performance of these algorithms using 7 module-based and 7 LOC-based metrics and run experiments under cross-release and cross-project settings, respectively. Finally, we obtain the ranking of these algorithms by performing the Scott-Knott ESD test.
Results: When module is used as effort, random forest regression performs the best under cross-release setting, and linear regression performs the best under cross-project setting among the *learning to rank* algorithms; (2) when LOC is used as effort, LTR-linear (Learning-to-Rank with the linear model) performs the best under cross-release setting, and Ranking SVM performs the best under cross-project setting.
Conclusion: This comprehensive experimental procedure allows us to discover a stable ranking of the studied algorithms to select the best ones according to the requirement of software projects.

1. Introduction

Software Defect Prediction (SDP) has typically been modeled by training a binary classifier using historical software data, and predicts whether or not a new software module is defective [1,2]. However, conventional SDP models based on the binary classification do not account for defect densities of various software modules [3,4]. Consequently, software modules that have different defect densities are allocated with the same amount of testing resources. Therefore, Mende et al. [3] and Kamei et al. [4] proposed the Effort-Aware Defect Prediction (EADP) model to rank software modules according to the predicted defect density. Specifically, they used the inspected Lines Of Code (LOC) as a proxy for effort. Recently, Yang et al. [5] proposed to rank software modules based on the predicted number of defects. They used the number of the inspected software modules as a proxy for effort. Therefore, in our study, we define EADP as a task of ranking software modules based on both the defect density (considering LOC as effort) and the number of defects (considering module as effort). The core component of EADP models is the learning to rank (henceforth, L2R) algorithm [5], which automatically constructs a ranking model using training data, such that the model can sort new software modules

* Corresponding author.
E-mail addresses: xiaoyu@whut.edu.cn (X. Yu), daiheng726@163.com (H. Dai), lilicoding@ieee.org (L. Li), xiaodong.gu@sjtu.edu.cn (X. Gu), jacky.keung@cityu.edu.hk (J.W. Keung), kwabena.bennin@wur.nl (K.E. Bennin), fyli@whut.edu.cn (F. Li), jinliu@whu.edu.cn (J. Liu).

https://doi.org/10.1016/j.infsof.2023.107165
Received 22 February 2022; Received in revised form 15 January 2023; Accepted 30 January 2023
Available online 2 February 2023

according to their defect quantity or density. A good L2R algorithm can precisely rank software modules with more defects or higher defect density first, allowing software testers to find more defects with less inspection effort. Hence, selecting the best L2R algorithm becomes the key to build successful EADP models.

Although there have been extensive studies on searching for the best L2R algorithms [6–13], different researchers often provide inconsistent rankings of L2R algorithms, i.e., there is no consensus on what is the "best" L2R algorithm for building the EADP model. Nguyen et al. [6] found that RankBoost had more stable prediction performance. Bennin et al. [7,8] discovered that Decision Tree Regression (DTR) performed the best in cross-release settings. Wang et al. [10] found that RankNet performed the best regarding Normalized Discounted Cumulative Gain (NDCG) in the scenario of cross-project EADP. Miletić et al. [11] found that Logistic Regression (LogR) gained the best results in the scenario of cross-release EADP. Yang et al. [12] found that Random Forest Regression (RFR) can achieve better results under cross-release setting. Yan et al. [9] found that ManualUp did not perform statistically significantly better than some classification models and Linear Regression (LR) under within-project setting, and can perform statistically significantly better than them under cross-project setting. Ni et al. [13] found that some supervised algorithms can statistically significantly outperform ManualUp for cross-project EADP. Such inconsistent findings make it hard to derive practical guidelines about which learning to rank algorithms should be employed to build EADP models, and Menzies et al. [14] and Keung et al. [15,16] refer to the inconsistent findings as the ranking instability problem. Therefore, we aim to empirically investigate the selection of L2R algorithms for EADP and find the best ones. As suggested by Menzies et al. [14] and Keung et al. [15,16], three experimental conditions should be carefully controlled for a stable ranking result: (1) Variants of datasets are sufficient to draw a conclusion; (2) The procedure to sample the training/test modules is logical; (3) The performance measures are sufficient in amount and are valid. The primary aim of EADP is to find more defective modules and defects and obtain the accurate global ranking of software modules according to the number of defects or defect density. It is also important to consider that inspecting too many modules and LOC may result in increased effort. Furthermore, developers may be hesitant to utilize an EADP model if false alarms are high. As such, these three factors should be considered when evaluating an EADP model. Last but not least, developers are reluctant to use a EADP model if the false alarms are high. Therefore, the three aspects should be taken into consideration to evaluate a EADP model. However, we observed that results of existing studies tend to be impacted, presumably without fully respecting these experimental conditions, such as comparison of few L2R algorithms on small number of datasets, and evaluation with inappropriate or few performance measures.

To address the aforementioned issues, we aim to find a stable ranking of various L2R algorithms that are widely used in existing EADP studies through a comprehensive experimental design. We investigate 34 algorithms, including 31 L2R algorithms, two unsupervised learning algorithms (i.e., ManualUp and ManualDown) and the OneWay algorithm on 49 module-level datasets. We also compare these algorithms with 7 module-based and 7 LOC-based effort-aware performance measures under cross-release and cross-project settings. For more robust result analysis, we use the Scott-Knott with Cohen's d effect size awareness (Scott-Knott ESD) [17] test to divide these algorithms into statistically distinct rankings. Compared with the existing empirical studies [6–13], we carefully control the three experimental conditions mentioned by Menzies and Keung et al. and evaluate more algorithms with a robust statistical test. This comprehensive experimental procedure allows us to draw stable conclusions about the performance of the algorithms.

The experimental results show that: (1) When using module as effort, ManualDown performs the best in terms of PofB@20%module (Proportion of the inspected Bugs when inspecting top 20% modules) and P_{opt}@module under cross-release and cross-project settings. However, ManualDown requires the inspection of significantly more LOC than other algorithms when inspecting top 20% modules. (2) When using module as effort, RFR (Random Forest Regression) performs only behind ManualDown under cross-release setting, and LR (Linear Regression) performs only behind ManualDown and OneWay under cross-project setting in terms of PofB@20%module and P_{opt}@module. But RFR and LR require software testers to inspect significantly less LOC than ManualDown and OneWay. (3) When using LOC as effort, LTR-linear (Learning-to-Rank with the linear model) performs the best under cross-release setting, and Ranking SVM performs the best under cross-project setting in terms of PofB@20%LOC (Proportion of the inspected Bugs when inspecting top 20% Lines Of Code) and P_{opt}@LOC among those algorithms with low IFA (Initial False Alarms encountered before we find the first defect) values.

This paper is an extended version of our study published in SANER [18]. The **extensions** include the following updates:

(1) We enhance the experimental settings by using the more practical cross-release and cross-project validations. In addition, we employ more evaluation measures, i.e., Precision, Recall, IFA, PLI (Proportion of LOC Inspected), PMI (Proportion of Modules Inspected), PofB, PofB/PLI (ratio between PofB and PLI), and PofB/PMI (ratio between PofB and PMI) to comprehensively evaluate the performance of EADP models. The performance measures allow us to evaluate a EADP model comprehensively from the above-mentioned three aspects. In summary, compared with our SANER paper, the paper carefully controls the last two experimental conditions suggested by Menzies et al. [14] and Keung et al. [15,16] so that we can find a stable ranking of L2R algorithms.

(2) We investigate the performance of more L2R algorithms, including simple logistic, radial basis functions network, logistic model tree, ripper, ripple down rules, random forest regression, least angle regression, multivariate adaptive regression splines, and K*, and compare the L2R algorithms with ManualUp, ManualDown, and OneWay. Compared with our SANER paper, we obtain additional experimental findings. Specifically, when module is considered as effort, ManualDown performs the best but requires the inspection of more LOC. Among the other models, RFR and LR achieve the best performance under cross-release and cross-project settings, respectively. When LOC is considered as effort, LTR-linear and Ranking SVM perform the best under cross-release and cross-project settings, respectively.

(3) We identify and analyze a set of 24 primary module-level EADP studies, and further discuss the differences between our work and previous studies.

(4) We give some implications based on our experimental results and provide a few suggestions to practitioners and researchers in the field of EADP.

Our **contributions** can be summarized as follows.

- We make a comprehensive comparison of 34 algorithms for EADP using 7 module-based and 7 LOC-based performance measures under cross-release and cross-project settings. To the best of our knowledge, this is the first large-scale study of finding a stable ranking of L2R algorithms for EADP.
- We identify and analyze a set of 24 primary studies related to module-level EADP published until 2022 from different perspectives, including the used datasets, L2R algorithms, evaluation measures, and ranking criterion. Community can use this set as a starting point to conduct further research on EADP. To the best of our knowledge, this is the first work to conduct a systematic literature review of EADP studies.
- We provide suggestions and guidelines based on our findings to encourage further research in the field of EADP.

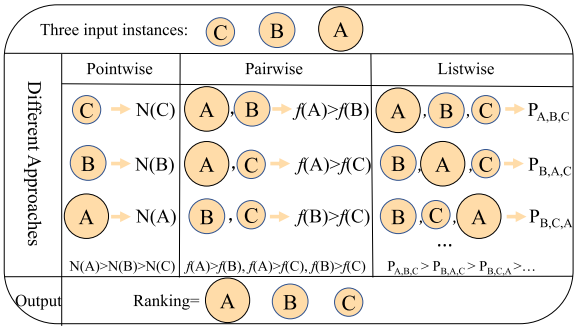


Fig. 1. An overview of the three types of L2R algorithms.

2. Background

Let $\boldsymbol{m}_i = (\boldsymbol{x}_i, y_i)$ denote a software module, where $\boldsymbol{x}_i = (x_1, x_2, \ldots, x_m)$ is a m-dimensional software feature vector of the ith module, and y_i is the number of defects of the corresponding module. A software module dataset containing both defective and non-defective modules can be represented as:

$$S = \{\boldsymbol{m}_1, \boldsymbol{m}_2, \ldots, \boldsymbol{m}_n\}, \tag{1}$$

where n is the number of modules in S. The goal of EADP is to train a prediction model from S in order to rank new modules according to the predicted number of defects or defect density. In the following, we briefly introduce the studied 34 algorithms, including 31 L2R algorithms, two unsupervised learning algorithms, and the OneWay algorithm. The 31 L2R algorithms cover three families, including 24 pointwise L2R algorithms (12 classification-based pointwise algorithms and 12 regression-based pointwise algorithms), 4 pairwise algorithms, and 3 listwise algorithms. Fig. 1 shows an overview of the three types of L2R algorithms, and Table 1 provides an overview of the 31 L2R algorithms. Following Tantithamthavorn et al. [19], when the candidate parameter values of the learning to algorithms are not unique, we use the grid search to tune parameters of the L2R algorithms to maximize the P_{opt} value, because P_{opt} evaluates the global ranking of EADP models. We set the number of iterations of SL, LMT, RankBoost, RankNet, LambdaRank, ListNet, and Coordinate Ascent as 500, 500, 300, 100, 100, 1500, and 25, since our preliminary experiments showed that more iterations of these algorithms did not result in significant improvement. Following You et al. [20] and Yang et al. [5], we set the feasible solution space to [−20,20], and set both population size and maximal generation to 100 for GP and LTR.

2.1. The pointwise algorithm

The pointwise algorithm ranks software modules by directly predicting the number of defects or defect density [21]. As shown in Fig. 1, there are three modules (i.e., A, B, and C) that need to be ranked. Assuming that the pointwise approach predicts that the number of defects in module A is N(A), the number of defects in module B is N(B), the number of defects in module C is N(C), and N(A)>N(B)>N(C), then the predicted ranking is A>B>C. According to different machine learning technologies used, the pointwise algorithm can be further divided into two subcategories: classification-based algorithms and regression-based algorithms [21]. The classification-based pointwise algorithm uses the classification technique to predict the possibility of a module being defective, and regards the possibility as the number of defects, while the regression-based algorithm uses the regression technique to directly predict the number of defects or defect density.

(1) Naive Bayes (NB) [22]: It is a classification algorithm based on the Bayes' theorem with the "naive" assumption that every pair of software features are independent.

(2) Logistic Regression (LogR) [23]: It is a classification algorithm to classify software modules into discrete outcomes. It maximizes the entropy of the labels conditioned on the software features with respect to the distribution.

(3) Simple Logistic (SL) [24]: It is a classifier for building linear logistic regression models. It uses LogitBoost [25] with simple regression functions as base learners for fitting the logistic models.

(4) Radial Basis Functions Network (RBFNet) [26]: It is an artificial neural network that uses radial basis functions as the activation functions. RBFNet's output is a linear combination of radial basis functions of the inputs and neuron parameters.

(5) Sequential Minimal Optimization (SMO) [27]: It is a support vector machines (SVM) classifier, which uses sequential minimal optimization to solve the quadratic programming (QP) problem during the training.

(6) Classification and Regression Tree (CART) [28]: It is a decision tree classifier, which partitions the training dataset into small segments using the Gini index, and labels these small segments with one of the class labels (i.e., defective or non-defective).

(7) C4.5 [29]: It is also a decision tree classifier, which partitions the training dataset into small segments using the information gain rather than the Gini index in CART.

(8) Logistic Model Tree (LMT) [24]: It is a classification model with an associated training algorithm that combines logistic regression and decision tree learning. It produces a logistic regression model at every node in the tree using the LogitBoost algorithm [25], and splits the node using the information gain.

(9) Random Forest (RF) [30]: It is an ensemble classifier that fits a number of decision tree classifiers on various subsets of the original dataset, and uses averaging to improve the predictive accuracy and control over-fitting.

(10) K-nearest Neighbors (KNN) [31]: It finds k training software modules closest to the new software module, and predicts the label of the new software module from these training modules.

(11) Repeated Incremental Pruning to Produce Error Reduction (Ripper) [32]: It is a propositional rule based algorithm that creates series of rules with pruning to remove the rules that lead to lower classification performance.

(12) Ripple Down Rules (Ridor) [33]: It is a rule-based decision tree algorithm, where the decision tree consists of four nodes: a classification, a predicate function, a true branch, and a false branch.

(13) Decision Tree Regression (DTR) [34]: It fits a regression model in the form of a decision tree structure by learning from the training dataset.

(14) Random Forest Regression (RFR) [35]: It is an ensemble regression model that fits a number of decision tree regression models on various subsets of the original dataset, and uses averaging to improve the predictive accuracy and control over-fitting.

(15) Linear Regression (LR) [36]. It trains a linear model:

$$y = f(\boldsymbol{b}, \boldsymbol{x}) = b_0 + b_1 x_1 + b_2 x_2 + \cdots + b_m x_m, \tag{2}$$

where $\boldsymbol{b} = (b_0, b_1, \ldots, b_m)$ represents a $(m+1)$-dimensional vector of regression coefficients, $\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$ is a m-dimensional software feature vector of the module, and y is the predicted number of defects or defect density of the module. LR uses the least square method to find the optimal $\boldsymbol{b}$ value by minimizing the following loss function:

$$\sum_{i=1}^{n} (y_i - f(\boldsymbol{b}, \boldsymbol{x}_i))^2, \tag{3}$$

where y_i is the actual number of defects or defect density of the ith module, and $\boldsymbol{x}_i$ is the m-dimensional software feature vector of the ith module.

(16) Ridge Regression (RR) [37]: It trains the same linear model as LR, but uses the gradient descent approach to find the optimal $\boldsymbol{b}$ value by minimizing the following loss function:

$$\sum_{i=1}^{n} (y_i - f(\boldsymbol{b}, \boldsymbol{x}_i))^2 + \lambda |\boldsymbol{b}|^2, \tag{4}$$

Table 1
Parameter value overview of the L2R algorithms.

Family		Label	Algorithm[a]	Parameter description[b]	Candidate parameter values (Default value is in bold)
Classification based Pointwise Approach	Bayes	C1	Naïve Bayes (NB)	[N] Laplace Correction (0 indicates no correction).	**{0}**
	Function	C2	Logistic Regression (LogR)	[N] Tolerance for stopping criteria.	{0.1,0.01,0.001,**0.0001**,0.00001}
		C3	Simple Logistic (SL)	[N] The number of iterations for LogitBoost.	**{500}**
		C4	Radial Basis Functions Network (RBFNet)	[N] Ridge factor for quadratic penalty on output weights.	{0.1,**0.01**,0.001, 0.0001, 0.00001}
		C5	Sequential Minimal Optimization (SMO)	[N] Penalty parameter of the error term with linear kernel.	{**1**}
	Tree	C6	Classification and Regression Tree (CART)	[N] The minimum number of samples required to split an internal node.	{**2**,3,4,5,6}
		C7	C4.5	[N] The confidence factor that are used for pruning (smaller values incur more pruning).	{**0.25**}
		C8	Logistic Model Tree (LMT)	[N] The number of iterations for LogitBoost.	{**500**}
		C9	Random Forest (RF)	[N] The number of trees in the forest.	{**10**,20,30,40,50}
	Lazy	C10	K-Nearest Neighbors (KNN)	[N] The number of neighbors.	{1,**5**,9,13,17}
	Rule	C11	Repeated Incremental Pruning to Produce Error Reduction (Ripper)	[N] The number of runs of optimizations.	{1,**2**,3,4,5}
		C12	Ripple Down Rules (Ridor)	[N] The number of shuffles to randomize the data in order to get better rule.	{2,6, **10**,14,18}
Regression based Pointwise Approach	Tree	R1	Decision Tree Regression (DTR)	[N] The minimum number of samples required to split an internal node.	{**2**,3,4,5,6}
		R2	Random Forest Regression (RFR)	[N] The number of trees in the forest.	{**10**,20,30,40,50}
	Linear	R3	Linear Regression (LR)	[L] Whether the regressor will be normalized before.	{true, **false**}
		R4	Ridge Regression (RR)	[N] Stop the algorithm if whas converged.	{0.1,0.01,**0.001**,0.0001,0.00001}
		R5	Least Angle Regression (LAR)	[L] Whether the regressor will be normalized before.	{**true**, false}
		R6	Genetic Programming (GP)	[C] Feasible solution space.	[-20,20]
				[N] Population size and maximal generation.	{**100**}
	Nolinear	R7	Neural Network Regression (NNR)	[N] The number of neurons in the hidden layers.	{4,8,16,32,64,**100**}
				[N] L2 penalty (regularization term) parameter.	{0, **0.0001**, 0.001, 0.01, 0.1}
		R8	Support Vector Regression (SVR)	[N] Penalty parameter of the error term.	{0.25, 0.5,**1**, 2, 4}
		R9	Relevance Vector Machine (RVM)	[N] Penalty parameter of the error term with Linear kernel.	{**1**}
		R10	Multivariate Adaptive Regression Splines (MARS)	[N] Penalty parameter of the error term. The maximum degree of interaction (Friedman's mi). The default is 1, meaning build an additive model (i.e., no interaction terms).	{**1**}
	Lazy	R11	K-nearest Neighbors Regression (KNR)	[N] The number of neighbors.	{1,**5**,9,13,17}
		R12	K*	N/A	N/A
Pairwise Approach		P1	Ranking SVM	[N] Penalty parameter of the error term with linear kernel.	{**1**}
		P2	RankBoost	[N] The number of rounds to train.	{**300**}
		P3	RankNet	[N] The number of epochs to train.	{**100**}
		P4	LambdaRank	[N] The number of epochs to train.	{**100**}
Listwise Approach		L1	ListNet	[N] The number of epochs to train.	{**1500**}
		L2	Coordinate Ascent	[N] The number of iterations to search in each direction.	{**25**}
		L3	Learning-to-Rank (LTR)	[C] Feasible solution space.	[-20,20]
				[N] Population size and maximal generation.	{**100**}

[a]We implement NB, LogR, CART, RF, KNN, DTR, RFR, LR, RR, LAR, GP, NNR, SVR, RVM, MARS, KNR, Ranking SVM and LTR based on *sklearn* (http://scikit-learn.github.io/stable). We implement SL, RBFNet, SMO, C4.5, LMT, Ripper, Ridor, and K* based on *Weka* (https://www.cs.waikato.ac.nz/~ml/weka/). We implement RankBoost, RankNet, LambdaRank, ListNet, Coordinate Ascent based on an open-source library of popular L2R algorithms *RankLib*(https://sourceforge.net/p/lemur/wiki/RankLib/).
[b][N] denotes a numeric value; [L] denotes a logical value; [C] denotes the continuous space.

where λ is a small regularization parameter.

(17) Least Angle Regression (LAR) [38]: It trains the same linear model as LR, but uses the least angle regression approach to find the optimal $\boldsymbol{b}$ value by minimizing the following loss function:

$$\sum_{i=1}^{n}(y_i - f(\boldsymbol{b}, \boldsymbol{x}_i))^2 + \lambda|\boldsymbol{b}|. \quad (5)$$

(18) Genetic Programming (GP) [39]: It trains the same linear model as LR, but uses the genetic algorithm to find the optimal $\boldsymbol{b}$ value by minimizing the following loss function:

$$\sum_{i=1}^{n}(y_i - f(\boldsymbol{b}, \boldsymbol{x}_i))^2. \quad (6)$$

(19) Neural Network Regression (NNR) [40]: It learns a non-linear function approximator using backpropagation with no activation function in the output layer.

(20) Support Vector Regression (SVR) [41]: It uses the same principles as SVM, with only a few minor differences. Because the output of SVR is a real number, it is difficult to predict the information at hand, which has infinite possibilities. In the case of regression, a margin of tolerance is set in approximation to the SVM which would have already requested from the problem.

(21) Relevance Vector Machine (RVM) [42]: It has an identical functional form to SVR, but uses Bayesian inference to obtain parsimonious solutions for regression.

(22) Multivariate Adaptive Regression Splines (MARS) [43]: It is a nonparametric regression algorithm based on the divide and conquer strategy, in which the training data is partitioned into separate piecewise linear segments.

(23) K-nearest Neighbors Regression (KNR) [44]: It finds k training software modules that are the nearest to the new software module based on the Euclidean distance, and predicts the number of defects or defect density of the new software module based on the mean of those of these nearest neighbors.

(24) K* [45] is similar to KNR, except that it uses an entropy-based distance function to calculate the distance between the new module and the training modules.

2.2. The pairwise algorithm

The pairwise approach views EADP problem as a classification task, i.e., learning a binary classifier f that can identity which module contains more defects or has higher defect density in a given module pair [6]. As shown in Fig. 1, assuming that the pairwise approach predicts that module A contains more defects than module B (i.e., $f(A)>f(B)$), module A contains more defects than module C (i.e., $f(A)>f(C)$), and module B contains more defects than module C (i.e., $f(B)>f(C)$), then the predicted ranking becomes A>B>C.

(1) Ranking SVM [46]: It first transforms the ranking problem into classification by computing $\boldsymbol{x}_1$ - $\boldsymbol{x}_2$, where $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ are the feature vectors of a pair of modules (i.e., $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$), and then uses SVM to classify ($\boldsymbol{x}_1$ - $\boldsymbol{x}_2$) into 1 or −1. If the class label is 1, $\boldsymbol{m}_1$ contains more defects than $\boldsymbol{m}_2$; otherwise, $\boldsymbol{m}_2$ contains more defects than $\boldsymbol{m}_1$.

(2) RankBoost [47]: It adopts AdaBoost to classify the modules pairs. The only difference between them is that the distribution is defined on modules pairs in RankBoost, while that is defined on individual modules in AdaBoost [48].

(3) RankNet [49]: The loss function of RankNet is also defined on module pairs, but the hypothesis is defined with the use of a scoring function.

(4) LambdaRank [50]: LambdaRank optimizes an upgraded version of the loss function in RankNet with less computing complexity using the gradient descent method.

2.3. The listwise algorithm

The listwise approach directly optimizes the performance measures to obtain a ranking model [21]. As shown in Fig. 1, assuming that the listwise approach predicts that the ranking list $P_{A,B,C}$ has the best performance among all possible ranking lists (i.e., $P_{A,B,C}$, $P_{A,C,B}$, $P_{B,A,C}$, $P_{B,C,A}$, $P_{C,A,B}$, and $P_{C,B,A}$), so the predicted ranking is A>B>C.

(1) ListNet [51]: Similar to RankNet, it uses a neural network approach with the gradient descent method to minimize a loss function. The loss function is defined by the probability distribution on all possible ranking lists.

(2) Coordinate Ascent [52]: It trains a ranking model by minimizing the Mean Average Precision (MAP) values. It does a number of restarts to guarantee avoidance of the local minimum.

(3) Learning-to-Rank (LTR).[1] It trains the same linear model as LR, i.e.,

$$y = f(\boldsymbol{b}, \boldsymbol{x}) = b_0 + b_1x_1 + b_2x_2 + \cdots + b_mx_m \quad (7)$$

but uses the composite differential evolution algorithm to directly optimize the FPA(Fault Percentile Average) value to find the optimal $\boldsymbol{b}$ value when considering module as effort. When considering LOC as effort, LTR directly optimizes the P_{opt}@LOC value to find the optimal $\boldsymbol{b}$ value, which is actually the LRM-GA algorithm [53]. In addition, some studies [54,55] used the logistic regression model to describe the relationship between software features and the number of defects or defect density:

$$y = \frac{1}{1 + e^{-(b_0+b_1x_1+b_2x_2+\cdots+b_mx_m)}}. \quad (8)$$

Therefore, we also investigate the performance of LTR when using the logistic regression model to build EADP models. We denote LTR with linear model as LTR-linear, and denote LTR with logistic regression model as LTR-logistic.

2.4. Other ranking algorithms

Besides the L2R technology, we also investigate other algorithms for ranking, including ManualDown [56], ManualUp [56] and OneWay [57]. ManualDown and ManualUp are two unsupervised learning algorithms. They are based on the observation of previous studies of Koru et al. [58,59] and Menzies et al. [56]: ManualDown assumes that larger modules tend to have more defects, while ManualUp assumes that smaller modules tend to have higher defect density. Given a module $\boldsymbol{m}$ and one software feature $\boldsymbol{x}_i$ of the module, ManualDown ranks all predicted modules in descending order according to $\boldsymbol{x}_i$, while ManualUp ranks all predicted modules in ascending order according to $\boldsymbol{x}_i$. OneWay [57] is a supervised learning algorithm which leverages the training data to automatically choose the best software feature for ManualUp. When module is used as effort, we employ OneWay to automatically choose the best software feature for ManualDown.

3. Experimental methodology

3.1. Datasets

Many publicly-available module-level module datasets such as NASA [60], SOFTLAB [61] and Relink [62] only contain the information of class label. Since EADP ranks software modules based on defect number or density, we only choose the 49 datasets from the PROMISE [63,64], AEEEM [65], and Eclipse [66] repositories which contain the information of the number of defects. Table 2 shows details of the selected 49 datasets, where #Module represents the number of modules in the dataset, %Defect represents the percentage of defective modules in the dataset, TotalDefects represents the total number of defects in the dataset, AvgDefects represents the average number of defects in the dataset, TotalLOC represents the total number of LOC in the dataset, and AvgLOC represents the average number of LOC in the dataset.

Due to the space limitation, we do not list the software features of the experimental datasets. For a more detailed description of the software features, please refer to [63,65,66] .

[1] The algorithm is named LTR by Yang et al. [5]. Its meaning is different with L2R.

Table 2
Details of the module-level datasets.

Corpora	Dataset	#Module	%Defects	TotalDefects	AvgDefects	TotalLOC	AvgLOC
PROMISE	Ant 1.3	125	16%	33	1.65	37699	301.6
	Ant 1.4	178	22.5%	47	1.18	54195	304.5
	Ant 1.5	293	10.9%	35	1.09	87047	297.1
	Ant 1.6	351	26.2%	184	2.00	113246	322.6
	Ant 1.7	745	22.3%	338	2.04	208653	280.1
	Camel 1.0	339	3.8%	14	1.08	33721	99.5
	Camel 1.2	608	35.5%	522	2.42	66302	109.0
	Camel 1.4	872	16.6%	335	2.31	98080	112.5
	Camel 1.6	965	19.5%	500	2.66	113055	117.2
	Ivy 1.1	111	56.8%	233	3.7	27292	245.9
	Ivy 1.4	241	6.6%	18	1.12	59286	246
	Ivy 2.0	352	11.4%	56	1.4	87769	249.3
	Jedit 3.2	272	33.1%	382	4.24	128883	473.8
	Jedit 4.0	306	24.5%	226	3.01	144803	473.2
	Jedit 4.1	312	25.3%	217	2.75	153087	490.7
	Jedit 4.2	367	13.1%	106	2.21	170683	465.1
	Jedit 4.3	492	2.2%	12	1.09	202363	411.3
	Log4j 1.0	135	25.2%	61	1.79	21549	159.6
	Log4j 1.1	109	33.9%	86	2.32	19938	182.9
	Log4j 1.2	205	92.2%	498	2.63	38191	186.3
	Lucene 2.0	195	46.7%	268	2.95	50596	259.5
	Lucene 2.2	247	58.3%	414	2.88	63571	257.4
	Lucene 2.4	340	59.7%	632	3.11	15508601	302.5
	Poi 1.5	237	59.5%	342	2.43	55428	233.9
	Poi 2.0	314	11.8%	39	1.05	93171	296.7
	Poi 2.5	385	64.4%	496	2.0	119731	311.0
	Poi 3.0	442	63.6%	500	1.78	129327	292.6
	Synapse 1.0	157	10.2%	21	1.31	28806	183.5
	Synapse 1.1	222	27%	99	1.65	42302	190.5
	Synapse 1.2	256	33.6%	145	1.69	53500	209.0
	Velocity 1.4	196	75%	210	1.43	51713	263.8
	Velocity 1.5	214	66.4%	331	2.33	53141	248.3
	Velocity 1.6	229	34.1%	190	2.44	57012	249.0
	Xalan 2.4	723	15.2%	156	1.42	225088	311.3
	Xalan 2.5	803	48.2%	531	1.37	304860	379.7
	Xalan 2.6	885	46.4%	625	1.52	411737	465.2
	Xalan 2.7	909	98.8%	1213	1.35	428555	471.5
	Xerces init	162	47.5%	167	2.17	90718	560.0
	Xerces 1.2	440	16.1%	115	1.62	159254	361.9
	Xerces 1.3	453	15.2%	193	2.8	167095	368.9
	Xerces 1.4	588	74.3%	1596	3.65	141180	240.1
AEEEM	Eclipse JDT Core 3.4	997	20.7%	374	1.82	224055	224.7
	Eclipse PDE UI 3.4.1	1497	14.0%	341	1.63	146952	98.2
	Equinox framework 3.4	324	39.8%	244	1.89	39534	122.0
	Mylyn 3.1	1862	13.2%	340	1.39	156102	83.8
	Apache Lucene 2.4.0	691	9.26%	97	1.52	146952	212.7
Eclipse	Eclipse_II_Package2.0	377	50.4%	917	4.83	796941	2114.0
	Eclipse_II_Package2.1	434	44.7%	662	3.41	987603	2275.6
	Eclipse_II_Package3.0	661	47.3%	1534	4.90	1305908	1975.7

3.2. Preparing training and testing data

We use the following two validation settings, and repeat the training and testing phases 10 times to alleviate possible bias during hyperparameter tuning.

(1) Cross-release Validation: We use the previous release as the training dataset, and use the current release as the testing dataset, i.e., cross-release validation. For example, we use Ant 1.3 as the training dataset and use Ant 1.4 as the testing dataset. Since the AEEEM datasets only contain the single release, we do not employ them under cross-release validation. Therefore, we obtain 32 pairs of training and testing datasets.

(2) Cross-Project Validation: In this setting, EADP models are tested on one project (i.e., the testing dataset), and trained on all other projects (i.e., the training datasets). Given n projects, there are n pairs of training and testing datasets. Since there are 41 releases of 11 projects in the PROMISE datasets and there are duplicated software modules in the different releases of the same project, we only choose the most recent release dataset as training dataset or testing dataset. Finally, we obtain 11 pairs of training and testing datasets for the PROMISE datasets, and 5 pairs of training and testing datasets for the AEEEM datasets.

3.3. Constructing EADP models

We use all software features as independent variables to build EADP models when using module as effort.

(1) For the **classification-based pointwise L2R algorithms**, we first discretize the number of defects into "defective" and "non-defective" class labels, and further use these class labels to train the algorithms to build EADP models. Then, we use the trained EADP models to predict the possibility of new software modules being defective, and regard the possibility as the predicted number of defects of new software modules.

(2) For the **regression-based pointwise L2R algorithms, pairwise L2R algorithms, and listwise L2R algorithms**, we use the actual number of defects as the target variable to build EADP models. Then, we use the trained EADP models to rank new software modules according to the number of defects.

(3) **ManualDown** with LOC software feature in the PROMISE datasets achieves the highest P_{opt}@module value under cross-release

and cross-project settings. ManualDown with NOA (Number of Attributes) software feature in the AEEEM datasets achieves the highest P_{opt}@module value under cross-project setting. ManualDown with Modifier software feature in the Eclipse datasets achieves the highest P_{opt}@module value under cross-release setting. Therefore, we choose the software features as the underlying software features for ManualDown.

We use 19 software features (excluding LOC) of the PROMISE datasets, 18 software features (excluding LOC) of the AEEEM datasets, 206 software features (excluding TotalLOC) of the Eclipse datasets as independent variables to build EADP models when using LOC as effort, since LOC and TotalLOC make up the effort value in the dependent variable of EADP models.

(1) For the **classification-based pointwise L2R algorithms**, we use the class labels to train EADP models. Then, we use the trained models to predict the class label (i.e., Label($\boldsymbol{m}$)) and possibility of a new software module $\boldsymbol{m}$ being defective (i.e., P($\boldsymbol{m}$)). Finally, we have three definitions of the predicted defect density (i.e., D($\boldsymbol{m}$)) of the new module $\boldsymbol{m}$. According to Mende et al. [3], the first definition is as follows:

$$\mathrm{D}(\boldsymbol{m}) = \mathrm{Label}(\boldsymbol{m})/\mathrm{effort}(\boldsymbol{m}), \tag{9}$$

where effort($\boldsymbol{m}$) is LOC of the module $\boldsymbol{m}$.

According to Yang et al. [67], the second definition is as follows:

$$\mathrm{D}(\boldsymbol{m}) = \mathrm{P}(\boldsymbol{m})/\mathrm{effort}(\boldsymbol{m}). \tag{10}$$

According to Ni et al. [13], the third definition is as follows:

$$D(\boldsymbol{m}) = \begin{cases} \mathrm{P}(\boldsymbol{m})/\mathrm{effort}(\boldsymbol{m}), \text{ if } \mathrm{P}(\boldsymbol{m}) \geq 0.5 & \text{(a)} \\ (\mathrm{P}(\boldsymbol{m}) - 1)/\mathrm{effort}(\boldsymbol{m}), \text{ if } \mathrm{P}(\boldsymbol{m}) < 0.5. & \text{(b)} \end{cases} \tag{11}$$

Note that the defect density will be negative according to Eq. (11)(b), but we only use the predicted defect density to rank modules. Eqs. (11)(a) and (11)(b) mean that following Ni et al.'s approach [13], we first inspect all the predicted defective modules according to their defect density, then inspect the predicted non-defective modules according to their defect density.

For the 12 investigated classification-based L2R algorithms, we denote the three definitions as X_1, X_2, X_3, where "X" represents the L2R algorithm, "1" represents the first definition to calculate the defect density, "2" represents the second definition to calculate the defect density, and "3" represents the third definition to calculate the defect density. For example, when employing NB to build EADP models and using the first definition to calculate the defect density, we denote it as NB_1. Ni et al. [13] proposed the EASC method, which uses the naive Bayes algorithm to build EADP models and then uses the third definition to calculate the defect density. That is, EASC is NB_3.

(2) For the **regression-based pointwise L2R algorithms, pairwise L2R algorithms, and listwise L2R algorithms**, we use the actual defect density as the target variable to build EADP models, then use the trained models to rank new software modules according to the defect density. The actual defect density is the ratio between the actual number of defects and LOC or TotalLOC.

(3) **ManualUp** with CAM (Cohesion Among Methods of class) software feature in the PROMISE dataset achieves the highest P_{opt}@LOC value under cross-release and cross-project settings. ManualUp with NOMI (Number Of Methods Inherited) software feature in the AEEEM datasets achieves the highest P_{opt}@LOC value under cross-project setting. ManualUp with NORM_CompilationUnit software feature in the Eclipse datasets achieves the highest P_{opt}@LOC value under cross-release setting. Therefore, we choose the software features as the underlying software features for ManualUp.

We use the grid search to tune hyperparameters of the L2R algorithms to maximize the P_{opt}@effort value, because P_{opt}@effort evaluates the global ranking of EADP models. The candidate parameter values of the classification-based pointwise algorithms and the corresponding regression-based pointwise algorithms are almost the same as the setting of Tantithamthavorn et al.'s work [19], which used the grid search to study the impact of automated parameter optimization on defect prediction models.

3.4. Evaluation measures

We restrict our effort to 20% of total effort in our study. The number 20% has been commonly used as a cutoff value to set the effort required for the defect inspection [68,69]. Suppose we have a dataset with M modules, N bugs and P LOC. Among the M modules, K modules are defective. Given 20% effort, we inspected m modules, which contain n bugs, and p LOC. Among the m modules, k modules are defective. Besides, when we find the first defective module, we have inspected m' modules. We use the following evaluation measures in the experiments, some of which are also widely used in the fields of both software engineering [70–75] and machine learning [76–81].

Precision@20%effort is the proportion of the inspected actual defective modules among all the inspected modules. When the effort is LOC, Precision@20%effort is denoted as Precision@20%LOC; when the effort is module, Precision@20%effort is denoted as Precision@20%module. The denotations of the following evaluation measures are similar to that of Precision@20%effort. A lower Precision@20%effort value indicates that software testers would encounter more false alarms, which may have negative impact on developers' confidence on the EADP model [54].

$$\mathrm{Precision@20\%effort} = k/m \tag{12}$$

Recall@20%effort is the proportion of the inspected actual defective modules among all the actual defective modules in the dataset. A higher Recall@20%effort value indicates that more defective modules can be detected.

$$\mathrm{Recall@20\%effort} = k/K \tag{13}$$

IFA@20%effort is the number of **I**nitial **F**alse **A**larms encountered before we find the first defect. A higher IFA@20%effort value indicates that software testers need to inspect more modules to find the first defect.

$$\mathrm{IFA@20\%effort} = m' \tag{14}$$

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

$$\mathrm{PMI@20\%LOC} = m/M \tag{15}$$

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

$$\mathrm{PLI@20\%module} = p/P \tag{16}$$

PofB@20%effort is the **P**roportion **of** the inspected **B**ugs among all bugs in the dataset. A higher PofB@20%effort value indicates that more bugs could be detected.

$$\mathrm{PofB@20\%effort} = n/N \tag{17}$$

PofB/PMI@20%LOC is the ratio between PofB@20%LOC and PMI@20%LOC. A higher PofB/PMI@20%LOC value indicates that, under the same number of LOC (i.e., 20% LOC) to inspect, software testers can find more bugs by inspecting per 1% module.

$$\mathrm{PofB/PMI@20\%LOC} = \mathrm{PofB@20\%LOC}/\mathrm{PMI@20\%LOC} \tag{18}$$

PofB/PLI@20%module is the ratio between PofB@20%module and PLI@20%module. A higher PofB/PLI@20%module value indicates

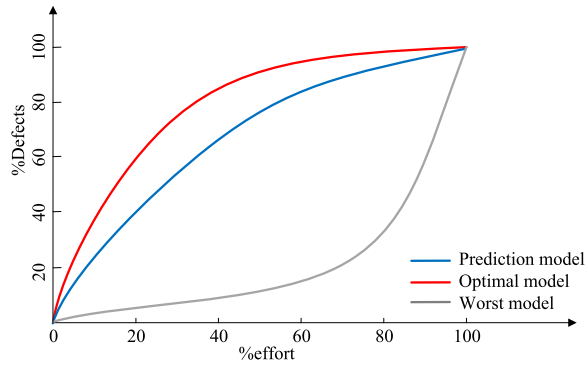


Fig. 2. A Cumulative Lift Chart.

that, under the same number of modules (i.e., 20% modules) to inspect, software testers can find more bugs by inspecting per 1% LOC.

$$\text{PofB/PLI@20\%module} = \frac{\text{PofB@20\%module}}{\text{PLI@20\%module}} \tag{19}$$

***P_{opt}*@effort** is based on the cumulative lift chart shown in Fig. 2, which was proposed by Kamei et al. [4]. In the chart, the x-axis is the cumulative percentage of effort to inspect, and the y-axis is the cumulative percentage of defects found given the effort. There are three curves in the chart, corresponding to the prediction model, the optimal model and the worst model. Modules are ranked by the decreasing predicted number of defects or defect densities according to the prediction model, modules are ranked by the decreasing actual number of defects or defect densities according to the optimal model, and modules are ranked by the increasing actual number of defects or defect densities according to the worst model. P_{opt}@effort is computed as follows:

$$P_{\text{opt}}@\text{effort} = \frac{\text{Area(prediction)} - \text{Area(worst)}}{\text{Area(optimal)} - \text{Area(worst)}} \tag{20}$$

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

When evaluating the performance of EADP models considering module as effort, researchers [5,12] usually employ FPA(Fault Percentile Average) as the evaluation measure. Yang et al. [5] proved that FPA was related to the area under the prediction curve when using module as effort. The difference between FPA and P_{opt}@module is that FPA only reports how well a prediction model is, rather than tells us how close to the optimal model a prediction model is. Therefore, we employ P_{opt}@module as the performance measure to evaluate the performance of EADP models when considering module as effort.

3.5. Statistical comparison test

The Scott-Knott test [82] is a multiple comparison technique that produces statistically distinct ranks at the significance level of 0.05 using hierarchical clustering algorithm. This test ranks and clusters the studied algorithms into significantly different groups, in which the algorithms in distinct groups have significant differences, while the algorithms in the same group have no significant differences [19]. Therefore, the Scott-Knott test can group the studied algorithms distinctly without any overlapping [82]. For more robust result analysis, we use the extended Scott-Knott with Cohen's d effect size awareness (Scott-Knott ESD) [17], which merges any pair of ranks that have a negligible Cohen's d effect size between them to post-processes the statistically distinct ranks produced by the traditional Scott-Knott test. Specifically, we provide the median performance measure values of the 10 cross-release or cross-project iterations of each L2R algorithm to the Scott-Knott ESD test. Then, the test generates statistically distinct ranks of the L2R algorithms.

4. Experimental results

4.1. RQ1: What is the best algorithm for EADP when using module as effort?

Visualizations: Figs. 3 and 4 show the distribution of PofB@20%module, PLI@20%module, and P_{opt}@module values with the Scott-Knott ESD test over all datasets under the cross-release and cross-project settings, respectively. Different colors of the boxplot indicate different Scott-Knott ESD test ranks. From top down, the order is red, green, blue, yellow, purple, orange, pink and gray. Table 3 lists the top-ranked algorithms in terms of P_{opt}@module and PofB@20%module, and the median performance values of the algorithms over all datasets under cross-release and cross-project settings.

Results under the cross-release setting: Fig. 3 and Table 3 reveal that (1) ManualDown and RFR have the highest ranks in terms of P_{opt}@module; (2) LTR-linear, ManualDown, RFR, and RR have the highest ranks in terms of PofB@20%module. Therefore, we will focus on discussing the four algorithms, i.e., ManualDown, RFR, LTR-linear and RR.

(1) ManualDown has the highest median P_{opt}@module value (0.788). When inspecting top 20% modules, ManualDown can identify median 50.2% bugs. But it requires inspecting median 66.5% LOC, which is significantly higher than the amount required by of LTR-linear, RR and RFR.

(2) RFR has the second highest median P_{opt}@module value (0.778). RFR has the highest rank in terms of PofB@20%module, and requires significantly less LOC to be inspected compared with LTR-linear, ManualDown and RR. When inspecting top 20% modules, RFR can identify median 49.6% bugs and requires inspecting median 48.6% LOC. In addition, RFR has the highest PofB/PLI@20%module value (1.033), which indicates that when inspecting top 20% modules, RFR can find median 1.033% bugs by inspecting per 1% LOC.

(3) LTR-linear performs significantly worse than ManualDown and RFR in terms of P_{opt}@module, but has the highest median PofB@20%module value. When inspecting top 20% modules, LTR-linear can identify median 50.4% bugs and requires inspecting median 60.1% LOC.

(4) RR achieves the lowest median P_{opt}@module value and PofB@20%module value among the four algorithms. When inspecting top 20% modules, RR can identify median 48.9% bugs, and requires inspecting median 50.9% LOC.

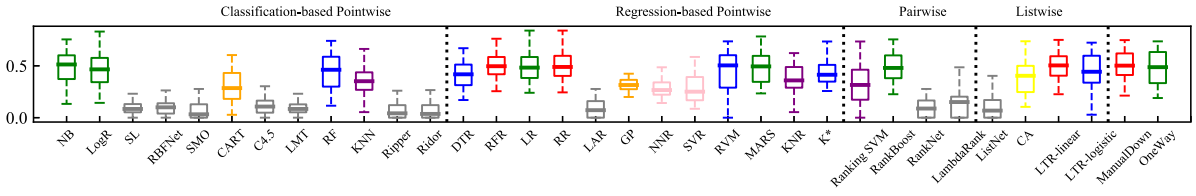
(5) The median IFA@20%module values of the four algorithms are zero, which indicates that following the recommendations given by the four algorithms, the ranked first module is defective in most situations. In terms of Precision@20%module and Recall@20%module, there is not significantly difference among ManualDown, RFR, LTR-linear and RR, since all of them belong to the first rank.

Summary under the cross-release setting: ManualDown and RFR perform the best in terms of Precision@20%module, Recall@20%module, PofB@20%module, and P_{opt}@module, since they have the highest ranks in these evaluation measures. When inspecting top 20% modules, RFR requires significantly less LOC to be inspected and can identify more defects by inspecting per 1% LOC than ManualDown.

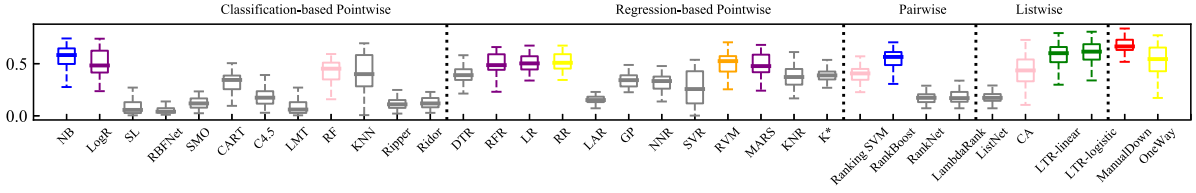
Results under the cross-project setting: Fig. 4 and Table 3 show that (1) ManualDown has the highest rank, OneWay has the second highest rank, and RFR, RankBoost, LTR-linear, and LTR-logistic have the third highest rank in terms of P_{opt}@module; (2) ManualDown has the highest rank, and LR, RR, RankBoost, LTR-linear, and LTR-logistic have the second highest rank in terms of PofB@20%module. Therefore, we will focus on discussing the seven algorithms, i.e., ManualDown, OneWay, RankBoost, LTR-logistic, RR, LR, and LTR-linear.

Table 3
The median performance of the top-ranked algorithms under the **cross-release** and **cross-project** settings when **module** is used as effort. (The algorithms that belong to the first rank are in bold; The algorithms that belong to the second rank are in black; The algorithms that belong to the other ranks are in gray.).

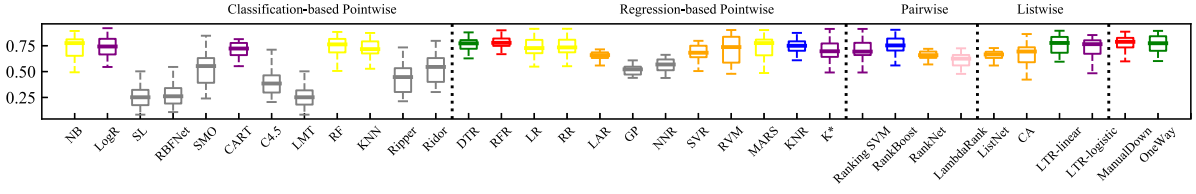
Performance measure	Cross-release setting	Cross-project setting
Precision@20%module	**ManualDown(0.638), RFR(0.619), LTR-linear(0.616), RR(0.605)**	**ManualDown(0.631), OneWay(0.627), LTR-linear(0.614), LTR-logistic(0.594), RR(0.589), LR(0.588), RankBoost(0.579)**
Recall@20%module	**ManualDown(0.366), RFR(0.362), LTR-linear(0.345), RR(0.334)**	LTR-linear(0.326), OneWay(0.314), LTR-logistic(0.31), **ManualDown(0.309)**, RR(0.296), LR(0.294), RankBoost(0.287)
IFA@20%module	ManualDown(0), RFR(0), LTR-linear(0), RR(0)	ManualDown(0), OneWay(0), RankBoost(0), LTR-logistic(0), RR(0), LR(0), LTR-linear(0)
PLI@20%module	**ManualDown(0.665)**, LTR-linear(0.601), RR(0.509), RFR(0.486)	**ManualDown(0.655)**, LTR-logistic(0.647), LTR-linear(0.601), OneWay(0.592), RankBoost(0.561), RR(0.556), LR(0.553)
PofB@20%module	**LTR-linear(0.504), ManualDown(0.502), RFR(0.496), RR(0.489)**	**ManualDown(0.492)**, LTR-linear(0.483), OneWay(0.479), LR(0.474), RR(0.473), LTR-logistic(0.456), RankBoost(0.452)
PofB/PLI@20%module	RFR(1.033), RR(0.933), LTR-linear(0.812), ManualDown(0.784)	LR(0.855), RR(0.848), RankBoost(0.846), OneWay(0.828), ManualDown(0.775), LTR-linear(0.759), LTR-logistic(0.728)
P_{opt}@module	**ManualDown(0.788), RFR(0.778)**, LTR-linear(0.777), RR(0.734)	**ManualDown(0.796)**, OneWay (0.765), RankBoost(0.75), LTR-logistic(0.748), RR(0.74), LR(0.739), LTR-linear(0.735)



(a) The boxplots of the PofB@20%module values.



(b) The boxplots of the PLI@20%module values.



(c) The boxplots of the P_{opt}@module values.

Fig. 3. The boxplots of the performance measure values for EADP under the **cross-release** setting when **module** is used as effort. (Different colors of the boxplot indicate different Scott-Knott ESD test ranks. From top down, the order is red, green, blue, yellow, purple, orange, pink and gray.).

(1) ManualDown has the highest median P_{opt}@module value (0.796), PofB@20%module value (0.492), and Precision@20%module value (0.631). When examining top 20% modules, ManualDown is able to identify median 49.2% bugs. However, it requires inspecting a median of 65.5% LOC, which is significantly higher than the amounts required by the other algorithms.

(2) There is not significant difference among LR, RR, RankBoost, LTR-linear and LTR-logistic, and OneWay in terms of PofB@20%module, as they all have the second highest rank. OneWay has the second highest rank, while LR, RR, RankBoost, LTR-linear and LTR-logistic have the third highest rank in terms of P_{opt}@module. LR requires significantly less LOC to be inspected compared with RR, RankBoost, LTR-linear and LTR-logistic, and OneWay when examining top 20% modules. LR, RR, RankBoost, LTR-linear and LTR-logistic, and OneWay can find median 0.855%, 0.848%, 0.846%, 0.759%, and 0.728% bugs by inspecting per 1% LOC, respectively.

(3) The median IFA@20%module values of the seven algorithms are zero, which indicates that following the recommendations given by the seven algorithms, the ranked first module is defective in most situations. LR, RR, RankBoost, LTR-linear and LTR-logistic, OneWay,

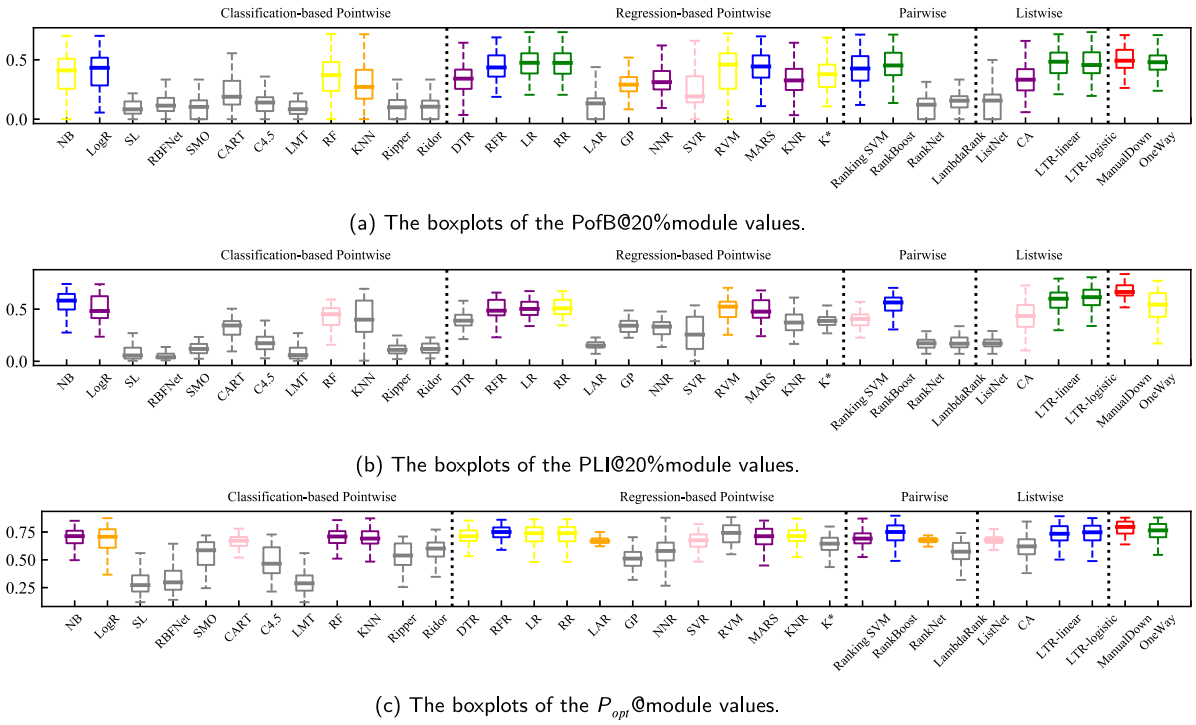


Fig. 4. The boxplots of the performance measure values for EADP under the **cross-project** setting when **module** is used as effort. (Different colors of the boxplot indicate different Scott-Knott ESD test ranks. From top down, the order is red, green, blue, yellow, purple, orange, pink and gray.).

and RFR have the highest rank in terms of Precision@20%module, and belong to the second rank in terms of Recall@20%module.

Summary under the cross-project setting: ManualDown performs the best in terms of Precision@20%module, Recall@20%module, PofB@20%module, and P_{opt}@module, but requires significantly more LOC to be inspected compared to the other algorithms. LR has the third highest rank in terms of P_{opt}@module, only behind ManualDown and OneWay, and has the second highest rank in terms of PofB@20%module, only behind ManualDown. When inspecting top 20% modules, LR is able to inspect more bugs by inspecting per 1% LOC than ManualDown and OneWay.

> **Answer to RQ1:** ManualDown performs the best in terms of PofB@20%module and P_{opt}@module, but requires inspecting much more LOC than other algorithms. RFR performs only behind ManualDown under the cross-release setting, and LR performs only behind ManualDown and OneWay under the cross-project setting in terms of PofB@20%module and P_{opt}@module.

4.2. RQ2: What is the best algorithm for EADP when using LOC as effort?

Visualizations: Figs. 5 and 6 present the boxplots to show the distribution of PofB@20%LOC, PMI@20%LOC, and P_{opt}@LOC values of the algorithms with the Scott-Knott ESD test results across all studied datasets under the cross-release and cross-project, respectively. In order to show the results clearly, we only list the algorithms with the best-performing defect density definition in terms of P_{opt}@LOC and PofB@20%LOC for NB, SL, RBFNet, SMO, CART, KNN, Ripper, and Ridor, since none of three defect density definitions of these algorithms gain high performance. Tables 4 lists the top-ranked algorithms in terms of P_{opt}@LOC and PofB@20%LOC, and the median performance values of the algorithms over all datasets under the cross-release and cross-project settings, respectively.

Results under the cross-release setting: From Fig. 5 and Table 4, we observe that LogR_1, LTR-linear, ManualUp, and OneWay belong to the first rank in terms of P_{opt}@%LOC, and LogR_1, SL_1, RBFNet_1, ManualUp, and OneWay belong to the first rank in terms of PofB@20%LOC. Therefore, we mainly discuss the six algorithms, i.e., LogR_1, SL_1, RBFNet_1, LTR-linear, ManualUp, and OneWay.

(1) LTR-linear achieves the highest median P_{opt}@%LOC value (0.695) and Precision@20%LOC value (0.394), and belongs to the second rank in terms of PofB@20%LOC. LTR-linear needs to inspect significantly less modules, and performs significantly better than LogR_1, SL_1, RBFNet_1, ManualUp, and OneWay in terms of PofB/PMI @20%LOC. When inspecting top 20% LOC, LTR-linear can find median 1.034% bugs by inspecting per 1% module. In addition, LTR-linear performs significantly better than LogR_1, SL_1, RBFNet_1, ManualUp, and OneWay in terms of IFA@20%LOC. The median IFA@20%LOC value of LTR-linear is 3.

(2) LogR_1, ManualUp, and OneWay belong to the first rank in terms of P_{opt}@LOC and PofB@20%LOC. SL_1 and RBFNet_1 belong to the first rank in terms of PofB@20%LOC, and belong to the second rank in terms of P_{opt}@LOC. In addition, LogR_1, SL_1, RBFNet_1, ManualUp, and OneWay perform the best in terms of Recall@20%LOC, since they need to inspect significantly more modules than LTR-linear. In addition, the Precision@20%LOC values of the algorithms are very low, and the IFA@20%LOC values are very high. Kochhar et al. [83] conducted a survey about the expectations of software testers on automated fault localization, and found that most practitioners thought it unacceptable if the first 10 suspicious program elements returned by a tool are all false alarms.

Summary under the cross-release setting: LogR_1, ManualUp, and OneWay perform the best in terms of PofB@20%LOC and P_{opt}@LOC. However, they need to inspect significantly more modules and produce many false alarms. LTR-linear achieves the highest median P_{opt}@%LOC value, and significantly performs better than LogR_1, ManualUp, and

OneWay in terms of Precision@20%LOC, IFA@20%LOC and PofB/PMI@20%LOC.

Results under the cross-project setting: From Fig. 6 and Table 4, we observe that LogR_1, SL_2, RBFNet_1, SMO_2, LMT_2, Ripper_1, Ridor_2, and ManualUp belong to the first rank, C4.5_2, RF_1, Ranking SVM, LTR-linear, LTR-logistic, and OneWay belong to the second rank in terms of P_{opt}@%LOC; Ripper_1, ManualUp, and OneWay belong to the first rank, LogR_1, SL_2, RBFNet_1, SMO_2, C4.5_2, LMT_2, Ridor_2, and LTR-logistic belong to the second rank in terms of PofB@20%LOC. Therefore, we mainly discuss the 14 algorithms, i.e., LogR_1, SL_2, RBFNet_1, SMO_2, LMT_2, Ripper_1, Ridor_2, C4.5_2, RF_1, Ranking SVM, LTR-linear, LTR-logistic, ManualUp, and OneWay.

(1) Ripper_1 and ManualUp belong to the first rank in terms of P_{opt}@%LOC and PofB@20%LOC. LogR_1, SL_2, RBFNet_1, SMO_2, LMT_2, Ridor_2, C4.5_2, RF_1, LTR-logistic and OneWay belong to the first or second rank in terms of P_{opt}@%LOC and PofB@20%LOC. However, the median IFA@20%LOC of the algorithms are very high. Following the recommendations given by the algorithms, software testers need to inspect more than ten modules to find the first defective modules in most situations. In addition, the Precision@20%LOC values of the algorithms are very low, which also indicates that software testers would encounter many false alarms when recommended by the algorithms.

(2) Ranking SVM belongs to the second rank in terms of P_{opt}@%LOC. When inspecting top 20% LOC, Ranking SVM can find median 28.5% bugs and needs to inspect median 16.1% modules. In other words, Ranking SVM can find median 1.555% bugs by inspecting per 1% module, which is significantly higher than other algorithms. The median IFA@20%LOC value of Ranking SVM is 3, which indicates that software testers need to inspect median three modules to find the first defective module in most situations. In addition, Ranking SVM belongs to the first rank in terms of Precision@20%LOC, which indicates that the false alarm rate of Ranking SVM is significantly lower than other algorithms.

(3) LTR-linear belongs to the second rank in terms of P_{opt}@%LOC, and belong to the fourth rank in terms of PofB@20%LOC. The median P_{opt}@%LOC and PofB@20%LOC values are lower than those of Ranking SVM.

Summary under cross-project setting: Ripper_1 and ManualUp perform the best in terms of PofB@20%LOC and P_{opt}@LOC. However, they need to inspect significantly more LOC and produce very high false alarms. Ranking SVM belongs to the second rank in terms of P_{opt}@%LOC, and significantly performs better than Ripper_1 and ManualUp in terms of Precision@20%LOC, IFA@20%LOC and PofB/PMI@20%LOC.

> **Answer to RQ2:** Among those algorithms with low IFA values, LTR-linear performs the best under the cross-release setting, and Ranking SVM performs the best in terms of Precision@20%LOC, PofB@20%LOC and P_{opt}@%LOC.

5. Discussion

5.1. Performance interpretation

When using module as effort, i.e., ranking modules according to the predicted number of defects, ManualDown and OneWay rank the larger modules first. Since larger modules tend to have more defects [58], ManualDown and OneWay can find more defective modules and defects under the same inspection cost, and most of the 20% modules are defective. Therefore, the Recall@20%module, PofB@20%module, and Precision@20%module values of ManualDown and OneWay are high. The modules with more defects are ranked first by a EADP model, the higher P_{opt}@module value of the model is. Therefore, the P_{opt}@module value is also high. But the larger modules contain more LOC, ManualDown and OneWay require software testers to inspect more LOC under the same inspection cost (i.e., 20% module). Therefore, the PLI@20%module values of ManualDown and OneWay are high. As a result, ManualDown and OneWay belong to the first or second rank in terms of Precision@20%module, Recall@20%module, PofB@20%module, P_{opt}@module, and PLI@20%module in most situations.

When using LOC as effort, i.e., ranking modules according to the predicted defect density, ManualUp and OneWay rank the smaller modules first. Most modules are small while a few are large. ManualUp and OneWay require software testers to inspect more modules (i.e., higher PMI@20%LOC) under the same inspection cost. Therefore, the Recall@20%LOC and PofB@20%LOC values of ManualUp and OneWay are high. The modules with higher defect density are ranked first by the model, the higher P_{opt}@LOC value of the model is. Therefore, the P_{opt}@LOC value is also high. But the smaller modules tend to be non-defective, the Precision@20%LOC values of ManualUp and OneWay are low and the IFA@20%LOC values are high. Therefore, ManualUp and OneWay belong to the first or second rank in terms of Recall@20%LOC, PofB@20%LOC, P_{opt}@LOC, PMI@20%LOC, PCI@20%LOC, and IFA@20%LOC in most situations.

The performance of the three categories of L2R algorithms can be due to the difference of utility functions during training.

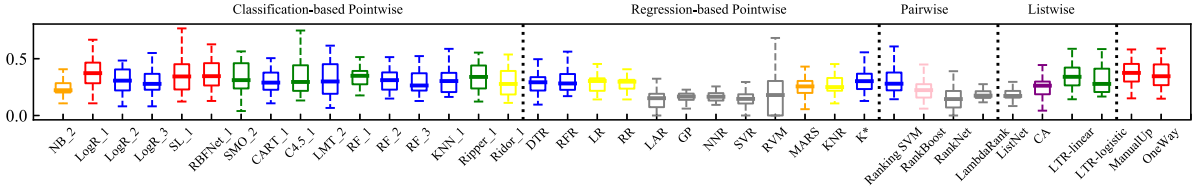
(1) As mentioned in Section 2, the listwise approach directly optimizes the performance measure to obtain a ranking function. We find that LTR-linear and LTR-logistic perform well in terms of P_{opt}@effort, because they directly optimize the P_{opt}@effort value. However, other listwise algorithms have poor performance. The main reason is that the goal of these algorithms is to optimize some information retrieval performance measures, such as normalized discounted cumulative gain and mean average precision, which do not take the effort into consideration [10]. Since the high P_{opt}@effort value indicates that the modules with more defects or higher defect density are ranked first, the Recall@20%effort and PofB@20%effort values of LTR-linear and LTR-logistic are also high. In addition, the performance of LTR-linear is better than that of LTR-logistic in most cases, which indicates that there is a stronger linear relationship than no-linear relationship between software features and the number of defects or defect density.

(2) When considering module as effort, the classification-based pointwise algorithms have poor performance in most situations. The main reason is that the algorithms only use the class label instead of the number of defects to build the EADP models. When considering LOC as effort, the PofB@20%LOC and P_{opt}@LOC values of the classification-based pointwise algorithms with the first and second definitions of defect density (i.e., Eq. (9) and Eq. (10)) are high. The main reason is that the predicted defect density is the ratio between the predicted class label or possibility of the module being defective and the LOC of the module, which makes that the modules with less LOC are more likely ranked first. Therefore, similar to ManualUp and OneWay, the classification-based pointwise algorithms have high Recall@20%LOC, PofB@20%LOC, P_{opt}@LOC, and PMI@20%LOC values in most situations. Since the classification algorithms are imperfect, for those modules that are predicted as defective, some of them may be false positives (i.e., they are actually non-defective modules), especially under the cross-project setting because there is different data distribution between cross-project data and within-project data. Therefore, the top ranked modules are likely to be non-defective according to the first and second definitions of defect density, i.e., the IFA@20%LOC values of the classification algorithms are high.

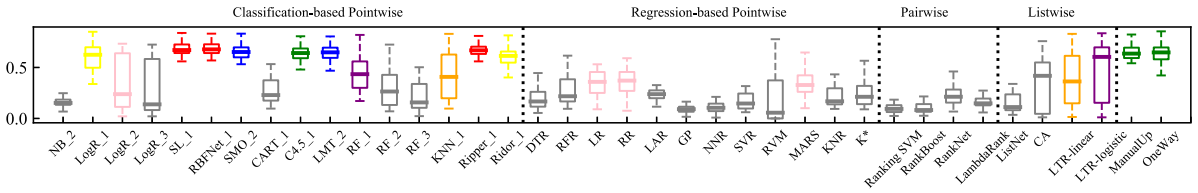
(3) Some regression-based pointwise algorithms perform well in terms of PofB@20%module and P_{opt}@module, such as RFR, RR, and LR. These algorithms outperform other regression algorithms for several reasons. RFR is an ensemble learning algorithm, which grows an ensemble of regression trees and allows them to vote on the decision to improve the performance. Since we find that there is a stronger linear

Table 4
The median performance of the top-ranked algorithms for EADP under the **cross-release** and **cross-project** settings when LOC is used as effort. (The algorithms that belong to the first rank are in bold; The algorithms that belong to the second rank are in black; The algorithms that belong to the other ranks are in gray.).

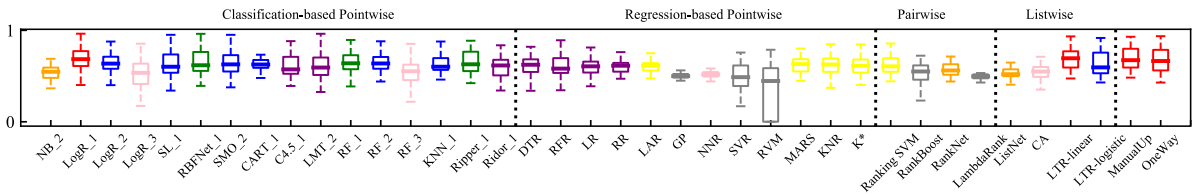
Performance measure	Cross-release setting	Cross-project setting
Precision@20%LOC	LTR-linear(0.394), LogR_1(0.201), ManualUp(0.201), OneWay(0.19), RBFNet_1(0.18), SL_1 (0.176)	**Ranking SVM(0.53),** LTR-linear(0.241), OneWay(0.219), RF_1(0.218), LTR-logistic(0.214), ManualUp(0.208), LMT_2(0.201), C4.5_2(0.2), Ridor_2(0.199), SL_2(0.198), RBFNet_1(0.197), LogR_1(0.197), Ripper_1(0.197), SMO_2(0.196)
Recall@20%LOC	**RBFNet_1(0.494), LogR_1(0.493), ManualUp(0.487), SL_1(0.485), OneWay(0.467),** LTR-linear(0.324)	**Ripper_1(0.532), ManualUp(0.53), SMO_2(0.529), SL_2(0.527), SMO_2(0.527), RBFNet_1(0.524), OneWay(0.521), Ridor_2(0.505),** LogR_1(0.461), C4.5_2(0.46), LTR-logistic(0.455), RF_1(0.365), LTR-linear(0.268), Ranking SVM(0.232)
IFA@20%LOC	**RBFNet_1(18), SL_1(17), LogR_1(15),** OneWay(11), ManualUp(9), LTR-linear(3)	RBFNet_1(24), SL_2(19), SMO_2(19), LMT_2(19), Ripper_1(19), Ridor_2(19), C4.5_2(19), LogR_1(18), RF_1(16), **ManualUp(14), OneWay(14),** LTR-logistic(11), LTR-linear(4), Ranking SVM(3)
PMI@20%LOC	**RBFNet_1(0.677), SL_1(0.671),** OneWay(0.648), ManualUp(0.636), LogR_1(0.625), LTR-linear(0.363)	**SMO_2(0.7), Ripper_1(0.699), SL_2 (0.697), LMT_2(0.691), RBFNet_1(0.689), Ridor_2(0.686),** ManualUp(0.666), OneWay(0.661), C4.5_2(0.66), LogR_1(0.649), RF_1 (0.624), LTR-logistic(0.613), LTR-linear(0.497), Ranking SVM (0.161)
PofB@20%LOC	**ManualUp(0.375), LogR_1(0.373), RBFNet_1(0.346), OneWay(0.345), SL_1(0.343),** LTR-linear(0.34)	**ManualUp(0.42), Ripper_1(0.416), OneWay(0.414),** RBFNet_1(0.388), LogR_1(0.387), SMO_2(0.386), SL_2(0.38), Ridor_2(0.38), LMT_2(0.38), C4.5_2(0.359), LTR-logistic (0.345), RF_1 (0.331), Ranking SVM (0.285), LTR-linear (0.283)
PofB/PMI@20%LOC	LTR-linear(1.034), LogR_1(0.615), ManualUp(0.601), OneWay (0.561), SL_1(0.542), RBFNet_1(0.536)	**Ranking SVM(1.555),** LTR-linear(0.672), RF_1(0.615), ManualUp(0.601), LTR-logistic(0.587), OneWay(0.587), LogR_1(0.583), SMO_2(0.568), Ripper_1(0.564), Ridor_2(0.563), C4.5_2(0.561), SL_2(0.556), RBFNet_1(0.556), LMT_2(0.552)
P_{opt}@LOC	**LTR-linear(0.695), LogR_1(0.685), ManualUp(0.674), OneWay(0.664),** RBFNet_1(0.618), SL_1(0.602)	**ManualUp(0.711), LogR_1(0.677), Ripper_1(0.672), SMO_2(0.669), SL_2(0.662), Ridor_2(0.66), LMT_2(0.655), RBFNet_1(0.653),** OneWay (0.652), LTR-logistic(0.649), C4.5_2(0.649), Ranking SVM(0.616), LTR-linear(0.609), RF_1(0.603)



(b) The boxplots of the PofB@20%LOC values.

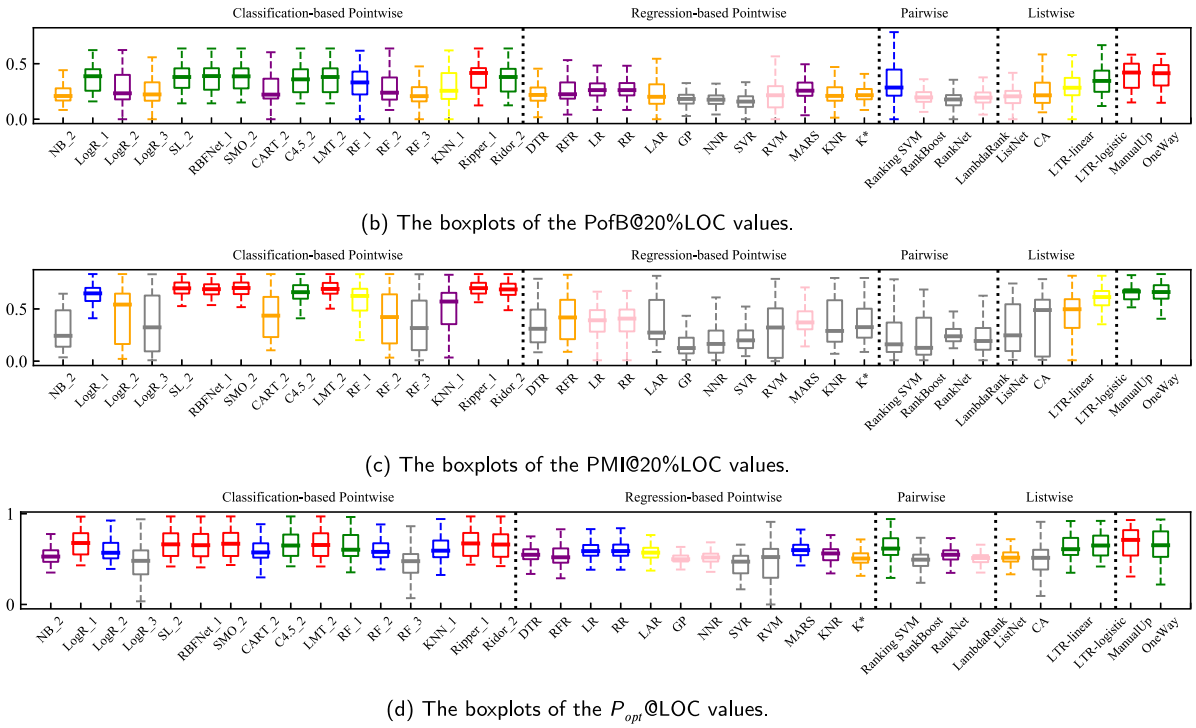


(c) The boxplots of the PMI@20%LOC values.



(d) The boxplots of the P_{opt}@LOC values.

Fig. 5. The boxplots of the performance measure values for EADP under the **cross-release** setting when **LOC** is used as effort. (Different colors of the boxplot indicate different Scott-Knott ESD test ranks. From top down, the order is red, green, blue, yellow, purple, orange, pink and gray.).



(b) The boxplots of the PofB@20%LOC values.

(c) The boxplots of the PMI@20%LOC values.

(d) The boxplots of the P_{opt}@LOC values.

Fig. 6. The boxplots of the performance measure values for EADP under the **cross-project** setting when **LOC** is used as effort. (Different colors of the boxplot indicate different Scott-Knott ESD test ranks. From top down, the order is red, green, blue, yellow, purple, orange, pink and gray.).

relationship than no-linear relationship between software features and the number of defects, RR and LR perform better than other non-linear regression algorithms, except RFR. As for the other two linear regression algorithms, GP has poor performances, and LAR has poor performances under the cross-release and cross-project settings. The reasons can be explained as follows. GP and LR minimize the same loss function (i.e., Eq. (3)), but LR uses the least square method, while GP uses the genetic algorithm. We find that it is more difficult for GP to obtain the small value of Eq. (6), thus GP has poor performances. There is a different data distribution between the training dataset and the testing dataset under the cross-release and cross-project settings, and LAR is known to be extremely sensitive to noise [38]. Consequently, LAR has poor performances under the cross-release and cross-project settings. When considering LOC as effort, the regression-based pointwise algorithms belong to the latter ranks in terms of Recall@20%LOC, PofB@20%LOC and P_{opt}@LOC in most situations. The main reason is that the classification-based pointwise algorithms have higher Recall@20%LOC, PofB@20%LOC, and P_{opt}@LOC values than the regression algorithms.

(4) In most cases, the pairwise algorithms have poor performance. The reasons can be explained as follows. The goal of the pairwise algorithms is to minimize the number of incorrect rankings. Here, an incorrect ranking means that a module with less defects or lower defect density is ranked ahead of a module with more defects or higher defect density in a given module pair. When a model ranks the modules with more defects or higher defect density correctly, the model will obtain higher P_{opt}@effort value. Therefore, EADP models should rank the modules with more defects or higher defect density correctly, and a higher cost should be assigned to the incorrect ranking of a module with more defects or higher defect density than a module with less defects or lower defect density. However, the pairwise algorithms allocate the same costs for incorrect ranked modules with more defects or higher defect density and incorrect ranked modules with one defect or lower defect density. That is, the training utility function of these pairwise algorithms also does not take the effort into consideration. But RankBoost perform well in terms of PofB@20%module and P_{opt}@module, and Ranking SVM performs well in terms of PofB@20%LOC and P_{opt}@LOC under the cross-project setting. The reason is that there is different data distribution between cross-project modules and within-project modules. Ranking SVM and RankBoost compute the difference of software features of two modules to generate the module pairs, then train EADP models on the module pairs. Since the distribution difference of cross-project module pairs and within-project module pairs is smaller than that of cross-project modules and within-project modules, the performances of Ranking SVM and RankBoost is higher than that of other algorithms under the cross-project setting.

5.2. Implications

In this subsection, we analyze implications based on our experimental results and provide a few suggestions to practitioners and researchers in this domain.

(1) When module is considered as effort, if software testers are insensitive to the number of inspected LOC, ManualDown is recommended due to the several advantages: (a) ManualDown enables testers to find most bugs when inspecting top 20% modules; (b) It does not require training data and L2R algorithms. Therefore, it can be easily applied in a new project, and it is faster to execute. (c) The intuition behind the algorithm is straightforward: larger modules tend to contain more bugs. Otherwise, RFR is recommended to build the cross-release EADP model, and LR is recommended to build the cross-project EADP model, when software testers would like to obtain more accurate global ranking of the predicted modules according to the number of defects (i.e., higher P_{opt}@module value) and find more bugs when inspecting top 20% modules (i.e., higher PofB@20%module value).

(2) When LOC is considered as effort, ManualUp, OneWay and some classification algorithms using the first and second definitions of defect density (e.g., LogR_1, RF_1, SMO_2) should be avoided for EADP in practice, although the algorithms achieve the highest PofB@20%LOC, Recall@20%LOC and P_{opt}@LOC values. The reason is that many highly ranked modules given by the algorithms are false alarms under the

cross-release and cross-project settings, which negatively impacts practitioners' patience and confidence. LTR-linear is recommended to build the cross-release EADP model, and Ranking SVM is recommended to build the cross-project EADP model, when software testers would like to encounter lower false alarms, obtain more accurate global ranking of the predicted modules according to the defect density (i.e., higher P_{opt}@LOC value), and find more bugs when inspecting top 20% LOC (i.e., higher PofB@20%LOC value).

(3) Some previous EADP empirical studies should be revisited. Yang et al. [67] and Ma et al. [84] proposed that slice-based cohesion features and network measures were useful when using logistic regression to build EADP models considering LOC as effort. Qu et al. [85] proposed that using *k*-core decomposition on class dependency networks to analyze software bug distribution can improve the performance when using logistic regression and random forest to build EADP models considering LOC as effort. However, our experimental results show that logistic regression and random forest with the first or second definitions of defect density need to inspect significantly more modules than other algorithms. In addition, many highly ranked modules produced by the algorithms are false alarms under the crossni'fa-release and cross-project settings, which make software testers unwilling to use the algorithms to build the prediction models. Therefore, these prior studies should be revisited to determine whether their findings are impacted when using better L2R algorithms to build EADP models.

5.3. Threats to validity

We use 49 module-level datasets from three repositories (i.e., PROMISE, AEEEM, and Eclipse), each of which provides different software features and different granularity of modules (e.g., file, package). Since the software features and granularity of modules vary, this is a point of variation between the studied datasets that may impact our results. In order to investigate the effect of different features and granularity, we separately analyze the experimental results on datasets that have the same set of software features, and find that our findings largely remain the same. Therefore, we conclude that the variation of software features and granularity of modules does not pose a threat to our study. On the other hand, the variety of software features and granularity of modules also strengthen the generalization of our results, i.e., our findings are not bound to one specific set of software features.

We investigate the performance of all L2R algorithms studied in the 24 previous EADP works, except PR, NBR, ZINBR, ZIPR, HNBR, HPR, the proposed segmented model [86], ROCPDP, (Bagging/AdaBoost/Rotation Forest/Random Subspace)+(LMT/NB/SL/SMO/C4.5), PCR, CamargoCruz09-DT, Turhan09-DT, Menzies11-RF, Watanabe08-DT, cost sensitive Ranking SVM, top-dev. The reasons are as follows.

(1) We used R packages to implement PR, NBR, ZINBR, ZIPR, HNBR, and HPR. Similar to Yang et al. [5], we also found these algorithms cannot produce a result for some datasets.

(2) The segmented model [86] used the software features that are computed on the sub-dependence graph of functions. Since our used datasets do not contain the feature information, we do not investigate the performance of the segmented model.

(3) You et al. [20] proposed the ROCPDP algorithm, which is actually the Ridge Regression (RR) algorithm.

(4) Yan et al. [9] and Yang et al. [69] investigated the performances of some ensemble learning methods (i.e., Bagging, AdaBoost, Rotation Forest and Random Subspace). Because the four ensemble learning methods can combine 11 base classification-based L2R algorithms (except RF) and 11 regression-based L2R algorithms (except RFR) investigated in our study, there are 88(=4 × 22) combinations. It is impractical to investigate all the combinations. Therefore, our study focuses on investigating which base algorithm has the best performance for EADP.

(5) The difference between PCR and LR is that PCR builds a linear regression model based on the principal components, and LR builds the model based on the original software features. Since our study focuses on investigating which base algorithm has the best performance for EADP, we do not consider the impact of software features. In addition, the works on feature selection for EADP have been limited. Only Yang et al. [5] applied information gain to investigate the effectiveness of different features for EADP, and Yang et al. [12] investigated PCR for EADP. These studies found that information gain and PCA did not significantly improve the performance of EADP models. Therefore, we do not investigate the performance of PCR for EADP.

(6) Yu et al. [87] proposed a cost sensitive Ranking SVM algorithm to rank modules according to the number of defects. The algorithm employs the cost sensitive learning approach to learns from imbalanced defect data for module-level EADP. Since the algorithms studied in our work do not consider the data imbalance problem, we do not investigate the cost sensitive Ranking SVM algorithm to ensure a fair comparison.

(7) Ni et al. [13] compared EASC and the four cross-project defect prediction algorithms (i.e., CamargoCruz09-DT, Turhan09-DT, Menzies11-RF, and Watanabe08-DT), and found that EASC outperformed them for cross-project EADP. In addition, the algorithms studied in our work do not consider the different data distribution between within-project and cross-project datasets. Therefore, we do not investigate the algorithms to ensure a fair comparison.

(8) Qu et al. [88] proposed an unsupervised algorithm and a supervised equation both named top-dev, which need the information of the number of developer working on a software modules. Since our experimental datasets do not contain the information, we do not investigate top-dev.

6. Related work

6.1. Literature review

To understand the progress in EADP studies, we conducted a search of the related articles published between 2010 and December 2022.[2] To the best of our knowledge, the first two EADP articles were published by Mende et al. [3] and Kamei et al. [4] in 2010, and thus we set the starting year of the search to 2010. Then, we followed Zhou et al.'s approach [89] to conduct a forward snowballing search. Specifically, we first searched the articles having cited the Mende et al.'s [3] and Kamei et al.'s [4] articles through Google Scholar, then filtered out the unrelated articles. Next, we repeated this process on all the reserved articles. Finally, the search process found 24 **module-level** EADP articles. Table 5 provides an overview of the studies, and the last column lists the ranking criterions. "Bugs(***m***)" represents ranking modules based on the predicted number of defects, and "Bugs(***m***)/LOC(***m***)" represents ranking modules based on the predicted defect density. Since classification algorithms can only predict the probability of the module being defective or the class label, we regard "P(***m***)" as ranking modules based on the number of defects, and regard "Label(***m***)/LOC(***m***)", "P(***m***)/LOC(***m***)", "P(***m***)×(1-LOC(***m***)/LOC_{max})", "[P(***m***)×coreness(***m***)]/LOC(***m***)", and [P(***m***) × $coreness_g$ (***m***)]/LOC(***m***) as ranking software modules based on the defect density.

The studies can be broadly classified into two categories: those that propose novel algorithms and those that empirically evaluate the performance of different algorithms to determine the best ones. In these empirical studies, Nguyen et al. [6], Bennin et al. [7,8], Mthukumaran et al. [92], Yan et al. [9], Wang et al. [10], Miletić et al. [11], Yang et al. [12] and Ni et al. [13] investigated the performance of different L2R algorithms for module-level EADP. Nguyen et al. [6]

[2] The search was conducted in December 2022.

Table 5
Literature overview of module-level EADP.

Study	Datasets used	L2R algorithms	Evaluation measures	Ranking criterion[a]
Jiang et al. [90] 2008	8 NASA datasets	NB (Naïve Bayes), LogR (Logistic Regression), KNN (K-Nearest Neighbors), C4.5, Bagging	CLC (Cumulative Lift Chart)	P(***m***)
Mende et al. [91] 2009	13 NASA datasets	NB, LogR, CART (Classification and Regression Tree), Bagging, RF (Random Forest)	cost effective, P_{opt}	P(***m***)/LOC(***m***)
Mende et al. [3] 2010	12 NASA datasets, 3 Eclipse datasets	RF	cost effective	P(***m***) × (1-LOC(***m***)/LOC_{max}), Label (***m***)/LOC(***m***)
Kamei et al. [4] 2010	3 Eclipse datasets	LR (Linear Regression), CART, RF	P_{opt}	Label(***m***)/LOC(***m***)
Yang et al. [67] 2014	6 open source datasets	LogR	cost effective, effort reduction	P(***m***)/LOC(***m***)
Yang et al. [5] 2014	6 Eclipse data sets, 5 AEEEM datasets	LTR (Learning-to-Rank), RFR (Random Forest Regression), NBR (Negative Binomial Regression), DTR (Decision Tree Regression), ZINBR (Zero-Inflated Negative Binomial Regression), ZIPR (Zero-Inflated Poisson Regression), HNBR (Hurdle Negative Binomial Regression), HPR (Hurdle Poisson Regression)	FPA(Fault-Percentile-Average)	Bugs(***m***)
Nguyen et al. [6] 2014	5 Eclipse datasets	KNR (K-Nearest neighbor Regression), LR, MARS (Multivariate Adaptive Regression Splines), Ranking SVM, RankBoost	SRCC (Spearman Rank Correlation Coefficient)	Bugs(***m***)
Mthukumaran et al. [92] 2016	3 open source datasets	LR	cost effective, CLC	Bugs(***m***), Bugs(***m***)/LOC(***m***)
Panichella et al. [53] 2016	20 PROMISE datasets	LR, CART, LRM-GA (Linear Regression Model with Genetic Algorithm)	P_{opt}	Bugs(***m***)/LOC(***m***)
Ma et al. [84] 2016	11 PROMISE datasets	LogR	cost effective, effort reduction	P(***m***)/LOC(***m***)
Yang et al. [86] 2016	5 open source datasets	a segmented model	cost effective, effort reduction	Label(***m***)/LOC(***m***)
Bennin et al. [7] 2016	25 open source datasets	LR, LAR (Least Angle Regression), RVM (Relevance Vector Machine), KNR, K*, NNR (Neural Network Regression), SVR (Support Vector Regression), DTR (Decision Tree Regression), RFR	P_{opt}	Label(***m***)/LOC(***m***)
Bennin et al. [8] 2016	10 open source datasets	LR, LAR, RVM, KNR, K*, NNR, SVR, DTR, RFR	P_{opt}	Label(***m***)/LOC(***m***)
You et al. [20] 2016	5 AEEEEM datasets, 34 PROMISE datasets	ROCPDP (Ranking-Oriented Cross-Project Defect Prediction), LR, RFR, GBR, DTR, SVR, Ranking SVM, RankBoost, GP (Genetic Programming)	Spearman, Kendall, Accuracy	Bugs(***m***)
Mthukumaran et al. [92] 2016	3 open source datasets	LR	cost effective, CLC	Bugs(***m***), Bugs(***m***)/LOC(***m***)
Yan et al. [9] 2017	10 PROMISE datasets	LR, SL, LogR, RBFNet, SMO, KNN, Ripper, Ridor, NB, C4.5, LMT, RF, (Bagging/AdaBoost/Rotation Forest/Random Subspace)+(LMT/NB/SL/SMO/C4.5)[b], ManualUp	cost effective, P_{opt}	Label (***m***)/LOC(***m***)
Wang et al. [10] 2018	34 PROMISE datasets	RFR, RankNet, LambdaRank, ListNet, Coordinate Ascent	NDCG	Bugs(***m***)
Miletić et al. [11] 2018	2 Eclipse datasets	RF, LogR, NB, C4.5, ManualUp	AUC (Area Under Curve), GM (Geometric Mean), P_{opt}	Label(***m***)/LOC(***m***)
Yang et al. [12] 2018	41 PROMISE datasets	LAR, RR (Ridge Regression), NBR, PCR (Principal Component Regression), RFR, LTR	CLC, FPA	Bugs(***m***)
Qu et al. [85] 2019	18 Java Software datasets	RF, LogR	P_{opt}	[P(***m***) × coreness(***m***)]/ LOC(***m***)
Ni et al. [13] 2020	12 NASA datasets, 5 AEEEM datasets, 3 RELINK datasets, and 65 PROMISE datasets	CamargoCruz09-DT, Turhan09-DT, Menzies11-RF, Watanabe08-DT, ManualUp, ManualDown, EASC (Effort-Aware Supervised Cross-project defect prediction)	F1-score, PF (Probability of False alarm), AUC, IFA, PMI@20%LOC (Proportion of Modules Inspected), CostEffort, P_{opt}	P(***m***) /LOC(***m***)
Yu et al. [87] 2020	41 PROMISE datasets	Cost-sensitive Ranking SVM, DTR, LR, RR, Ranking SVM, LTR	Recall@20%module, PofB@20%module, FPA	Bugs(***m***)
Qu et al. [88] 2021	9 open source Java datasets	top-dev, ManualUp, CBS+	PMI@20%LOC, P_{opt}	Label(***m***)/LOC(***m***), P(***m***)/LOC(***m***), P(***m***)×(1-LOC(***m***)/LOC_{max})
Rao et al. [93] 2021	41 PROMISE datasets	LTR, CBS+, EASC, LR	PofB@20%LOC, PMI@20%LOC, Recall@20%LOC, Precision@20%LOC, IFA, P_{opt}	Bugs(***m***) /LOC(***m***)
Du et al. [94] 2022	18 open source Java datasets	LogR	P_{opt}	[P(***m***) × $coreness_g$ (***m***)]/LOC(***m***), [P(***m***) × coreness(***m***)]/LOC(***m***), P(***m***) /LOC(***m***)

[a] Bugs(***m***) is the predicted number of defects of the module ***m***, P(***m***) is the probability of the module being defective, Label(***m***) is the class label of the module, LOC(***m***) is the line of codes of the module, LOC_{max} is the maximum value of the lines of codes of all predicted modules, and coreness(***m***) is the coreness of the module in the class dependency network [85].

[b] (Bagging/AdaBoost/Rotation Forest/Random Subspace)+(LMT/NB/SL/SMO/C4.5) represents that Yan et al. [9] investigated the performance of four ensemble learning methods (i.e., Bagging, AdaBoost, Rotation Forest and Random Subspace) using LMT, NB, SL, SMO and C4.5 as the base learners.

discovered that RankBoost had more stable prediction performance than LR, MARS, KNN, Ranking SVM. Bennin et al. [7,8] found that DTR performed the best in terms of P_{opt} when using the cross-release setup. Wang et al. [10] found that RankNet performed the best in terms of NDCG in the scenario of cross-project EADP. Miletić et al. [11] found that LogR gained the best results in the scenario of cross-release EADP. Yang et al. [12] found that RR can achieve better results than LR and NBR, slightly better results than LAR, PCR and LTR [5], and slightly worse results than RFR under cross-release setting. Yan et al. [9] found that ManualUp did not perform statistically significantly better than some classification models and LR under within-project setting, and can perform statistically significantly better than them under cross-project setting. Ni et al. [13] found that EASC can statistically significantly outperform ManualUp for cross-project EADP. Based on these experimental results observed in these studies, the ranking instability problem exists, i.e., different researchers offer inconsistent rankings of L2R algorithms as to what is best. Such inconsistent findings make it hard to provide guidance about which L2R algorithms should be used to build module-level EADP models. The main reason of the ranking instability problem in EADP are as follows:

(1) Comparing few L2R algorithms with a small number of datasets. For example, Nguyen et al. [6] investigated five algorithms on five open source software projects. Miletić et al. [11] investigated five algorithms on two Eclipse software projects.

(2) Using datasets without indicating the number of defects. Some studies (e.g., Bennin et al. [7,8], Yan et al. [9], and Ni et al. [13]) used the class label rather than the number of defects of modules to build EADP models, which may impact the prediction accuracy, because the L2R algorithms use the actual number of defects or defect density as the target variable to build EADP models.

(3) Using improper or few evaluation measures. Nguyen et al. [6] and Wang et al. [10] employed Spearman rank correlation coefficient and normalized discounted cumulative gain as the evaluation measure, which is usually used to measure the ranking quality of web search engine algorithms. Bennin et al. [7,8], Mthukumaran et al. [92], Yan et al. [9] and Miletić et al. [11] employed cost effective or P_{opt} to evaluate how many defective modules can be inspected when inspecting top 20% LOC and how accurate the global ranking of the predicted modules according to the predicted defect density is. But the number of inspected modules when inspecting top 20% LOC should also be considered as an important evaluation measure, because inspecting many modules would frequently switch between different modules, and this may increase the actual time and effort spent [54]. Yang et al. [12] employed CLC(Cumulative Lift Chart) and FPA to evaluate how accurate the global ranking of the predicted modules according to the predicted number of defect is, but the found bugs and the number of inspected LOC when inspecting top 20% modules should also be evaluated.

6.2. Comparison with recent studies

We discuss the difference of the conclusions between our study and the three recent studies [9,12,13].

(1) Yan et al. [9] investigated the performance of some classification algorithms, LR (Linear Regression), and ManualUp for EADP on 10 PROMISE datasets when LOC is used as effort, and found that ManualUp can perform statistically significantly better than them under cross-project setting in terms of Recall@20%LOC and P_{opt}@LOC, but ManualUp needed to inspect many modules. As shown in Table 4, when LOC is used as effort, ManualUp achieves the highest median Recall@20%LOC and P_{opt}@LOC values under cross-project setting, and ManualUp needs to inspect many modules when inspecting 20% LOC. Therefore, we can find that our finding in terms of Recall@20%LOC, P_{opt}@LOC, and PMI@20%LOC is consistent with that of Yan et al. [9]. However, Yan et al. [9] only employed Recall@20%LOC, P_{opt}@LOC, and PMI@20%LOC as the evaluation measures, and the scope of the study limited to one type of L2R algorithm (i.e., pointwise L2R algorithm). Therefore, they did not find that ManualUp performed badly in terms of Precision@20%LOC and IFA@20%LOC, and LTR-linear performed well under cross-release setting and Ranking SVM performed well under cross-project setting.

(2) Yang et al. [12] investigated the performance of LAR, RR, NBR, PCR, RFR, LTR-linear for EADP on 41 PROMISE datasets when module is used as effort, and found that RFR performs the best in terms of CLC and FPA under cross-release setting. As shown in Table 3, only RFR and ManualDown belong to the first rank in terms of P_{opt}@module under cross-release setting. Therefore, we can find that our finding is consistent with that of Yang et al. [12]. However, Yang et al. [12] only employed CLC and FPA as the evaluation measures, and the scope of the study limited to some regression algorithms and one listwise L2R algorithm (i.e., LTR-linear). Therefore, they did not find that ManualDown can achieve higher median P_{opt}@module and PofB@20%module values than RFR, but require to inspect more LOC.

(3) Ni et al. [13] proposed the EASC algorithm for cross-project module-level EADP. The core idea of EASC is that all the predicted modules are divided into two lists: the first list sorts the predicted defective modules in descending order of the predicted defect density, and the second list sorts the predicted non-defective modules in descending order of the predicted defect density. Software testers first inspect the modules in the first list, and then inspect the modules in the second list. The predicted defective modules by naive Bayes contain very large modules, since larger modules are more likely to be defective. The very large modules require a lot of effort cost. When inspecting top 20% LOC, software testers can only inspect a small number of modules. Therefore, the PofB@20%LOC and P_{opt}@LOC values of EASC are low. In other words, EASC can reduce the false alarms, but the downside is reducing the ability to find bugs and obtaining the accurate global ranking of the predicted modules, which are measured in terms of PofB@20%LOC and P_{opt}@LOC. Because the scope of Ni et al.'s [13] study limited to some classification algorithms, they did not find that Ranking SVM outperformed EASC for cross-project EADP. Since LTR-linear under cross-release setting and Ranking SVM under cross-project setting have acceptable Precision@20%LOC and IFA@20%LOC values, **different from Ni et al.'s conclusion** [13], **we recommend these algorithms rather than EASC to build EADP models when LOC is used as effort**.

7. Conclusion

In this paper, we investigate the ranking stability of *learning to rank* (L2R) algorithms for EADP, motivated by inconsistent performance results in the literature. Compared with existing studies, we perform more comprehensive experiments by evaluating 34 algorithms, using a greater number of datasets (i.e., 49 module-level datasets), a greater number of performance measures (i.e., Precision, Recall, IFA, PMI, PLI, PofB, PofB/PMI, PofB/PLI, and P_{opt}), and a more robust statistical method (i.e., the Scott-Knott ESD test) under cross-release and cross-project settings. This comprehensive procedure allows us to find a stable ranking on the relative performance of L2R algorithms in different situations. Experimental results show that: (1) If software testers want to find more bugs and are insensitive to the number of inspected LOC, ManualDown is more desirable for EADP; otherwise, RFR performs better for cross-release EADP model, and LR is recommended for cross-project EADP model when module is used as effort. (2) LTR-linear is recommended for cross-release EADP model, and Ranking SVM is recommended for cross-project EADP model when LOC is considered as effort.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.infsof.2023.107165.

Data availability

Data will be made available on request.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China (61972290), the Project of Sanya Yazhou Bay Science and Technology City, China (SCKJ-JYRC-2022-17), the General Research Fund of the Research Grants, Council of Hong Kong (11208017), the Research Funds of the City University of Hong Kong (7005028, 7005217 and 6000796), and the Funding Support from other Industry Projects, China (9229109, 9229098, 9229029, 9220103, 9220097, and 9440227), the Youth Fund Project of Hainan Natural Science Foundation, China (622QN344), and the Sanya Science and Education Innovation Park of Wuhan University of Technology, China (2022KF0020).

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] S. Feng, J. Keung, X. Yu, Y. Xiao, M. Zhang, Investigation on the stability of SMOTE-based oversampling techniques in software defect prediction, Inf. Softw. Technol. 139 (2021) 106662.
[3] T. Mende, R. Koschke, Effort-aware defect prediction models, in: 2010 14th European Conference on Software Maintenance and Reengineering, IEEE, 2010, pp. 107–116.
[4] 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.
[5] X. Yang, K. Tang, X. Yao, A learning-to-rank approach to software defect prediction, IEEE Trans. Reliab. 64 (1) (2014) 234–246.
[6] T.T. Nguyen, T.Q. An, V.T. Hai, T.M. Phuong, Similarity-based and rank-based defect prediction, in: 2014 International Conference on Advanced Technologies for Communications (ATC 2014), IEEE, 2014, pp. 321–325.
[7] 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.
[8] K.E. Bennin, J. Keung, A. Monden, Y. Kamei, N. Ubayashi, Investigating the effects of balanced training and testing datasets on effort-aware fault prediction models, in: 2016 IEEE 40th Annual Computer Software and Applications Conference, Vol. 1, COMPSAC, IEEE, 2016, pp. 154–163.
[9] 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.
[10] F. Wang, J. Huang, Y. Ma, A top-k learning to rank approach to cross-project software defect prediction, in: 2018 25th Asia-Pacific Software Engineering Conference, APSEC, IEEE, 2018, pp. 335–344.
[11] M. Miletić, M. Vukušić, G. Mauša, T.G. Grbac, Cross-release code churn impact on effort-aware software defect prediction, in: 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO, IEEE, 2018, pp. 1460–1466.
[12] X. Yang, W. Wen, Ridge and lasso regression models for cross-version defect prediction, IEEE Trans. Reliab. 67 (3) (2018) 885–896.
[13] C. Ni, X. Xia, D. Lo, X. Chen, Q. Gu, Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction, IEEE Trans. Softw. Eng. (2020).
[14] T. Menzies, O. Jalali, J. Hihn, D. Baker, K. Lum, Stable rankings for different effort models, Autom. Softw. Eng. 17 (4) (2010) 409–437.
[15] J. Keung, E. Kocaguneli, T. Menzies, A ranking stability indicator for selecting the best effort estimator in software cost estimation, Autom. Softw. Eng. (2011).
[16] P. Phannachitta, J. Keung, A. Monden, K. Matsumoto, A stability assessment of solution adaptation techniques for analogy-based software effort estimation, Empir. Softw. Eng. 22 (1) (2017) 474–504.
[17] C. Tantithamthavorn, ScottKnottESD: The scott-knott effect size difference (ESD) test, R Package Version 2 (2016).
[18] 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.
[19] C. Tantithamthavorn, S. McIntosh, A.E. Hassan, A. Ihara, K. Matsumoto, The impact of mislabelling on the performance and interpretation of defect prediction models, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1, IEEE, 2015, pp. 812–823.
[20] G. You, F. Wang, Y. Ma, An empirical study of ranking-oriented cross-project software defect prediction, Int. J. Softw. Eng. Knowl. Eng. 26 (09n10) (2016) 1511–1538.
[21] T.Y. Liu, Learning to Rank for Information Retrieval, Springer Science & Business Media, 2011.
[22] I. Rish, et al., An empirical study of the naive Bayes classifier, in: IJCAI 2001 Workshop on Empirical Methods in Artificial Intelligence, Vol. 3, (22) 2001, pp. 41–46.
[23] D.W. Hosmer Jr., S. Lemeshow, R.X. Sturdivant, Applied Logistic Regression, Vol. 398, John Wiley & Sons, 2013.
[24] N. Landwehr, M. Hall, E. Frank, Logistic model trees, Mach. Learn. 59 (1–2) (2005) 161–205.
[25] S.B. Kotsiantis, Logitboost of simple bayesian classifier, Informatica 29 (1) (2005).
[26] J. Park, I.W. Sandberg, Universal approximation using radial-basis-function networks, Neural Comput. 3 (2) (1991) 246–257.
[27] J. Platt, Sequential minimal optimization: A fast algorithm for training support vector machines, 1998.
[28] L. Breiman, Classification and Regression Trees, Routledge, 2017.
[29] J.R. Quinlan, C4. 5: Programs for Machine Learning, Elsevier, 2014.
[30] A. Liaw, M. Wiener, et al., Classification and regression by randomforest, R News 2 (3) (2002) 18–22.
[31] L.E. Peterson, K-nearest neighbor, Scholarpedia 4 (2) (2009) 1883.
[32] W.W. Cohen, Repeated incremental pruning to produce error reduction, in: Machine Learning Proceedings of the Twelfth International Conference ML95, 1995.
[33] P. Compton, G. Edwards, B. Kang, L. Lazarus, R. Malor, T. Menzies, P. Preston, A. Srinivasan, C. Sammut, Ripple down rules: possibilities and limitations, in: Proceedings of the Sixth AAAI Knowledge Acquisition for Knowledge-Based Systems Workshop, Calgary, Canada, University of Calgary, 1991, 6–1.
[34] M. Xu, P. Watanachaturaporn, P.K. Varshney, M.K. Arora, Decision tree regression for soft classification of remote sensing data, Remote Sens. Environ. 97 (3) (2005) 322–336.
[35] M.R. Segal, Machine learning benchmarks and random forest regression, 2004.
[36] G.A. Seber, A.J. Lee, Linear Regression Analysis, Vol. 329, John Wiley & Sons, 2012.
[37] A.E. Hoerl, R.W. Kennard, Ridge regression: Biased estimation for nonorthogonal problems, Technometrics 12 (1) (1970) 55–67.
[38] B. Efron, T. Hastie, I. Johnstone, R. Tibshirani, et al., Least angle regression, Ann. Statist. 32 (2) (2004) 407–499.
[39] J.R. Koza, Genetic programming, 1997.
[40] D.F. Specht, A general regression neural network, IEEE Trans. Neural Netw. 2 (6) (1991) 568–576.
[41] H. Drucker, C.J. Burges, L. Kaufman, A.J. Smola, V. Vapnik, Support vector regression machines, in: Advances in Neural Information Processing Systems, 1997, pp. 155–161.
[42] M.E. Tipping, The relevance vector machine, in: Advances in Neural Information Processing Systems, 2000, pp. 652–658.
[43] J.H. Friedman, et al., Multivariate adaptive regression splines, Ann. Statist. 19 (1) (1991) 1–67.
[44] M. Maltamo, A. Kangas, Methods based on k-nearest neighbor regression in the prediction of basal area diameter distribution, Can. J. Forest Res. 28 (8) (1998) 1107–1115.
[45] J.G. Cleary, L.E. Trigg, K*: An instance-based learner using an entropic distance measure, in: Machine Learning Proceedings 1995, Elsevier, 1995, pp. 108–114.
[46] R. Herbrich, T. Graepel, K. Obermayer, Large margin rank boundaries for ordinal regression, Adv. Neural Inf. Process. Syst. (2000).
[47] Y. Freund, R. Iyer, R.E. Schapire, Y. Singer, An efficient boosting algorithm for combining preferences, J. Mach. Learn. Res. 4 (Nov) (2003) 933–969.
[48] M. Collins, R.E. Schapire, Y. Singer, Logistic regression, AdaBoost and Bregman distances, Mach. Learn. 48 (1–3) (2002) 253–285.
[49] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, G.N. Hullender, Learning to rank using gradient descent, in: Proceedings of the 22nd International Conference on Machine Learning, ICML-05, 2005, pp. 89–96.
[50] Q. Wu, C.J. Burges, K.M. Svore, J. Gao, Adapting boosting for information retrieval measures, Inf. Retr. 13 (3) (2010) 254–270.
[51] Z. Cao, T. Qin, T.Y. Liu, M.F. Tsai, H. Li, Learning to rank: from pairwise approach to listwise approach, in: Proceedings of the 24th International Conference on Machine Learning, ACM, 2007, pp. 129–136.
[52] D. Metzler, W.B. Croft, Linear feature-based models for information retrieval, Inf. Retr. 10 (3) (2007) 257–274.
[53] A. Panichella, C.V. Alexandru, S. Panichella, A. Bacchelli, H.C. Gall, A search-based training algorithm for cost-aware defect prediction, in: Proceedings of the Genetic and Evolutionary Computation Conference 2016, ACM, 2016, pp. 1077–1084.
[54] Q. Huang, X. Xia, D. Lo, Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction, Empir. Softw. Eng. (2018) 1–40.
[55] X. Chen, Y. Zhao, Q. Wang, Z. Yuan, MULTI: Multi-objective effort-aware just-in-time software defect prediction, Inf. Softw. Technol. 93 (2018) 1–13.

[56] 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.
[57] W. Fu, T. Menzies, Revisiting unsupervised learning for defect prediction, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, 2017, pp. 72–83.
[58] G. Koru, H. Liu, D. Zhang, K. El Emam, Testing the theory of relative defect proneness for closed-source software, Empir. Softw. Eng. 15 (6) (2010) 577–598.
[59] A.G. Koru, K. El Emam, D. Zhang, H. Liu, D. Mathew, Theory of relative defect proneness, Empir. Softw. Eng. 13 (5) (2008) 473.
[60] M. Shepperd, Q. Song, Z. Sun, C. Mair, Data quality: Some comments on the nasa software defect datasets, IEEE Trans. Softw. Eng. 39 (9) (2013) 1208–1215.
[61] B. Turhan, T. Menzies, A.B. Bener, J. Di Stefano, On the relative value of cross-company and within-company data for defect prediction, Empir. Softw. Eng. 14 (5) (2009) 540–578.
[62] R. Wu, H. Zhang, S. Kim, S.C. Cheung, Relink: recovering links between bugs and changes, in: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ACM, 2011, pp. 15–25.
[63] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, B. Turhan, The promise repository of empirical software engineering data, 2012.
[64] M. Jureczko, L. Madeyski, Towards identifying software project clusters with regard to defect prediction, in: T. Menzies, G. Koru (Eds.), Proceedings of the 6th International Conference on Predictive Models in Software Engineering, PROMISE 2010, Timisoara, Romania, September 12-13, 2010, ACM, 2010, p. 9.
[65] M. D'Ambros, M. Lanza, R. Robbes, Evaluating defect prediction approaches: a benchmark and an extensive comparison, Empir. Softw. Eng. 17 (4–5) (2012) 531–577.
[66] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects for eclipse, in: Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007), IEEE, 2007, p. 9.
[67] 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.
[68] T.J. Ostrand, E.J. Weyuker, R.M. Bell, Predicting the location and number of faults in large software systems, IEEE Trans. Softw. Eng. 31 (4) (2005) 340–355.
[69] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, H. Leung, Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2016, pp. 157–168.
[70] 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.
[71] 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.
[72] Y. Zhen, J.W. Keung, Y. Xiao, X. Yan, J. Zhi, J. Zhang, On the significance of category prediction for code-comment synchronization, ACM Trans. Softw. Eng. Methodol. (2022).
[73] T. Cheng, K. Zhao, S. Sun, M. Mateen, J. Wen, Effort-aware cross-project just-in-time defect prediction framework for mobile apps, Front. Comput. Sci. (FCS) 16 (6) (2022) 1–15.
[74] 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.
[75] K. Zhao, Z. Xu, M. Yan, T. Zhang, D. Yang, W. Li, A comprehensive investigation of the impact of feature selection techniques on crashing fault residence prediction models, Inf. Softw. Technol. (IST) 139 (2021) 106652.
[76] 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.
[77] 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.
[78] 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.
[79] Z. Yang, J. Keung, M.A. Kabir, X. Yu, Y. Tang, M. Zhang, S. Feng, AComNN: Attention enhanced compound neural network for financial time-series forecasting with cross-regional features, Appl. Soft Comput. 111 (2021) 107649.
[80] C. He, J. Wu, Q. Zhang, Characterizing research leadership on geographically weighted collaboration network, Scientometrics 126 (5) (2021) 4005–4037.
[81] Y. Chen, X. Lu, X. Li, Supervised deep hashing with a joint deep network, Pattern Recognit. 105 (2020) 107368.
[82] B. Ghotra, S. McIntosh, A.E. Hassan, Revisiting the impact of classification techniques on the performance of defect prediction models, in: Proceedings of the 37th International Conference on Software Engineering-Vol. 1, IEEE Press, 2015, pp. 789–800.
[83] 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, ACM, 2016, pp. 165–176.
[84] W. Ma, L. Chen, Y. Yang, Y. Zhou, B. Xu, Empirical analysis of network measures for effort-aware fault-proneness prediction, Inf. Softw. Technol. 69 (2016) 50–70.
[85] Y. Qu, Q. Zheng, J. Chi, Y. Jin, A. He, D. Cui, H. Zhang, T. Liu, Using K-core decomposition on class dependency networks to improve bug prediction model's practical performance, IEEE Trans. Softw. Eng. (2019).
[86] Y. Yang, M. Harman, J. Krinke, S. Islam, D. Binkley, Y. Zhou, B. Xu, An empirical study on dependence clusters for effort-aware fault-proneness prediction, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, 2016, pp. 296–307.
[87] 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) (2020) 139–153.
[88] Y. Qu, J. Chi, H. Yin, Leveraging developer information for efficient effort-aware bug prediction, Inf. Softw. Technol. 137 (2021) 106605.
[89] Y. Zhou, Y. Yang, H. Lu, L. Chen, Y. Li, Y. Zhao, J. Qian, B. Xu, How far we have progressed in the journey? an examination of cross-project defect prediction, ACM Trans. Softw. Eng. Methodol. (TOSEM) 27 (1) (2018) 1.
[90] Y. Jiang, B. Cukic, Y. Ma, Techniques for evaluating fault prediction models, Empir. Softw. Eng. 13 (5) (2008) 561–595.
[91] T. Mende, R. Koschke, Revisiting the evaluation of defect prediction models, in: Proceedings of the 5th International Conference on Predictor Models in Software Engineering, ACM, 2009, p. 7.
[92] K. Muthukumaran, N.B. Murthy, G.K. Reddy, P. Talishetti, Testing and code review based effort-aware bug prediction model, in: Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, Springer, 2016, pp. 17–30.
[93] J. Rao, X. Yu, C. Zhang, J. Zhou, J. Xiang, Learning to rank software modules for effort-aware defect prediction, in: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion, QRS-C, IEEE, 2021, pp. 372–380.
[94] X. Du, T. Wang, L. Wang, W. Pan, C. Chai, X. Xu, B. Jiang, J. Wang, CoreBug: improving effort-aware bug prediction in software systems using generalized k-core decomposition in class dependency networks, Axioms 11 (5) (2022) 205.