Revisiting Supervised and Unsupervised Methods for Effort-Aware Cross-Project Defect Prediction

Chao Ni, Xin Xia, David Lo, Xiang Chen, and Qing Gu

Abstract—Cross-project defect prediction (CPDP), aiming to apply defect prediction models built on source projects to a target project, has been an active research topic. A variety of supervised CPDP methods and some simple unsupervised CPDP methods have been proposed. In a recent study, Zhou et al. found that simple unsupervised CPDP methods (i.e., ManualDown and ManualUp) have a prediction performance comparable or even superior to complex supervised CPDP methods. Therefore, they suggested that the ManualDown should be treated as the baseline when considering non-effort-aware performance measures (NPMs) and the ManualUp should be treated as the baseline when considering effort-aware performance measures (EPMs) in future CPDP studies. However, in that work, these unsupervised methods are only compared with existing supervised CPDP methods using a small subset of NPMs, and the prediction results of baselines are directly collected from the primary literatures. Besides, the comparison has not considered other recently proposed EPMs, which consider context switches and developer fatigue due to initial false alarms. These limitations may not give a holistic comparison between the supervised methods and unsupervised methods. In this paper, we aim to revisit Zhou et al.'s study. To the best of our knowledge, we are the first to make a comparison between the existing supervised CPDP methods and the unsupervised methods proposed by Zhou et al. in the same experimental setting when considering both NPMs and EPMs. We also propose an improved supervised CPDP method EASC and make a further comparison with the unsupervised methods. According to the results on 82 projects in terms of 11 performance measures, we find that when considering NPMs, EASC can achieve prediction performance comparable or even superior to unsupervised method ManualDown in most cases. Besides, when considering EPMs, EASC can statistically significantly outperform the unsupervised method ManualUp with a large improvement in terms of Cliff's delta in most cases. Therefore, the supervised CPDP methods are more promising than the unsupervised method in practical application scenarios, since the limitation of testing resource and the impact on developers cannot be ignored in these scenarios.

Index Terms—Defect prediction, cross-project, supervised model, unsupervised model

1 Introduction

Software defect prediction (SDP) [1]–[4] is a hot research topic in software engineering research domain and aims to help prioritizing testing resource allocation by predicting defect-prone program modules in advance. Given the prediction results, a project manager can (1) classify the modules into two categories, high defect-prone or low defect-prone [5], [6], or (2) rank the modules from the highest to lowest in terms of defect-proneness [7], [8]. In both scenarios, more resources can be allocated to perform code inspection or software testing on highly defect-prone program modules. A large number of defect prediction methods have been proposed, which mainly apply machine learning techniques to build prediction model by mining data stored in software repositories (such as version control systems, bug tracking systems) [9], [10]. For a given project, it is common to use the historical project data to build a model. Besides, prior studies have shown that the model can predict defects well on test data if a sufficiently large amount of data is avaliable [11].

However, in practice, it is challenging that sufficient training data is available for a new project. Thus, researchers focus on cross-project defect prediction (CPDP) [3], [6], [12]–[17] which builds a model using training data from other projects (i.e., source projects) to predict defective modules in a particular project (i.e., target project). Many methods have been proposed for CPDP scenario and have achieved promising prediction performance [6], [13], [18]. Most of them are supervised methods which build models with the help of labelled data. Recently, some researchers proposed unsupervised methods [5], [19]. Most recently, Zhou et al. [5] conducted large-scale empirical studies on comparison between unsupervised and supervised methods. Their empirical results showed that the simple module size based methods (i.e., ManualDown and ManualUp that predicts the defect-proneness of a module based on the lines of code) have a prediction performance comparable or even superior

- *Chao Ni is with School of Software Technology, Zhejiang University, Ningbo, China and Ningbo Research Institute, Zhejiang University, Ningbo, China and PengCheng Laboratory, Shenzhen, China. E-mail: jacknichao920209@gmail.com.*
- *Xin Xia is with the Faculty of Information Technology, Monash University, Melbourne, Australia. E-mail: xin.xia@monash.edu*
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore. E-mail: davidlo@smu.edu.sg*
- *Xiang Chen is with the School of Information Science and Technology Science, Nantong University, China and Nanjing University, Nanjing, China. E-mail: xchencs@ntu.edu.cn*
- *Qing Gu is with State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China. E-mail: guq@nju.edu.cn*
- *Xin Xia and Qing Gu are the corresponding authors.*

Manuscript received ; revised.

to existing supervised CPDP methods. The result is surprising as supervised models which benefit from historical data are expected to perform better than unsupervised ones. Besides, their findings indicated that previous studies on defect prediction have made a simple problem too complex and consequently have a high influence on two-folds. For practitioners, it will assist in determining whether it is worth to apply the existing supervised CPDP methods in practice. If simple module size methods perform similarly or even better, there seems to have no practical reasons to adopt complex supervised CPDP methods. For researchers, if simple module size methods perform similarly or even better, they strongly need to improve the prediction performance of the existing supervised CPDP methods.

However, there have a few **limitations** in Zhou et al.'s study, such as no implementation of baseline methods, non-uniform performance measures, and no recently proposed effort-aware performance measures. In particular, **First**, Zhou et al. did not re-implement the baseline CPDP methods and just reported the baseline methods' performance values published in corresponding original papers. Researchers may conduct experiments with different default experimental settings [20], which may result in unfair comparisons and consequently draw unreliable conclusions. For example, the experiments in these works [6], [21]–[23] are conducted by Java programming language, and the experiments in these works [24], [25] are conducted by Matlab programming language. All of them are treated as partial baseline methods in Zhou et al.'s work. However, Zhou et al. [5] conducted their own experiments by R programming language. **Second**, different performance measures have been used to investigate the effectiveness of different CPDP methods. In particular, although Zhou et al. discussed a large number of performance measures in their work, they only use a small subset of them in a specific comparison between supervised and unsupervised methods. For example, Ryu et al. [23] just reported AUC measure, and Peters et al. [21] just reported G1 measure. Therefore, Zhou et al. only compared with Ryu et al.'s work in terms of AUC and compared with Peters et al.'s work in terms of G1. They did not compare with these methods in terms of any other performance measures. Limited performance measures can barely provide a holistic comparison of these methods' ability in CPDP scenario. **Third**, recently proposed effort-aware performance measures [26], [27], which consider context switches and developer fatigue due to initial false alarms, have not been considered. Since the limitation of testing resources and the impact on developers cannot be ignored in practice, their comparison should take these measures into consideration.

Considering these limitations and yet the high impact of Zhou et al.'s work [5], we want to revisit their work by conducting a comprehensive comparison between supervised and unsupervised methods **considering the same experimental settings** and a **more comprehensive set of performance measures** especially recently proposed effort-aware performance measures [26], [27].

In this paper, we conduct a revisit study with the help of CrossPare developed by Herbold et al. [28]. CrossPare is the sole benchmark toolkit for cross-project defect prediction comparison and has implemented all these existing baselines in Zhou et al.'s work. We investigate the difference between the top four comprehensive ranking supervised CPDP methods [28], [29] and two unsupervised methods [5] under the same experimental settings. We also take, for a holistic view, recently proposed effort-aware performance measures into consideration to compare supervised and unsupervised methods since the limitation of testing resource and the impact on developers cannot be ignored in practice.

Besides, different types of performance measures are considered for different purposes. Non-effort-aware performance measures (NPMs) consider merely how prediction methods work on projects, while effort-aware performance measures (EPMs) consider not only how prediction methods work on projects but also how the prediction results of methods affect participants. However, the existing CPDP methods barely consider the influences on participants, which will hinder the practical usage of CPDP methods. Therefore, inspired by both Qiao et al.'s [26] and Zhou et al.'s works [5], we would like to propose a new supervised method EASC to boost the performance of a supervised method. Notice EASC differs from Qiao et al.'s work [26], [27]. Qiao et al. proposed their method for change-level within-project defect prediction, while EASC is proposed for file-level cross-project defect prediction.

Our study focuses on answering the following research questions:

RQ1: What are the performance differences between the supervised and unsupervised methods when different types of performance measures are considered?

We revisit the comparison between state-of-the-art supervised CPDP methods and recently proposed unsupervised CPDP methods (i.e., ManualDown and ManualUp) by Zhou et al. [5] considering two types of performance measure: **(1) non-effort-aware performance measures** (i.e., $F1\text{-}score$ [6], [30], [31], AUC [22], [23], [32] and PF [33]–[35]) and **(2) effort-aware performance measures** (i.e., IFA, $PII@L$, $CostEffort@L$ and P_{opt} [27], [36]–[38]).

By revisiting Zhou et al.'s work with the same datasets but more performance measures, we find that in terms of NPMs, ManualDown outperforms the existing state-of-the-art supervised methods in most cases. We further analyze why the unsupervised method performs better than the existing supervised methods in terms of NPMs and figure out that the unsupervised method achieves high performance at the cost of higher inspection effort and higher false alarms, which may cause developer fatigue and tool abandonment. For EPMs, both the existing supervised methods and ManualUp have their own advantages on different performance measures.

RQ2: Could the supervised method be enhanced by leveraging the intuition of unsupervised methods?

We propose an improved supervised CPDP method called EASC and make a deep comparison between EASC and ManualDown (ManualUp) proposed by Zhou et al. [5]. Based on the large-scale experiment on 82 projects, we find that (1) when considering NPMs, supervised method EASC can achieve prediction performance comparable or even superior to unsupervised method ManualDown; (2) when considering EPMs, EASC can statistically significantly outperform ManualUp with a large improvement with respect to the Cliff's delta in most cases.

The main contributions of our paper can be summarized as follows:

(1) We make a comprehensive comparison between supervised CPDP methods and unsupervised CPDP methods (i.e., ManualDown and ManualUp) under the same experiment settings using a more comprehensive set of performance measures.
(2) We perform an in-depth analysis of the experiment results in Zhou et al.'s work, and analyze the reasons why their simple module size method can obtain a prediction performance comparable or even superior to most of the existing supervised CPDP methods. We figure out that unsupervised method achieves high performance measures at the cost of higher inspection cost and high false alarm, which may cause developer fatigue and tool abandonment.
(3) We propose an enhanced supervised method EASC and perform a holistic evaluation on EASC *vs.* unsupervised methods. We find that EASC can outperform unsupervised methods when limited inspection effort is considered.

The remainder of this paper is organized as follows. We describe the problem and the general workflow of cross-project defect prediction in Section 2. We introduce our improved supervised method in Section 3. We describe the non-effort-aware and effort-aware performance measures in Section 4. We present the experimental design, including the datasets, the research setting and the research questions in Section 5. We analyse the experimental results in Section 6. We analyse the potential threats to validity in our empirical studies in Section 7. We summarize related work on cross-project defect prediction in Section 8. We conclude this paper and show future work in Section 9.

2 Problem Statement and General Workflow

Software defect prediction (SDP) [6], [16], [39], [40], a hot research topic in current software engineering research domain, can help to optimize testing resource allocation by predicting defect-prone modules [1] in advance [33]. A large number of defect prediction methods have been proposed, which mainly apply machine learning techniques to build prediction methods by mining data stored in historical software repositories [9], [10], [41]. These methods typically extract various features (i.e., metrics) from repositories, e.g., process features, previous-defect features, source code features, etc., to measure extracted modules and apply a machine learning algorithm to predict if a module is defective or not. Most of the proposed methods work on within-project defect prediction (WPDP) setting, i.e., the prediction models are trained and then applied to modules from the same project. These WPDP methods require sufficient training (historical) data from a project to achieve satisfactory performance.

However, in practice, it is rare that sufficient training data is available for a new project or those projects have a few or even no historical data. Thus, researchers focus on cross-project defect prediction (CPDP) [3], [18], [22], [42]–[44], which builds a model using training data from other projects (i.e., source projects) to predict defective instances in a particular project (i.e., target project). To predict defects in the target project, it follows a two-phase process (i.e., model building phase and model application phase) which is the same as WPDP. In the model building phase, the metric data and the defect data are first collected from the modules in historical releases of source projects. Then, a specific prediction model is built based on these collected data to capture the relationships between the metrics and defect-proneness. In the model application phase, the same metrics are first collected from the target projects. Then, the prediction model built in the previous phase is used to predict the defect-proneness of each module in the target project. After the prediction on the target project, the predicted performance can be evaluated by comparing the predicted defect-proneness with the actual defect information for the target project.

There are at least four variants of CPDP studies, which can be found in the literature [28]: *strict* CPDP, *mixed* CPDP, *mixed-project* defect prediction, and *pair-wise* CPDP. Different types of CPDP may have a different general workflow. In this paper, we consider the setting of **strict CPDP** [32] and its general workflow of the experiment can be found in [28]. For a dataset with information about software products, when one of these software products is selected as the target product, the other products of the dataset are used as the source projects and used for the defect prediction model building. If other revisions of the target product exist in the dataset, they are also discarded such that no information from the same project context remains. For example, consider a dataset D that contains three projects (e.g., P_a, P_b and P_c) and each project has two versions (e.g., 1.0 and 2.0). That means $D = \{P_{a.1.0}, P_{a.2.0}, P_{b.1.0}, P_{b.2.0}, P_{c.1.0}\ and\ P_{c.2.0}\}$. When $P_{c.1.0}$ is selected as the target project, then the rest of the projects except for $P_{c.2.0}$ in D are used as the source projects (i.e., $P_{a.1.0}$, $P_{a.2.0}$, $P_{b.1.0}$ and $P_{b.2.0}$). Besides, we consider the **homogeneous CPDP** as same as Zhou et al.'s work.

3 EASC: An Improved Supervised Method

In this section, we propose an improved and effective supervised method for CPDP scenario: EASC. We first introduce the motivation of EASC, then we present the technical details in the form of pseudo-codes.

Motivation. Labeled data can provide important information for building a model, and previous studies have made significant progress in the CPDP scenario [3], [6], [42]. Therefore, we propose a supervised method based on the following findings in previous works:

- **Finding 1: Unlimited inspection effort.** When inspecting instances without considering inspection effort, a larger instance should be first considered since previous studies report that a larger instance tends to have more defects [5], [45].

[1] The granularity of extracted module can be set as package, class, or code change as needed.

- **Finding 2: Limited inspection efforts.** When inspecting instances, taking into consideration inspection effort, an instance with a larger ratio between each instance defect proneness (i.e., a probability outputted by a classifier) and its inspection effort (i.e., LOC) should be first considered. This is the case since previous studies argue that a smaller instance is proportionally more defect-prone [27], [46]–[48].

Ideally, we can inspect all defect-prone instances without considering inspection effort. However, in practice, we cannot ignore the limitation of inspection effort, context switches and developer fatigue due to initial false alarms. Therefore, we should consider different strategies for different usage scenarios [5]. To benefit from the recent findings of Huang et al. [27] and Zhou et al. [5], we propose EASC (Effort-Aware Supervised Cross-project defect prediction). EASC assumes that for these identified potential defective instances, the instances with higher defect-proneness should be inspected first.

Technical Details. EASC contains two phases: model building phase and model evaluating phase. A model can be built with a specific classifier after some pre-processing in the former phase, while in the latter phase, two types of performance measure will be calculated after the prediction using the specific classifier. The technical details of EASC are presented in Algorithm 1 and Algorithm 2.

ALGORITHM 1: EASC: Model Building Phase

Input:
$projects$: all projects in a specific dataset;
$classifier$: the basic classifier;
$effort$: the available effort to decide whether a instance is defective or not, the default is 20% total lines;
Output:
$Results$: a list which contains all performance pairs of non-effort-aware measures and effort-aware measures (e.g., (NPM,EPM));
1: Filter unsuitable projects from $projects$;
2: **for all** $TestProject$ in $projects$ **do**
3: $TrainSet$ = Set(a copy of $projects$)-Set($TestProject$, **any other versions of** $TestProject$);
4: Build a predictor by using $classifier$ on $TrainSet$;
5: (NPM, EPM)=**EASC:Model Evaluating(**$classifier$,$TestProject$,$effort$**)**, and append them to $Results$;
6: **end for**
7: **return** $Results$.

Algorithm 1 presents the pseudo-code to build a classifier. First, projects will be removed if they do not have the required minimum number of instances (i.e., 5; following the same setting as Herbold et al. [28]) in each class (i.e., defective and non-defective) (Line 1). Then, each qualified project will be treated as the target project (i.e., $TestProject$) in order and be used to evaluate the performance of a built model (Lines 2-6). As we consider the strict CPDP scenario, the $TestProject$ itself and any other versions of the $TestProject$ will be excluded from $TrainSet$ (Line 3). Then, a model can be built with a specific classifier (e.g., Naive Bayes) (Line 4). Followed that, the NPM and EPM performance measures can be obtained and appended to $Results$ after a call of $Model\ Evaluating$ (Line 5). Finally, all NPM and EPM performance values will be returned after the iteration in this dataset (Line 7).

ALGORITHM 2: EASC: Model Evaluating Phase

Input:
$TestProject$: test project to evaluate performance;
$classifier$: the classifier built on training projects;
$effort$: the effort available to decide whether a instance is defective or not;
Output:
NPM: the performance value of non-effort-aware performance measures;
EPM: the performance value of effort-aware performance measures;
1: Initialize $TargetList$, $Defective$, $NonDefective$=ϕ;
2: **for all** $testInstance \in TestProject$ **do**
3: Append $testInstance$ into $Defective$ if $classifier$ predicts it as defective instance, otherwise append $testInstance$ into $NonDefective$;
4: **end for**
$/***Calculating\ EPMs***/$
5: Sort separately instances in $Defective$ and $NonDefective$ in descending order by $score/LOC$;
6: Append $NonDefective$ to the end of $Defective$;
7: Select those instances in front of $Defective$ into $TargetList$ and make sure that the total cost of them accounts for $effort$;
8: Calculate effort-aware performance based on $TargetList$, $Defective$ and $classifier$, then save them into EPM;
$/***CalculatingNPMs***/$
9: Sort $Defective$ in descending order by $score \times LOC$, then calculate non-effort-aware and save them into NPM ;
10: **return** (NPM, EPM).

Algorithm 2 presents the pseudo-code of evaluating a classifier. We first classify potentially defective and non-defective instance with the $classifier$ built on the training dataset (Lines 2-4). When classifying a new instance, the $classifier$ will output a probability $score$, which indicates the defect-proneness of the instance. An instance will be classified as potentially defective if its predicted $score$ is larger than 0.5; otherwise, it will be classified as non-defective. After all the instances in the target project (i.e.,$TestProject$) are predicted, we get two lists (i.e., $Defective$ and $NonDefective$) which contain defective-prone instances and non-defective-prone instances separately. Then we sort the instances in the two lists in descending order (Line 5). In particular, when calculating effort-aware performance measures, we sort the instances in the two lists in descending order by $score/LOC$, in which LOC represents the proxy of inspection effort and $score$ represents the defect-proneness outputted by a classifier. After that, we append the sorted non-defective list to the end of the defective list (Line 6). Then, we select those instances to be inspected into $TargetList$ from the top of combined $Defective$ list with limited inspection cost (i.e., $effort$) (Line 7). After that, effort-aware performance measures can be obtained (Line 8). Followed that, $Defective$, a combination of the original $Defective$ in Line 5 and the original $NonDefective$, are sorted again by $score \times LOC$ in descending order for calculating non-effort-aware performance (Line 9). Finally, two types of performance measures will be returned (Line 10).

Notice that, inspired by Zhou et al.'s work, we use different strategies for different usage scenarios. In this paper, two scenarios are considered: unlimited inspection efforts and limited inspection effort. Therefore, we use both $score \times LOC$ and $score/LOC$ in our proposed method EASC for the two usage scenarios. In particular, when calcu-

lating NPMs, EASC sorts the instances in descending order by $score \times LOC$ which is consistent with Finding 1, when calculating EPMs, EASC sorts the instances in descending order by $score/LOC$ which is consistent with Finding 2.

4 Evaluation Performance Measures

In this section, we introduce 11 performance measures to comprehensively evaluate the performances of both supervised and unsupervised methods. These measures can be divided into two groups: 3 non-effort-aware performance measures (NPMs) and 8 effort-aware performance measures (EPMs).

We consider the three NPMs since this paper aims to revisit the Zhou et al.'s work. Although Zhou et al. discussed a large number of performance measures in their work, they only used a small subset of them in a specific comparison between supervised and unsupervised methods. Therefore, we use a few but representative performance measures [6], [22], [23], [30]–[32], [45], [49] to compare the difference between supervised and unsupervised methods.

We consider additional eight EPMs since Zhou et al.'s work did not consider most recently proposed EPMs, which can effectively assess the value of the prediction model to developers. Consequently, their work may not give a holistic view on the comparison between supervised and unsupervised methods.

4.1 Non-Effort-Aware Performance Measures

This group includes three widely used performance measures in SDP: $F1\text{-}score$ [6], [30], [31] , AUC [22], [23], [32] and PF [33]–[35], which are the representative of threshold-dependent performance measure and threshold-independent performance measure, respectively [50]. There are four possible outcomes for an instance in a target project: An instance can be classified as defective when it is truly defective (true positive, TP); it can be classified as defective when it is actually non-defective (false positive, FP); it can be classified as non-defective when it is actually defective (false negative, FN); or it can be classified as non-defective and it is truly non-defective (true negative, TN). Therefore, based on the four possible outcomes, $F1\text{-}score$ and PF can be defined as follows:

F1-score: a summary measure that combines both $Precision = \frac{TP}{TP+FP}$ and $Recall = \frac{TP}{TP+FN}$. It is computed as: $F1-score = \frac{2\times Precision\times Recall}{Precision+Recall}$.

PF: The probability of false alarm is defined as the ratio of false positives to all non-defective instances: $PF = \frac{FP}{FP+TN}$.

AUC: the area under the receiver operating characteristic (ROC) curve [51], which is a 2D illustration of true positive rate (TPR) on the y-axis versus false positive rate (FPR) on the x-axis. ROC curve is obtained by varying the classification threshold over all possible values, separating clean and buggy predictions. A well performed predictor provides an AUC value close to 1. The ROC analysis is robust in case of imbalanced class distributions and asymmetric misclassification costs. It also represents the probability that a method will rank a randomly chosen defective module higher than a randomly chosen not defective one.

4.2 Effort-Aware Performance Measures

In practical settings, non-effort-aware performance measures cannot provide enough information to help practitioners to fully evaluate a CPDP method considering limited testing resources. We consider a few additional effort-aware performance measures which are proposed by Qiao et al. [26], [27] and have not been investigated in CPDP scenario. This group includes eight performance measures: IFA [52], [53], $PII@20\%$, $PII@1000$, $PII@2000$ [36], $CostEffort@20\%$, $CostEffort@1000$, $CostEffort@2000$ and P_{opt} [27], [37], [38], [54]. We consider IFA because previous studies [52], [53] have shown that developers are not willing to use the prediction method if the value of IFA is quite large which means the first few recommendations are all false alarms and will seriously affect the confidence of developers. We consider $PII@L$ to measure the additional effort needed due to context switches between instances, since context switching has been shown harmful to developer productivity [36] and thus make developers' work harder. We consider $CostEffort@L$ because we want to find more detective instances under the limited inspection effort. We also take P_{opt} into consideration due to its widely usage in previous works [27], [37], [38], [54].

For the convenience of the subsequent description, we first give some notations to easily define these measures. Suppose we have a dataset with M instances and N defective instances in total. After inspecting L lines of code, suppose we inspected m instances and observed n defective instances. Additionally, let's consider that we inspected k instances when we find the first defective instance. Then these evaluation measures can be defined as follows:

IFA: the number of Initial False Alarms encountered before we find the first defective instance. It is computed as: $IFA = k$.

PII@L: Proportion of Instances Inspected when L LOC of all instances are inspected. A high $PII@L$ indicates that, under the same number of LOC to be inspected, developers need to inspect more instances. For example, suppose that team A and team B are planning to investigate instances which have 500 LOC in total. For the team A, they had to review 500 instances where each instance has only 1 LOC. For the team B, they only need to review one instance where this instance has 500 LOC. Apparently, the number of LOC that needs to be inspected by the two teams are the same (i.e., 500 LOC in total). However, developers in the team A would frequently switch between different instances which consequently increase the time cost and effort spent. For example, Meyer et al. [36] conducted a survey with 379 professional software developers and they found that developers perceive their days as productive when they complete many or big tasks without significant interruptions or context switches. Also, a large number of instances may cover many different localities (e.g., hundreds of files and modules), and more coordination and communication between developers with different expertise are required. Thus, the additional effort required due to context switches and additional communication overhead among developers should not be ignored.

Besides, different instances may have different size. For example, some instances may have a hundred of LOC, while

some instances may have a thousand of LOC. Therefore, to comprehensively investigate $PII@L$, two kinds of $PII@L$ are considered: relative LOC of PII and absolute LOC of PII. To the best of our knowledge, this is the first paper that takes these factors into consideration to evaluate effort-aware CPDP methods. $PII@20\%$, $PII@1000$, $PII@2000$ can be computed as follows:

$$PII@20\% = \frac{m}{M},\ where\ L\ accounts\ for\ 20\%\ of\ total\ LOC \quad (1)$$

$$PII@1000 = \frac{m}{M},\ where\ L\ equals\ to\ 1000\ LOC \quad (2)$$

$$PII@2000 = \frac{m}{M},\ where\ L\ equals\ to\ 2000\ LOC \quad (3)$$

Notice that the smaller of these measures' value, the better of these methods' performance.

CostEffort@L: proportion of inspected defective instances among all the actual defective instances when L LOC of all instances are inspected. The high $CostEffort@L$ indicates more defective instances could be detected. Besides, different instances may also have different sizes. Therefore, to comprehensively investigate $CostEffort@L$, two kinds of $PII@L$ are considered: relative LOC $CostEffort$ and absolute LOC of $CostEffort$. To the best of our knowledge, this also is the first paper that takes these factors into consideration to evaluate effort-aware CPDP methods. $CostEffort@20\%$, $CostEffort@1000$, $CostEffort@2000$ can be computed as follows:

$$CostEffort@20\% = \frac{n}{N},\ where\ L\ accounts\ for\ 20\%\ of\ total\ LOC \quad (4)$$

$$CostEffort@1000 = \frac{n}{N},\ where\ L\ equals\ to\ 1000\ LOC \quad (5)$$

$$CostEffort@2000 = \frac{n}{N},\ where\ L\ equals\ to\ 2000\ LOC \quad (6)$$

$\boldsymbol{P_{opt}}$: is the normalized version of the effort-aware performance measure originally introduced by Mende and Koschke [54]. The P_{opt} is based on the concept of the "code-churn-based" Alberg diagram [55]. An Alberg diagram (see Figure 1 for an example) shows the relationship between the number of defect-including instance (e.g., y-axis) obtained by a prediction model and the inspection cost for specific prediction model (e.g., the effort $LOCs$ in x-axis). Besides, P_{opt} is widely used effort-aware performance measure in previous works [7], [27], [38], [56], and in their works, the x-axis and y-axis have the same meaning. Therefore, in our paper, we calculate P_{opt} as same as they do.

To compute P_{opt}, two additional curves are included: the optimal model and the worst model. In the optimal model and the worst model, instances are respectively sorted in decreasing and ascending order according to their actual defect densities. The actual prediction model should outperform the random model and try best to get close to the optimal model. For a given prediction model m, its P_{opt} can be computed as: $P_{opt}(m) = 1 - \frac{Area(O,P)}{Area(O,P)+Area(P,R)+Area(R,W)}$. O represents the optimal curve, P represents the prediction curve, R represents the random curve, and W represents the worst curve, respectively. The function $Area(parameter1, parameter2)$ represents the corresponding area between two curves. For example, $Area(O, P)$ represents the area between the optimal curve and the prediction curve. $Area(P, R)$ represents the area between the prediction curve and the random curve, and $Area(R, W)$ represents the area between the random curve and the worst curve. Thus, a larger P_{opt} value means a smaller difference between the prediction model and the optimal model. In this paper, we calculate P_{opt} following the previous works [27], [37], [57] when 20% of the LOCs are inspected.

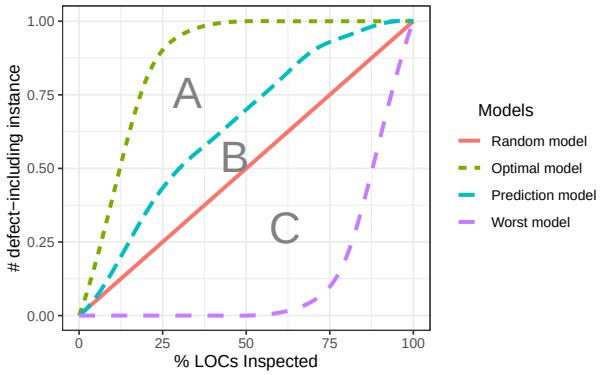


Fig. 1: An example of the relationship between the number of defective instances and the inspection cost for different prediction models.

When calculating EPMs, different methods have different sorting strategies. In particular, for all state-of-the-art CPDP methods, the testing instances will be sorted in descending order of $score$ (i.e., the probability of defect-prone outputted by prediction model). For ManualUp/ManualDown method, the testing instances will be sorted in descending order of risk (i.e., $1/LOC$ for ManualUp and LOC for ManualDown), which is consistent with Zhou et al.'s work [5]. For EASC method, the testing instances are firstly divided into two groups: defective group in which all instances are identified as defective ones, and clean group in which all instances are identified as non-defective ones. Then, instances in the two groups will be sorted in descending order of $score/LOC$, respectively.

For a better understanding of how these methods calculating EPMs (i.e., IFA, $PII@L$ and $CostEffort@L$), we describe the calculating process with an example. The details can be found in the online **APPENDIX A** [58].

5 EXPERIMENTAL SETUP

In this section, we first introduce the characteristics of datasets and then describe the experimental settings. Followed that, we present our research questions.

5.1 Experimental Subjects

In our experimental studies, we evaluate CPDP methods on four publicly available datasets: AEEEM [59], NASA [60], [61], PROMISE [62] and RELINK [63], which are widely used in [6], [21]–[23], [25], [28]. We also give an overview of the datasets, including the number of products, statistical values and the proxies of inspection effort for different datasets. Notice that for a fair comparison and for consistency with Zhou et al.'s work, we also use LOC as the proxy

of inspection effort. The detailed information about these datasets can be found in the online **APPENDIX B** [58].

We note that defect severity and module importance are often taken into consideration when developers perform corrective maintenance efforts. However, there is no information about defect severity or module importance in the four publicly-available and widely-used datasets. Besides, in the current research of SDP, there is no clear instruction about how to incorporate defect severity and importance of software modules into the evaluation process. Therefore, in this paper, we do not take the defect severity or module importance into consideration, which is the same setting that was followed by Zhou et al.'s work.

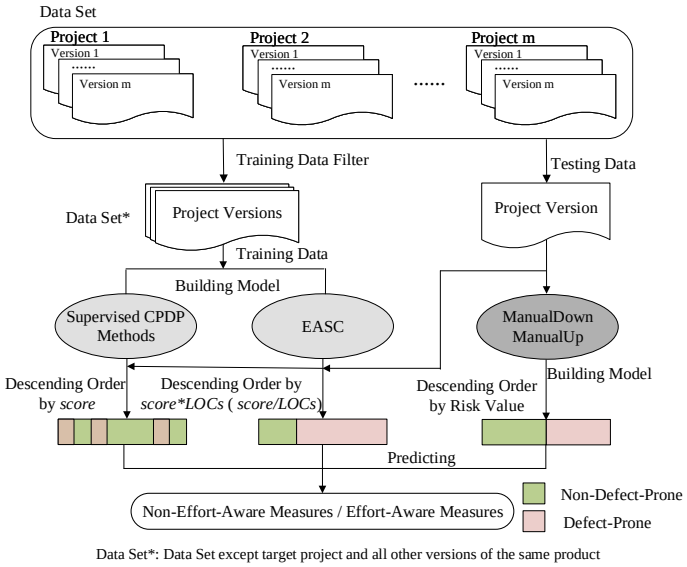


Fig. 2: The workflow of supervised methods and unsupervised methods in CPDP scenario.

5.2 Experiment Setting

5.2.1 Baseline Methods

Selection Criterion. To evaluate the performance of supervised methods and unsupervised methods in different scenarios, we set up strict selection criterion for selecting baseline methods which are considered in our experiments.

Criterion for selecting supervised methods: best performance on both NPMs and EPMs. We choose four methods: CamargoCruz09-DT proposed by Camargo and Cruz [64],Turhan09-DT proposed by Turhan et al. [33], Menzies11-RF proposed by Menzies et al. [65], Watanabe08-DT proposed by Watanabe et al. [66]. The four methods are supervised and their comprehensive good performances have been verified by Herbold et al. [28], [29]. In particular, Herbold et al. [28] conducted a large-scale comprehensive comparison among 24 CPDP methods on 86 projects and measured these CPDP methods with NPMs. According to the results in their work, they found that the four methods perform best in a holistic view in the CPDP scenario. Herbold et al. [29] further investigated how these CPDP methods performed when considering EPMs, and found that the four methods still ranked at the top. Therefore, we choose the four methods as the representatives of supervised methods and use the names of four supervised approaches as same as the ones used in Herbold et al.'s work. The brief introduction to four state-of-the-art supervised methods can be found in the online **APPENDIX C** [58].

Criterion for selecting unsupervised methods: best performance on both NPMs and EPMs. We choose two simple module size methods: ManualDown and ManualUp. The two methods are proposed by Zhou et al. [5] and the concept behinds the two methods can date back to [45], [46]. In particular, ManualDown considers a larger module as more defect-prone, as previous study reports that a larger module tends to have more defects [45]. However, ManualUp considers a smaller module as more defect-prone, as recent studies argue that a smaller module is proportionally more defect-prone and hence should be inspected first [46]–[48]. Zhou et al. found that ManualDown and ManualUp have a prediction performance comparable or even superior to complex supervised CPDP methods. Recently, Chen et al.'s work [43] further confirmed the competitiveness of the two methods over other unsupervised ones.

5.2.2 Methods Implementation and Statistical Analysis

To avoid implementation errors, we utilize the CrossPare: a cross project defect prediction tool developed and shared by Herbold et al. [28]. The four supervised methods have been implemented and we use them in this tool without modification. We also extend it to implement ManualDown, ManualUp and EASC. Besides, to overcome a possible bias of randomness in Menzies11-RF, we run Menzies11-RF 10 times with different random seeds and report the average.

To check the significance of performance comparison, we conduct the Wilcoxon signed-rank test [67], which is a non-parametric statistical hypothesis test on the performance measures. For all the statistical testings, the null hypotheses are that there is no difference between two prediction methods, and the significance level α is set to 0.05. If p-value is smaller than 0.05, we reject the null hypotheses; otherwise we accept the null hypotheses.

We also use Cliff's delta (δ) [68], which is a non-parametric effect size measure that quantifies the amount of difference between two methods. The range of Cliff's delta is [-1,1]. $|\delta|$ equals to 1 indicates the absence of overlap between two methods. It means all data from one group are higher than that from the other group, and vice versa. $|\delta|$ equals to zero means that the two methods are overlapping completely. We consider delta that are less than 0.147, between 0.147 and 0.33, between 0.33 and 0.474 and above 0.474 as "Negligible (N)", "Small (S)", "Medium (M)", "Large (L)" effect size, respectively following [68].

5.3 Research Questions

Our study explores the following research questions:

RQ1: What are the performance differences between the supervised and unsupervised methods when different types of performance measures are considered?

RQ2: Could the supervised method be enhanced by leveraging the intuition of unsupervised methods?

TABLE 1: Comparisons among supervised methods and ManualDown (ManualUp) on four datasets in terms of non-effort-aware performance measures in the form of average±variance.

Measures	Datasets	CamargoCruz09-DT	Menzies11-RF	Turhan09-DT	Watanabe08-DT	ManualDown	ManualUp
$F1\text{-}score^{\uparrow}$	AEEEM	0.31±0.01	0.27±0.02	0.27±0.00	0.30±0.00	0.39±0.03	0.13±0.00
	NASA	0.09±0.01(L)**	0.12±0.01(L)*	0.16±0.01(L)*	0.11±0.00(L)*	0.27±0.02	0.09±0.01
	PROMISE	0.37±0.02(M)***	0.33±0.03(L)***	0.36±0.03(M)***	0.37±0.01(M)***	0.50±0.03	0.22±0.03
	RELINK	0.54±0.00	0.59±0.03	0.53±0.09	0.49±0.03	0.64±0.01	0.24±0.00
$AUC^{\uparrow}$	AEEEM	0.60±0.01(L)*	0.58±0.00(L)*	0.53±0.00(L)**	0.59±0.00(L)*	0.73±0.00	0.27±0.00
	NASA	0.70±0.01	0.53±0.00(L)***	0.62±0.00(L)***	0.67±0.01	0.74±0.01	0.26±0.01
	PROMISE	0.58±0.01(L)***	0.59±0.01(L)***	0.59±0.01(L)***	0.59±0.01(L)***	0.73±0.01	0.27±0.01
	RELINK	0.65±0.00	0.68±0.01	0.63±0.01	0.60±0.02	0.74±0.01	0.26±0.00
$PF^{\downarrow}$	AEEEM	**0.06±0.00**(L)**	**0.04±0.00**(L)**	**0.13±0.01**(L)**	**0.11±0.00**(L)**	0.43±0.00	0.56±0.00
	NASA	**0.01±0.00**(L)***	**0.03±0.00**(L)***	**0.05±0.00**(L)***	**0.02±0.00**(L)***	0.46±0.00	0.53±0.00
	PROMISE	**0.20±0.01**(L)***	**0.13±0.01**(L)***	**0.18±0.01**(L)***	**0.26±0.02**(L)***	0.38±0.01	0.61±0.01
	RELINK	0.21±0.01	0.20±0.00	0.17±0.02	0.17±0.01	0.33±0.01	0.64±0.00

Notes: (1) *** means $p < 0.001$, ** means $p < 0.01$, * means $p < 0.05$.
(2) L/M/S: Large/Medium/Small effect size according to Cliff's delta.
(3) '↓' indicates 'the smaller the better'; '↑' indicates 'the larger the better'.

6 EXPERIMENT RESULTS

In this section, we first report in detail the experimental results in terms of the comparison of the existing supervised CPDP methods and unsupervised CPDP methods (i.e., ManualDown and ManualUp). Then, we make a deep comparison between supervised method EASC proposed in this paper and the unsupervised methods.

6.1 RQ1: What are the performance differences between the supervised and unsupervised methods when different types of performance measures are considered?

Motivation. In the work of Zhou et al. [5], they compared the performance of state-of-the-art supervised methods proposed for the CPDP scenario and two novel unsupervised methods proposed by themselves. They concluded that the simple module size methods have a prediction performance comparable or even superior to most of the existing CPDP methods in the literature, including many newly proposed models. However, there are a few limitations introduced in Section 1 in Zhou et al.'s study. Considering these limitations, we want to conduct a comprehensive comparison between supervised and unsupervised methods using the **same experimental settings** and the **same performance measures**.

Method. In practical applications, these NPMs (i.e., $F1\text{-}score$, AUC and PF) cannot provide enough information to help practitioners fully evaluate a prediction method especially when the testing resources are limited. Thus, we consider a few additional EPMs, namely IFA, $PII@L$, $CostEffort@L$ (i.e., L equals to 20%, 1000 or 2000) and P_{opt}.

To answer this RQ, we investigate two specific sub-questions:

- **Question 1: What is the performance difference between unsupervised methods and supervised methods?**
- **Question 2: What is the relationship between the inspection effort and instance quality?**

In **Question 1**, we replicate the comparison between state-of-the-art supervised CPDP methods and unsupervised CPDP methods recently proposed by Zhou et al. [5]. To avoid implementation errors and make the comparison fairer, we comprehensively use CrossPare [28], a tool for benchmarking CPDP, since it has implemented a large number of CPDP methods and provided a full analysis in terms of different performance measures. Besides, based on this tool, we implement EASC, ManualDown and ManualUp. We consider four CPDP methods due to their overall better performance than other alternatives as confirmed by a previous work [28]: CamargoCruz09-DT, Turhan09-DT, Menzies11-RF and Watanabe08-DT. Besides, all classical classification performance measures and recently proposed performance measures are considered. The workflow of these methods are presented in Figure 2.

In **Question 2**, we explore the characteristics of dataset in each project. We want to explore how unsupervised methods (i.e., ManualDown and ManualUp) perform on the four datasets, and analyze why or why not unsupervised method outperforms the supervised methods. Based on our intuition, larger instances have higher possibility to be defective. Therefore, we want to analyse the relationship between instance inspection effort (e.g., LOC) and instance quality (e.g., defective or non-defective). Notice that the definition of inspection effort for each dataset can be found in the online **APPENDIX B** [58]. We sort instances of each project in descending/ascending order of inspection effort (e.g., LOC) and want to figure out whether unsupervised method requires developers to inspect more instances.

Results for Question 1:

What is the performance difference between unsupervised methods and supervised methods?

To present the result in a comprehensible way, Table 1 and Table 2 present the average results [2] (i.e., the mean performance value) of each method following previous works [6], [26], [27], [38] and statistical analysis results of supervised and unsupervised methods when NPMs and EPMs are considered, respectively. In both of the two tables, the first column lists the performance measures. The second column lists the datasets we experiment on. In the following four columns, the average performance of four supervised methods are given. The last two columns list the average performance of ManualDown and ManualUp. For columns of supervised methods, we use different ways to present the statistical analysis results. In particular, the cells are in **bold** if the supervised method is significantly superior to the

[2] All the average performance in this paper represents the mean of performance value in statistics.

unsupervised method, the cells are in underline if the supervised method is significantly inferior to the unsupervised method. Besides, we use different number of symbol "*" to represent the level of p-value (i.e., *** means $p < 0.001$, ** means $p < 0.01$, * means $p < 0.05$). The effect sizes are also indicated using the "L/M/S" character, which correspondingly represents the Large/Medium/Small effect size according to Cliff's delta.

Notice that Zhou et al. [5] proposed two simple size based methods ManualDown and ManualUp. They concluded that ManualDown has better performance on NPMs, while ManualUp has better performance on EPMs. Therefore, they suggested ManualDown should be treated as a baseline method when considering NPMs, while ManualUp should be treated as a baseline method when considering EPMs. Therefore, for a fair comparison, we present the statistical information among four supervised methods and ManualDown (ManualUp) in Table 1 (Table 2) in terms of NPMs (EPMs). More statistical information between supervised methods and unsupervised methods can be found in the online **APPENDIX D** [58].

Non-effort-aware Performance Measures Comparison. From the results shown in Table 1, we make the following observations:

(1) On average, ManualDown always performs better than ManualUp in terms of all the three NPMs, which is consistent with Zhou et al.'s conclusion.

(2) ManualDown statistically significantly outperforms supervised methods with a large effect size in terms of $F1\text{-}score$ and AUC on the datasets of AEEEM, NASA, and PROMISE in most cases. However, on RELINK, the difference between ManualDown and the supervised methods are not statistically significant.

(3) Supervised methods always perform better than ManualUp in terms of these NPMs.

(4) In terms of AUC, by analyzing the essence of ManualDown and ManualUp, the results seem to confirm that large size modules may have more possibility to be defect-prone.

(5) In terms of PF, supervised methods statistically significantly outperform ManualDown and ManualUp with a large effect size in almost all cases except for RELINK.

Effort-aware Performance Measures Comparison. From the results shown in Table 2 , we make the following observations:

(1) When compared with ManualUp, supervised methods perform statistically significantly better than ManualUp in terms of IFA and $PII@L$ in most cases, and perform worse than ManualUp in terms of $CostEffort@L$ and P_{opt}. It means that in practice ManualUp may cause many false alarms and require much context switch than supervised methods.

(2) When compared with ManualDown, supervised methods perform better than ManualDown in terms of $CostEffort@L$ and P_{opt} in most cases. Besides, ManualDown obtains a better average performance of IFA and $PII@L$. It means that even though ManulDown reduces the number of initial false alarms and the number of context switch, it also reduces the performance of $CostEffort@L$ and P_{opt}, and consequently obtains lower returns.

(3) In terms of EPMs, both ManualDown and ManualUp has their own advantages on different performance measures. In particular, ManualDown has priority over ManualUp in terms of IFA and $PII@L$, while ManualUp has priority over ManualDown in terms of $CostEffort@L$ and P_{opt}. However, in practice, from the perspective of cost, we may not consider ManualDown to inspect larger instances first although it has a good performance of IFA and $PII@L$. We also find that ManualDown obtains a few recalls when inspecting instances with 20% of total effort. Besides, we may also not consider ManualUp as preferred method since it may case developer fatigue due to larger initial false alarms and more context switches.

(4) From the perspective of benefits (e.g., more returns and no consideration of influence on developers), ManualUp outperforms ManualDown since it has a better performance of recall, which is consistent with Zhou et al.'s conclusion.

Results for Question 2:

What is the relationship between instance inspection effort and instance quality?

Firstly, we sort instances of each project based on their inspection effort (i.e., LOC) in **descending** order and analyse the relationship between instance inspection effort and instance quality. The sorting strategy is consistent with ManualDown. The results are shown in Table 3.

In Table 3, the first column lists the name of the dataset. The second column lists the number of projects in this dataset. In the following five columns, we list the percentage of defective instances in the top sorted instances when inspecting $T\%$ of instances. In Zhou et al.'s method, they used 50% as the classification threshold. We list the average results of five different thresholds (i.e., 10%, 20%, 30%, 40% and 50%). Next, we list the total number of defective instances in each dataset. In the following five columns, we list the percentage of effort in the top sorted instances when inspecting $T\%$ of instances. The last column lists total number of inspection effort in each dataset. Take AEEEM as an example (i.e., classification threshold set as 10%), there are five projects with 853 defective instances. Inspecting all instances in AEEEM, it needs to check 639,827 lines of code. When sorting instances in descending order by LOC, on average, we will identify 222 (i.e., 853×0.26) defective instances and inspect 339,108 (i.e., $639,827 \times 0.53$) lines of codes.

According to the results in Table 3, for each dataset, when we sort instances according to its inspection effort (i.e., LOC) in descending order and inspect the top 50% instances, the majority of defective instances (i.e., at least **more than 70%**) will be ranked at the top. As for ManualDown method, the classification threshold is set as 50%, which means the top 50% instances will be classified as defective instances and the rest will be classified as non-defective instances. Therefore, when the classification threshold is set as 50%, ManualDown will obtain a higher $Recall$ (i.e., at least 70% on average), which consequently contributes to a higher AUC. Besides, for a dataset, if the majority of defective instances are ranked in the top 50%, then the majority of non-defective instances are ranked in the rest 50%. ManualDown classifies the instances in the top 50% and the instances in the last 50% as defect-prone and non-defect-

TABLE 2: Comparisons among supervised methods and ManualUp (ManualDown) on four datasets in terms of effort-aware performance measures in the form of average±variance.

Measures	Datasets	CamargoCruz09-DT	Menzies11-RF	Turhan09-DT	Watanabe08-DT	ManualUp	ManualDown
$IFA^{\downarrow}$	AEEEM	**1**±1(L)*	**0**±**0**(L)**	**2**±**5**(L)*	**1**±**1**(L)*	30±417	1±2
	NASA	**33**±**6341**(L)**	**28**±**5637**(L)***	**5**±**43**(L)***	**4**±**39**(L)***	1268±7251939	1±16
	PROMISE	**3**±**114**(L)***	**5**±**194**(L)***	**6**±**205**(L)***	**3**±**50**(L)***	19±473	1±2
	RELINK	0±0	0±0	0±0	1±0	8±1	0±0
$PII@20\%^{\downarrow}$	AEEEM	**0.22**±**0.00**(L)**	**0.22**±**0.00**(L)**	**0.22**±**0.00**(L)**	**0.22**±**0.00**(L)**	0.69±0.00	0.02±0.00
	NASA	**0.18**±**0.00**(L)***	**0.18**±**0.00**(L)***	**0.18**±**0.00**(L)***	**0.18**±**0.00**(L)***	0.54±0.02	0.03±0.00
	PROMISE	**0.22**±**0.00**(L)***	**0.22**±**0.00**(L)***	**0.22**±**0.00**(L)***	**0.22**±**0.00**(L)***	0.68±0.01	0.03±0.00
	RELINK	0.17±0.00	0.17±0.00	0.17±0.00	0.17±0.00	0.68±0.00	0.04±0.00
$PII@1000^{\downarrow}$	AEEEM	**0.02**±**0.00**(L)**	**0.02**±**0.00**(L)**	**0.02**±**0.00**(L)**	**0.02**±**0.00**(L)**	0.19±0.02	0.00±0.00
	NASA	**0.09**±**0.01**(L)**	**0.09**±**0.01**(L)**	**0.09**±**0.01**(L)**	**0.09**±**0.01**(L)**	0.25±0.02	0.02±0.00
	PROMISE	**0.08**±**0.02**(L)***	**0.08**±**0.02**(L)***	**0.08**±**0.02**(L)***	**0.08**±**0.02**(L)***	0.35±0.04	0.03±0.00
	RELINK	0.08±0.00	0.08±0.00	0.08±0.00	0.08±0.00	0.48±0.05	0.02±0.00
$PII@2000^{\downarrow}$	AEEEM	**0.02**±**0.00**(L)**	**0.02**±**0.00**(L)**	**0.02**±**0.00**(L)**	**0.02**±**0.00**(L)**	0.26±0.02	0.00±0.00
	NASA	**0.18**±**0.05**(L)*	**0.18**±**0.05**(L)*	**0.18**±**0.05**(L)*	**0.18**±**0.05**(L)*	0.39±0.05	0.05±0.01
	PROMISE	**0.14**±**0.05**(L)***	**0.14**±**0.05**(L)***	**0.14**±**0.05**(L)***	**0.14**±**0.05**(L)***	0.45±0.05	0.06±0.03
	RELINK	0.18±0.04	0.18±0.04	0.18±0.04	0.18±0.04	0.61±0.07	0.05±0.00
$CostEffort@20\%^{\uparrow}$	AEEEM	0.26±0.00	0.20±0.03	0.24±0.01	0.30±0.01	0.26±0.00	0.05±0.00
	NASA	0.07±0.01	0.09±0.00	0.13±0.01	0.11±0.02	0.18±0.02	0.10±0.00
	PROMISE	0.29±0.02	0.27±0.02	0.28±0.02	0.26±0.01	0.27±0.02	0.08±0.00
	RELINK	0.29±0.00	0.31±0.00	0.26±0.02	0.22±0.00	0.28±0.00	0.09±0.00
$CostEffort@1000^{\uparrow}$	AEEEM	0.04±0.00	0.05±0.00	0.03±0.00(L)*	0.04±0.00	0.08±0.00	0.00±0.00
	NASA	0.06±0.00	0.06±0.00	0.08±0.01	0.05±0.00	0.08±0.02	0.05±0.00
	PROMISE	0.09±0.02(M)***	0.06±0.01(L)***	0.06±0.01(L)***	0.09±0.02(M)***	0.15±0.01	0.05±0.01
	RELINK	0.16±0.02	0.15±0.02	0.14±0.02	0.11±0.01	0.19±0.01	0.06±0.00
$CostEffort@2000^{\uparrow}$	AEEEM	0.05±0.00	0.07±0.00	0.05±0.00(L)*	0.04±0.00(L)*	0.12±0.01	0.01±0.00
	NASA	0.06±0.00	0.08±0.00	0.10±0.01	0.06±0.00	0.11±0.02	0.11±0.02
	PROMISE	0.12±0.02(M)***	0.10±0.02(M)***	0.10±0.03(M)***	0.13±0.03(M)***	0.18±0.02	0.08±0.02
	RELINK	0.21±0.05	0.29±0.10	0.24±0.09	0.29±0.12	0.24±0.00	0.12±0.03
$P_{opt}^{\uparrow}$	AEEEM	0.49±0.01(L)*	0.49±0.02	0.40±0.00(L)**	0.51±0.02	0.65±0.00	0.22±0.03
	NASA	0.34±0.04	0.36±0.04	0.34±0.04(L)*	0.35±0.03	0.49±0.04	0.41±0.04
	PROMISE	0.45±0.04(L)***	0.39±0.03(L)***	0.39±0.04(L)***	0.43±0.05(L)***	0.63±0.04	0.20±0.08
	RELINK	0.51±0.13	0.46±0.04	0.50±0.12	0.62±0.12	0.63±0.00	0.32±0.08

Notes: (1) *** means $p < 0.001$, ** means $p < 0.01$, * means $p < 0.05$.
(2) L/M/S: Large/Medium/Small effect size according to Cliff's delta.
(3) '↓' indicates 'the smaller the better'; '↑' indicates 'the larger the better'.

TABLE 3: The relationship between instance inspection effort and instance quality when sorting testing instances by **ManualDown**.

Dataset	# Project	Percentage of Defects					# Total Defect	Percentage of Efforts					# Total Effort
		10%	20%	30%	40%	50%		10%	20%	30%	40%	50%	
AEEEM	5	0.26	0.44	0.58	0.67	0.74	853	0.53	0.69	0.80	0.87	0.92	639,827
NASA	12	0.34	0.49	0.61	0.70	0.78	3,199	0.44	0.61	0.72	0.80	0.86	630,912
PROMISE	62	0.24	0.40	0.53	0.63	0.72	6,062	0.49	0.66	0.77	0.85	0.91	5,249,888
RELINK	3	0.20	0.36	0.50	0.64	0.72	238	0.45	0.66	0.79	0.87	0.93	76,811

prone instances respectively. Therefore, ManualDown can obtain a higher $Recall$, and consequently obtain a higher $F1\text{-}measure$. Besides, in most cases, ManualDown obtains small values of PF, and only in some cases, ManualDown achieves very high performance of PF, which consequently results in a large average value of PF.

However, when analyzing the percentage of inspection effort in the top 50% instances, the total inspection effort accounts for the majority of all inspection effort (i.e., **at least 86% on average**). Thus, it is clear that unsupervised method ManualDown obtains better NPMs at the cost of higher inspection efforts. The detailed results of each project can be found in the online **APPENDIX E** [58].

Secondly, we sort instances of each project based on their inspection effort (i.e., LOC) in **ascending** order and analyse the relationship between instance inspection effort and instance quality. The sorting strategy is consistent with ManualUp. The results are shown in Table 4.

From the results shown in Table 4, it can be found that inspecting the top 50% instances will consume a few of the total inspection effort. For example, inspecting the top 50% instances of AEEEM, NASA, PROMISE and RELINK needs to consume only 8%, 14%, 10% and 7% of the total inspection effort, respectively. In other words, inspecting instances with 20% of the total inspection effort will inspect much more than 50% of instances. That is the reason why ManualUp performs bad in terms of IFA and $PII@L$ but performs well in terms of $CostEffort@L$. It also can be found that, at least on the four datasets, the smaller instances are more likely to be clean, while the larger instance are more likely to be defective.

TABLE 4: The relationship between instance inspection effort and instance quality when sorting testing instances by **ManualUp**.

Dataset	# Project	Percentage of Defects					# Total Defect	Percentage of Efforts					# Total Effort
		10%	20%	30%	40%	50%		10%	20%	30%	40%	50%	
AEEEM	5	0.05	0.09	0.12	0.19	0.26	853	0	0.01	0.03	0.05	0.08	639,827
NASA	12	0.02	0.04	0.09	0.14	0.2	3,199	0.01	0.03	0.06	0.1	0.14	630,912
PROMISE	62	0.05	0.1	0.15	0.21	0.29	6,062	0	0.01	0.03	0.06	0.1	5,249,888
RELINK	3	0.04	0.1	0.16	0.2	0.28	238	0	0.01	0.02	0.04	0.07	76,811

When considering NPMs, the unsupervised CPDP method ManualDown performs significantly better than supervised methods on most performance measures (i.e., $F1\text{-}score$ and AUC) at the cost of higher inspection efforts and higher false alarms. When considering EPMs, the supervised CPDP methods 1) perform significantly better than the unsupervised method ManualUp on IFA and $PII@L$, and 2) perform significantly worse than the unsupervised method ManualUp on $CostEffort@L$ and P_{opt}. ManualDown always outperforms ManulaUp in terms of NPMs, while ManualDown and ManualUp have their own advantages in terms of EPMs.

6.2 RQ2: Could the supervised method be enhanced by leveraging the intuition of unsupervised methods?

Motivation. Labeled data can provide useful information for building a high-quality model, and previous supervised works have made great progress in the CPDP scenario [3], [6], [13], [42], [69]. Besides, inspired by Zhou et al. [5], we should consider different methods for different scenarios. When the inspection costs are unlimited, we should first consider a larger instance since as previous study reports that a larger instance tends to have more defects [45]. However, in practice, we cannot ignore the limitation of inspection effort, context switches and developer fatigue due to initial false alarms. Therefore, when the inspection costs are limited, inspired by Huang et al. [27], we should first inspect the instances with a larger ratio between each instance defect proneness (i.e., a probability outputted by a classifier) and its inspection effort (i.e., LOC) since recent studies argue that a smaller instance is proportionally more defect-prone and hence should be inspected first [46]–[48]. Consequently, both the findings of Huang et al.' work [27] and Zhou et al.'s work [5] should be further leveraged in future work, and we want to investigate whether there exists an enhanced supervised method having superiority over the unsupervised methods when more NPMs and EPMs are considered in the CPDP scenario.

Method. We first propose an improved supervised method EASC (Effort-Aware Supervised Cross-project defect prediction) which utilizes the advantage of classical supervised methods and takes inspection efforts into consideration. Then, we make a comparison between the supervised method (i.e., EASC) and the unsupervised methods (i.e., ManualDown and ManualUp) when NPMs and EPMs are considered.

For a fair comparison, according to the suggestions of Zhou et al. [5], we should compare EASC with ManualDown when the NPMs are considered, while we should compare EASC with ManualUp when the EPMs are considered. Besides, in previous work [70], Lessmann et al. propose a framework for comparative software defect prediction experiments about the inconsistent findings regarding the superiority among different classifiers. They found that the performance differences of classifier are not significant. Therefore, Naive Bayes is used as the default classifier in EASC and the effect of the choice of EASC's underlying classifier can be found in the online **APPENDIX F** [58].

Besides, Menzies et al. [45] found that $manualUp$ tuned with a defect predictor could achieve better performance. In particular, in the phase of model building, a defect predictor should be trained on training instances. In the phase of model applying, the defect predictor firstly makes a binary decision (e.g., defective or clean) on testing instances. Then, all instances identified as defective are sorted in ascending order of LOC. For convenience, we refer to the tuned $manualUp$ method as $TunedmanualUp$. In this section, we will make a further comparison between EASC and $TunedmanualUp$ in terms of both NPMs and EPMs to figure out whether EASC has priority over $TunedmanualUp$. Notice that Naive Bayes is used as the default classifier in $TunedmanualUp$ and the effect of the choice of $TunedmanualUp$'s underlying classifier can be found in the online **APPENDIX F** [58].

Results 1: Comparison between EASC and ManualDown/ManualUp.

Table 5 and Table 6 present the average results and the statistical test results comparing EASC and ManualDown (ManualUp). In Table 5 and Table 6 , the first column lists the performance measures. The second column lists the datasets we experiment on. The following two columns present the average performance of EASC and ManualDown (ManualUp) in Table 5 (Table 6), respectively. We also present the results of $TunedmanualUp$ in the last one column in Table 5 and Table 6.

Non-effort-aware Performance Comparison. From the results shown in Table 5, in terms of $F1\text{-}score$ and AUC, the supervised method EASC can achieve similar performance with ManualDown (with no statistically significant difference) in almost all datasets except for PROMISE. However, EASC statistically significantly performs better than ManualDown in terms of PF.

Effort-aware Performance Comparison. From the results shown in Table 6 , we make the following observations:

(1) In terms of IFA, EASC achieves the best results on all datasets and statistically significantly improves ManualUp with large effect size on almost all dataset except for RELINK. On average, the IFA scores of EASC are no larger than 6, while those of ManualUp vary in large range (i.e., 8∼1268). For example, on AEEEM, EASC on average can successfully detect the first defective instance with at most

TABLE 5: Comparisons between EASC and ManualDown (TunedmanualUp) on four datasets in terms of non-effort-aware performance measures in the form of average±variance.

Measures	Datasets	EASC	ManualDown	EASC	TunedmanualUp
$F1\text{-}score^{\uparrow}$	AEEEM	0.32±0.02	0.39±0.03	0.32±0.02	0.42±0.02
	NASA	0.26±0.01	0.27±0.02	0.26±0.01	0.30±0.02
	PROMISE	0.28±0.02(L)***	0.50±0.03	0.28±0.02(L)***	0.49±0.03
	RELINK	0.67±0.02	0.64±0.01	0.67±0.02	0.66±0.01
$AUC^{\uparrow}$	AEEEM	0.75±0.00	0.73±0.00	**0.75±0.00**(L)**	0.59±0.00
	NASA	0.77±0.01	0.74±0.01	**0.77±0.01**(L)**	0.60±0.01
	PROMISE	0.73±0.01	0.73±0.01	**0.73±0.01**(L)***	0.60±0.01
	RELINK	0.79±0.01	0.74±0.01	0.79±0.01	0.63±0.01
$PF^{\downarrow}$	AEEEM	**0.07±0.01**(L)**	0.43±0.00	**0.07±0.01**(L)**	0.29±0.03
	NASA	**0.07±0.00**(L)***	0.46±0.00	**0.07±0.00**(L)***	0.47±0.06
	PROMISE	**0.07±0.00**(L)***	0.38±0.01	**0.07±0.00**(L)***	0.28±0.02
	RELINK	0.23±0.05	0.33±0.01	0.23±0.05	0.43±0.02

$Notes$: (1) *** means $p < 0.001$, ** means $p < 0.01$, * means $p < 0.05$.
(2) L/M/S: Large/Medium/Small effect size according to Cliff's delta.
(3) '↓' indicates 'the smaller the better'; '↑' indicates 'the larger the better'.

TABLE 6: Comparisons between EASC and ManualUp (TunedmanualUp) on four datasets in terms of effort-aware performance measures in the form of average±variance.

Measures	Dataset	EASC	ManualUp	EASC	TunedmanualUp
$IFA^{\downarrow}$	AEEEM	**1±2**(L)**	30±417	1±2	2±0
	NASA	**5±50**(L)***	1268±7251939	5±50	21±745
	PROMISE	**6±118**(L)***	20±538	6±118	6±247
	RELINK	1±2	8±1	1±2	4±26
$PII@20\%^{\downarrow}$	AEEEM	**0.08±0.00**(L)**	0.69±0.00	**0.08±0.00**(L)*	0.23±0.02
	NASA	**0.07±0.00**(L)***	0.54±0.02	**0.07±0.00**(L)***	0.25±0.01
	PROMISE	**0.11±0.00**(L)***	0.68±0.01	**0.11±0.00**(L)***	0.24±0.00
	RELINK	0.22±0.01	0.68±0.00	0.22±0.01	0.32±0.01
$PII@1000^{\downarrow}$	AEEEM	**0.02±0.00**(L)**	0.19±0.02	0.02±0.00	0.04±0.00
	NASA	**0.04±0.00**(L)***	0.25±0.02	**0.04±0.00**(L)*	0.14±0.02
	PROMISE	**0.06±0.01**(L)***	0.35±0.04	**0.06±0.01**(L)***	0.08±0.01
	RELINK	0.12±0.00	0.48±0.05	0.12±0.00	0.18±0.01
$PII@2000^{\downarrow}$	AEEEM	**0.02±0.00**(L)**	0.26±0.02	0.02±0.00	0.06±0.00
	NASA	**0.07±0.01**(L)***	0.39±0.05	**0.07±0.01**(L)*	0.22±0.03
	PROMISE	**0.10±0.04**(L)***	0.45±0.05	**0.10±0.04**(L)***	0.16±0.04
	RELINK	0.18±0.00	0.61±0.07	0.18±0.00	0.27±0.01
$CostEffort@20\%^{\uparrow}$	AEEEM	0.18±0.01	0.26±0.00	0.18±0.01	0.31±0.00
	NASA	0.20±0.01	0.18±0.02	0.20±0.01(L)*	0.31±0.02
	PROMISE	0.17±0.01(M)***	0.27±0.02	0.17±0.01(L)*	0.28±0.01
	RELINK	0.33±0.00	0.28±0.00	0.33±0.00	0.38±0.00
$CostEffort@1000^{\uparrow}$	AEEEM	0.04±0.00	0.08±0.00	0.04±0.00	0.07±0.00
	NASA	0.11±0.02	0.08±0.02	0.11±0.02	0.17±0.05
	PROMISE	0.05±0.00(L)***	0.15±0.01	0.05±0.00	0.10±0.02
	RELINK	0.22±0.04	0.19±0.01	0.22±0.04	0.20±0.02
$CostEffort@2000^{\uparrow}$	AEEEM	0.05±0.00	0.12±0.01	0.05±0.00	0.08±0.00
	NASA	0.16±0.02	0.11±0.02	0.16±0.02	0.25±0.08
	PROMISE	0.07±0.01(L)***	0.18±0.02	0.07±0.01	0.15±0.03
	RELINK	0.32±0.06	0.24±0.00	0.32±0.06	0.32±0.06
$P_{opt}^{\uparrow}$	AEEEM	0.73±0.02	0.65±0.00	0.73±0.02	0.63±0.01
	NASA	0.62±0.02	0.49±0.04	0.62±0.02	0.59±0.01
	PROMISE	0.66±0.08	0.63±0.04	0.66±0.08	0.62±0.06
	RELINK	0.74±0.02	0.64±0.00	0.74±0.02	0.58±0.00

$Notes$: (1) *** means $p < 0.001$, ** means $p < 0.01$, * means $p < 0.05$.
(2) L/M/S: Large/Medium/Small effect size according to Cliff's delta.
(3) '↓' indicates 'the smaller the better'; '↑' indicates 'the larger the better'.

one initial false alarm, while ManualUp on average gets 30 initial false alarms before the first defective instance is found. Besides, ManualUp has thousands of initial false alarms on NASA (i.e., 1268) which may cause developer fatigue in using a defect prediction tool.

(2) In terms of $PII@20\%$, EASC statistically significantly outperforms ManualUp with a large improvement with respect to Cliff's delta on almost all datasets except for RELINK. In particular, the performance of ManualUp is many times that of EASC, which may cause more context switches. For a comprehensive comparison with ManualUp, we also consider another two performance measures: $PII@1000$ and $PII@2000$. According to the results in Table 6, we can draw similar conclusions as with $PII@20\%$.

(3) In terms of $CostEffort@20\%$, the difference between EASC and ManualUp are not statistically significant in almost all cases except for PROMISE. In addition, for a comprehensive comparison with ManualUp, we also consider another two performance measures: $CostEffort@1000$ and $CostEffort@2000$. According to the results in Table 6, we can draw similar conclusions as with $PII@20\%$.

(4) In terms of P_{opt}, EASC also outperforms ManualUp in all cases since EASC (i.e., 0.69) obtains higher performance than ManualUp (i.e., 0.60) on average.

Results 2: Comparison between EASC and TunedmanualUp.

Non-effort-aware Performance Comparison. From the results shown in Table 5, we find that EASC achieves similar performance with TunedmanualUp in terms of $F1\text{-}score$ and the difference is not statistically significant except for PROMISE. However, in terms of AUC and PF, EASC statistically significantly outperforms $TunedmanualUp$ with a large improvement with respect to Cliff's delta in most cases.

Effort-aware Performance Comparison. From the results shown in Table 6, we find that in terms of $PII@L$, EASC statistically significantly performs better than $TunedmanualUp$ in most cases. In terms of IFA and P_{opt}, EASC also achieve better average performance than $TunedmanualUp$. On NASA and PROMISE, EASC performs worse than $TunedmanualUp$ in terms of $CostEffort@20\%$. Besides, in terms of $CostEffort@L$, the difference between EASC and $TunedmanualUp$ is not statistically significant.

When considering NPMs, supervised method EASC achieves prediction performance comparable or even superior to unsupervised method ManualDown. When considering EPMs, EASC can significantly outperform ManualUp with a large improvement with respect to Cliff's delta in most cases. Besides, EASC can obtain better performance than TunedmanualUp in most cases in terms of both NPMs and EPMs.

7 THREATS TO VALIDITY

Threats to internal validity relate to faults in the implementation of the methods when we revisit the supervised and unsupervised methods, especially for the unsupervised methods (i.e., ManualDown and ManualUp) which are both published by their authors using R language. To minimize the internal threats, we not only implement these methods by pair programming but also make full use of third-party implementations such as the CrossPare [28] and Weka [71]. We use the default hyper-parameters suggested by CrossPare and Weka. For the unsupervised method, although our code is written in Java, we have carefully read the published paper and strictly follow the description of these methods. All of the datasets used in our paper are publicly available from previous works, and most datasets are cleaned for quality or manually verified in previous works.

Threats to external validity relate to the quality and generalizability of our datasets. We use four datasets with 82 projects, which belong to different application domains, vary in size, cover a long period of time and are written in different programming languages. However, there are still many other projects in other domains using other programming languages, which are not considered in our study. Besides, in our experiment, most of these projects are open source projects. Thus, it is still unclear whether our conclusions are generalizable for commercial projects. In the future, we plan to reduce this threat by considering more additional software projects especial commercial projects.

Threats to construct validity relate to the suitability of our performance measures. In addition to state-of-the-art NPMs, we consider another eight EPMs, namely IFA, $PII@L$, $CoftEffort@L$ and P_{opt}. We use IFA because previous studies have shown that developers are not willing to use the prediction method if its IFA is quite large which will heavily depress the confidence of developer. We use $PII@L$ because the developers are always in heavy work. The high value of $PII@L$ means developers need to inspect more instances under the same inspection effort, which will make developers' work harder. We use $CostEffort$ because we want to find more detective instances under the limited inspection effort. We use P_{opt} because it has been widely used in previous works [27], [37], [38] as the effort-aware performance measure. We have carefully discussed the motivation for using these additional evaluation measures and cited previous studies to support our assumptions. However, it is difficult to accurately measure the inspection effort of an instance in practice. In this paper, we treat number of lines of code inspected as the proxy of inspection effort, which is widely used in previous works [5], [27], [38]. However, number of lines of code inspected may not be appropriate to measure the true effort associated with code inspections activities. In this future, we want to investigate other proxies of inspection effort. Besides, we use the non-parametric statistical hypothesis Wilcoxon signed-rank test and compute non-parametric effect size measure Cliff's δ to compare the performance of different methods, and ensure that the differences are statistically significant and substantial. These tests have been used by past studies [5], [6]. Thus, we believe we have little threats to construct validity.

8 RELATED WORK

Since the target software projects usually lack the labelled modules, a possible solution is to use other historical projects with labelled modules to train the prediction models. This issue is called the cross-project defect prediction (CPDP) [18], [22], [44]. However, the dataset distribution of the target and source projects is usually different, which makes CPDP a challenging task. Zimmermann et al. [72] conducted a large-scale empirical study to investigate the feasibility of CPDP and their results were not optimistic.

Consequently, many supervised CPDP methods are proposed in past decades to improve the performance of CPDP [5], [28]. Most researchers focus on homogeneous CPDP, which assumes that the source and target projects have the same feature sets. Turhan et al. [33] proposed Burak filter to first transform the metric data with the logarithm and then applied a relevancy filter to the available training data based on the k (i.e., 10) nearest instances algorithm. Through the relevancy filter, the k nearest instances to each instance in the target data are selected. Peters et al. [21] improved the filter mechanism, which took in the infra-structure of source projects. Menzies et al. [65] created a local model through clustering of the training data with the WHICH algorithm. Separate WHICH rules are created for each cluster to create local models. In addition to WHICH, random forest is used in this paper due to its better performance. Ma et al. [22] proposed a method which assigns higher weights to the

source instances that are similar to the target instances. Camargo Cruz and Ochimizu [64] proposed to apply a power transformation to the metric data and then standardize it. The power transformation is based on the logarithm and the observation that software metrics, especially the size and complexity, often follow exponential distributions, which is the same as what Turhan et al. [33] do for the treatment of the data. Besides, they considered a single training product as reference. Watanabe et al. [66] proposed to compensate differences between products through a standardization technique that rescales the data. In a scenario with only one candidate product as training data, they proposed to use this product as reference for the standardization of the target data. This shall increase the homogeneity between the target product and the candidate product. As formula for standardization, the authors proposed to multiply each metric value of the target product with the mean value of the candidate product and divide this by the mean of the target product itself. Wang et al. [73] leveraged a representation-learning algorithm (i.e., deep learning) to learn semantic representation of the modules from the projects. Nam et al. propose [18] TCA+ which extends TCA [74] which transforms data from source and target projects to a latent space where the two datasets are close to each other with some data pre-processing options and a heuristic to decide the best pre-processing option to use. Xia et al. [6] proposed a two-layer framework Hydra, which combined the genetic algorithm and ensemble learning to capture general properties between the source and target projects and merits of multiple prediction models. Zhang et al. [75] investigated seven composite algorithms that integrate multiple machine learning classifiers to improve cross-project defect prediction.

Some researchers investigate heterogeneous CPDP, which assumes that the source and target projects have different feature sets. Nam and Kim [16] proposed the heterogeneous CPDP method, including feature selection phase and feature mapping phase. Jing et al. [76] solved the problem by defining unified feature space and applying CCA (Canonical Correlation Analysis)-based transfer learning. Li et al. [77] proposed multiple kernel learning and ensemble learning to improve heterogeneous CPDP performance. Then they [78], [79] further studied two importance issues (i.e., privacy preservation and cost) in heterogeneous CPDP.

Other researchers considered unsupervised learning methods. Nam and Kim [19] performed defect prediction on unlabelled data using a cluster based method which has two phases. They further used feature selection and instance selection to remove noises in dataset to improve CLA and proposed CLAMI. Zhang et al. [80] designed a connectivity-based unsupervised prediction method. Recently, Zhou et al. [5] proposed two unsupervised methods (ManualDown and ManualUp) and they suggested that these two simple methods should be set as baseline methods in the future CPDP research.

Ideally, we can inspect all defect-prone instances during the process of development. However, in practice, a developer has a limited time and can only inspect a limited number of lines of code. Therefore, in this paper, we propose an improved supervised method EASC based on the findings of Huang et al. [27] and Zhou et al. [5]. EASC takes both NPMs and EPMs into consideration. The results analyzed in previous Sections prove that supervised methods have priority over unsupervised methods.

9 CONCLUSIONS AND FUTURE WORK

In this paper, we first revisit a comparison between the state-of-the-art supervised CPDP methods and unsupervised methods (i.e., ManualUp and ManualDown) recently proposed by Zhou et al. [5] under **the same experimental settings**. We conduct this experiment based on CrossPare which was developed and shared by Herbold et al. [28] to make CPDP method comparisons easier. The experimental results show that 1) when considering NPMs, the unsupervised method (i.e., ManualDown) performs better than state-of-the-art supervised methods in most cases in terms of $F1\text{-}score$ and AUC; 2) when considering EPMs, the supervised CPDP methods perform better than the unsupervised method (i.e., ManualUp) in most cases in terms of IFA and $PII@L$ while perform worse than ManualUp in terms of $CostEffort@L$ and P_{opt}. We further analyze why the unsupervised method performs better than the existing supervised methods in terms of NPMs and figure out that the unsupervised method achieve higher performance at the cost of higher inspection effort and false alarms which may cause developer fatigue and tool abandonment. In addition, since we cannot ignore the limited inspection efforts in practical applications, we propose an improved supervised method EASC to compare with the unsupervised method especially for the scenario when limited inspection cost is considered. EASC contains two phases: model building phase and model evaluating phase. In the former phase, a model can be built with a specific basic classifier (i.e., Naive Bayes is used as the default classifier) after some pre-processing. In the latter phase, it sorts the testing set in descending order by $score \times LOC$ when considering NPMs, or it separately sorts instances predicted as defective and instances predicted as non-defective in descending order by $score/LOC$ when considering EPMs. In which, $score$ is the probability outputted by a classifier to indicate the proneness of an instance to be defective, and LOC is the inspection effort of an instance. The experimental results proved that EASC can significantly outperform ManualUp in most cases with medium or large effect size and its performance does not heavily rely on the trained classifiers.

In the future, firstly, we plan to collect more datasets, especially datasets gathered from commercial projects, to verify the generality of our empirical results of EASC. Secondly, we plan to design more new EPMs to guide our work on improving the performance in the practical usage scenario.

ACKNOWLEDGMENTS

We would like to thank Herbold et al. [28] for sharing the tool and datasets in their study. This research was partially supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE200100021), the National Natural Science Foundation of China (61872057, 61972192, 61872263 and 61702041), and the

Open Project of State Key Laboratory for Novel Software Technology at Nanjing University (KFKT2019B14).

REFERENCES

[1] Y. Kamei and E. Shihab, "Defect prediction: Accomplishments and future challenges," in Proceedings of 23rd Software Analysis, Evolution, and Reengineering (SANER), vol. 5. IEEE, 2016, pp. 33–45.

[2] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang, "Perceptions, expectations, and challenges in defect prediction," IEEE Transactions on Software Engineering, 2018.

[3] C. Ni, W.-S. Liu, X. Chen, Q. Gu, D.-X. Chen, and Q.-G. Huang, "A cluster based feature selection method for cross-project software defect prediction," Journal of Computer Science and Technology, vol. 32, no. 6, pp. 1090–1107, 2017.

[4] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," in Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE). ACM, 2017, pp. 72–83.

[5] Y. Zhou, Y. Yang, H. Lu, L. Chen, Y. Li, Y. Zhao, J. Qian, and B. Xu, "How far we have progressed in the journey? an examination of cross-project defect prediction," ACM Trans. Softw. Eng. Methodol., vol. 27, no. 1, pp. 1:1–1:51, 2018.

[6] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "Hydra: Massively compositional model for cross-project defect prediction," IEEE Transactions on Software Engineering, vol. 42, no. 10, pp. 977–998, 2016.

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

[8] X. Yang, K. Tang, and X. Yao, "A learning-to-rank approach to software defect prediction," IEEE Transactions on Reliability, vol. 64, no. 1, pp. 234–246, 2014.

[9] S. Kim, E. J. Whitehead Jr, and Y. Zhang, "Classifying software changes: Clean or buggy?" IEEE Transactions on Software Engineering, vol. 34, no. 2, pp. 181–196, 2008.

[10] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," IEEE transactions on software engineering, no. 1, pp. 2–13, 2007.

[11] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," IEEE Transactions on Software Engineering, vol. 38, no. 6, pp. 1276–1304, 2012.

[12] A. Boucher and M. Badri, "Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison," Information and Software Technology, vol. 96, pp. 38–67, 2018.

[13] F. Wu, X.-Y. Jing, Y. Sun, J. Sun, L. Huang, F. Cui, and Y. Sun, "Cross-project and within-project semisupervised software defect prediction: A unified approach," IEEE Transactions on Reliability, 2018.

[14] R. Krishna and T. Menzies, "Bellwethers: A baseline method for transfer learning," IEEE Transactions on Software Engineering, 2018.

[15] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," IEEE Transactions on Software Engineering (TSE), 2017.

[16] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," IEEE Transactions on Software Engineering (TSE), 2017.

[17] X.-Y. Jing, F. Wu, X. Dong, and B. Xu, "An improved sda based defect prediction framework for both within-project and cross-project class-imbalance problems," IEEE Transactions on Software Engineering, vol. 43, no. 4, pp. 321–339, 2017.

[18] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in Proceedings of the 35th International Conference on Software Engineering (ICSE). IEEE, 2013, pp. 382–391.

[19] J. Nam and S. Kim, "Clami: Defect prediction on unlabeled datasets," in Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2015, pp. 452–463.

[20] C. Tantithamthavorn, "Towards a better understanding of the impact of experimental components on defect prediction modelling," in Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016, pp. 867–870.

[21] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in Proceedings of the 10th Working Conference on Mining Software Repositories, 2013, pp. 409–418.

[22] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," Information and Software Technology, vol. 54, no. 3, pp. 248–256, 2012.

[23] D. Ryu, O. Choi, and J. Baik, "Value-cognitive boosting with a support vector machine for cross-project defect prediction," Empirical Software Engineering, vol. 21, no. 1, pp. 43–71, 2016.

[24] B. Turhan, A. Tosun, and A. Bener, "Empirical evaluation of mixed-project defect prediction models," in Proceedings of 37th EUROMICRO Conference on Software Engineering and Advanced Applications. IEEE, 2011, pp. 396–403.

[25] D. Ryu and J. Baik, "Effective multi-objective naïve bayes learning for cross-project defect prediction," Applied Soft Computing, vol. 49, pp. 1062–1077, 2016.

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

[27] ——, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," Empirical Software Engineering, pp. 1–40, 2018.

[28] S. Herbold, A. Trautsch, and J. Grabowski, "A comparative study to benchmark cross-project defect prediction approaches," IEEE Trans. Software Eng., vol. 44, no. 9, pp. 811–833, 2018.

[29] S. Herbold, "Benchmarking cross-project defect prediction approaches with costs metrics," arXiv preprint arXiv:1801.04107, 2018.

[30] J. Stuckman, J. Walden, and R. Scandariato, "The effect of dimensionality reduction on software vulnerability prediction models," IEEE Transactions on Reliability, vol. 66, no. 1, pp. 17–37, 2017.

[31] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," Automated Software Engineering, vol. 19, no. 2, pp. 167–199, 2012.

[32] S. Herbold, A. Trautsch, and J. Grabowski, "Global vs. local models for cross-project defect prediction," Empirical Software Engineering, vol. 22, no. 4, pp. 1866–1902, 2017.

[33] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," Empirical Software Engineering, vol. 14, no. 5, pp. 540–578, 2009.

[34] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang, "Implications of ceiling effects in defect predictors," in Proceedings of the 4th international workshop on Predictor models in software engineering, 2008, pp. 47–54.

[35] W. Fu, T. Menzies, and X. Shen, "Tuning for software analytics: Is it really necessary?" Information and Software Technology, vol. 76, pp. 135–146, 2016.

[36] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2014, pp. 19–29.

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

[38] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," IEEE Transactions on Software Engineering, vol. 39, no. 6, pp. 757–773, 2013.

[39] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, and S. Mensah, "MAHAKIL: diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction," in Proceedings of the 40th International Conference on Software Engineering, 2018, p. 699.

[40] A. Agrawal and T. Menzies, "Is "better data" better than "better data miners"?: on the benefits of tuning SMOTE for defect prediction," in Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 1050–1061.

[41] A. E. Hassan, "Predicting faults using the complexity of code changes," in Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, 2009, pp. 78–88.

[42] C. Ni, W. Liu, Q. Gu, X. Chen, and D. Chen, "Fesch: A feature selection method using clusters of hybrid-data for cross-project defect prediction," in Proceedings of the 41st Annual Computer

Software and Applications Conference (COMPSAC). IEEE, 2017, pp. 51–56.
[43] X. Chen, D. Zhang, Y. Zhao, Z. Cui, and C. Ni, "Software defect number prediction: Unsupervised vs supervised methods," Information and Software Technology, vol. 106, pp. 161–181, 2019.
[44] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the imprecision of cross-project defect prediction," in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, 2012, p. 61.
[45] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," Automated Software Engineering, vol. 17, no. 4, pp. 375–407, 2010.
[46] A. G. Koru, K. El Emam, D. Zhang, H. Liu, and D. Mathew, "Theory of relative defect proneness," Empirical Software Engineering, vol. 13, no. 5, p. 473, 2008.
[47] G. Koru, H. Liu, D. Zhang, and K. El Emam, "Testing the theory of relative defect proneness for closed-source software," Empirical Software Engineering, vol. 15, no. 6, pp. 577–598, 2010.
[48] A. G. Koru, D. Zhang, K. El Emam, and H. Liu, "An investigation into the functional form of the size-defect relationship for software modules," IEEE Transactions on Software Engineering, vol. 35, no. 2, pp. 293–304, 2009.
[49] Y. Jiang, B. Cukic, and T. Menzies, "Fault prediction using early lifecycle data," in Proceeding of the 18th IEEE International Symposium on Software Reliability (ISSRE). IEEE, 2007, pp. 237–246.
[50] C. Tantithamthavorn and A. E. Hassan, "An experience report on defect modelling in practice: pitfalls and challenges," in Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP), F. Paulisch and J. Bosch, Eds. ACM, 2018, pp. 286–295.
[51] J. A. Hanley and B. J. McNeil, "The meaning and use of the area under a receiver operating characteristic (roc) curve." Radiology, vol. 143, no. 1, pp. 29–36, 1982.
[52] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in Proceedings of the international symposium on software testing and analysis. ACM, 2011, pp. 199–209.
[53] P. S. Kochhar, X. Xia, D. Lo, and 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.
[54] T. Mende and R. Koschke, "Effort-aware defect prediction models," in Proceedings of the 14th European Conference on Software Maintenance and Reengineering. IEEE, 2010, pp. 107–116.
[55] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," Journal of Systems and Software, vol. 83, no. 1, pp. 2–17, 2010.
[56] Y. Yang, M. Harman, J. Krinke, S. Islam, D. Binkley, Y. Zhou, and B. Xu, "An empirical study on dependence clusters for effort-aware fault-proneness prediction," in Proceeding of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2016, pp. 296–307.
[57] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in Proceedings of 2010 IEEE International Conference on Software Maintenance. IEEE, 2010, pp. 1–10.
[58] N. Chao, X. Xin, L. David, C. Xiang, and G. Qing, "Online appendix for "revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction"," 2020. [Online]. Available: https://github.com/jacknichao/EASC
[59] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in Proceedings of the 7th Mining Software Repositories (MSR). IEEE, 2010, pp. 31–41.
[60] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the nasa software defect datasets," IEEE Transactions on Software Engineering, vol. 39, no. 9, pp. 1208–1215, 2013.
[61] D. Gray, D. Bowes, N. Davey, and Y. Sun, "The misuse of the nasa metrics data program data sets for automated software defect prediction," in Proceedings of Evaluation & Assessment in Software Engineering, 2011, pp. 96–103.
[62] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in Proceeding of International Conference on Predictive MODELS in Software Engineering, 2010, pp. 1–10.
[63] R. Wu, H. Zhang, S. Kim, and 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.
[64] A. E. Camargo Cruz and K. Ochimizu, "Towards logistic regression models for predicting fault-prone code across software projects," in Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE Computer Society, 2009, pp. 460–463.
[65] T. Menzies, A. Butcher, A. Marcus, and D. Zimmermann, Thomas a nd Cok, "Local vs. global models for effort estimation and defect prediction," in Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, 2011, pp. 343–351.
[66] S. Watanabe, H. Kaiya, and K. Kaijiri, "Adapting a fault prediction model to allow inter languagereuse," in Proceedings of the 4th international workshop on Predictor models in software engineering. ACM, 2008, pp. 19–24.
[67] F. Wilcoxon, "Individual comparisons by ranking methods," Biometrics bulletin, vol. 1, no. 6, pp. 80–83, 1945.
[68] N. Cliff, Ordinal methods for behavioral data analysis. Psychology Press, 2014.
[69] S. Amasaki, "Cross-version defect prediction using cross-project defect prediction approaches: Does it work?" in Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering. ACM, 2018, pp. 32–41.
[70] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," IEEE Trans. Software Eng., vol. 34, no. 4, pp. 485–496, 2008.
[71] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," ACM SIGKDD explorations newsletter, vol. 11, no. 1, pp. 10–18, 2009.
[72] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. ACM, 2009, pp. 91–100.
[73] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in Proceeding of the 38th International Conference on Software Engineering (ICSE). IEEE, 2016, pp. 297–308.
[74] S. J. Pan, I. W. Tsang, J. T. Kwok, and Q. Yang, "Domain adaptation via transfer component analysis," IEEE Transactions on Neural Networks, vol. 22, no. 2, pp. 199–210, 2011.
[75] Y. Zhang, D. Lo, X. Xia, and J. Sun, "Combined classifier for cross-project defect prediction: an extended empirical study," Frontiers of Computer Science, vol. 12, no. 2, pp. 280–296, 2018.
[76] X. Jing, F. Wu, X. Dong, F. Qi, and B. Xu, "Heterogeneous cross-company defect prediction by unified metric representation and cca-based transfer learning," in Proceedings of the 10th Joint Meeting on Foundations of Software Engineering. ACM, 2015, pp. 496–507.
[77] Z. Li, X.-Y. Jing, X. Zhu, and H. Zhang, "Heterogeneous defect prediction through multiple kernel learning and ensemble learning," in Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2017, pp. 91–102.
[78] Z. Li, X.-Y. Jing, X. Zhu, H. Zhang, B. Xu, and S. Ying, "On the multiple sources and privacy preservation issues for heterogeneous defect prediction," IEEE Transactions on Software Engineering (TSE), 2017.
[79] Z. Li, X.-Y. Jing, F. Wu, X. Zhu, B. Xu, and S. Ying, "Cost-sensitive transfer kernel canonical correlation analysis for heterogeneous defect prediction," Automated Software Engineering, vol. 25, no. 2, pp. 201–245, 2018.
[80] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan, "Cross-project defect prediction using a connectivity-based unsupervised classifier," in Proceedings of the 38th International Conference on Software Engineering. ACM, 2016, pp. 309–320.

APPENDICES

for

" Revisiting Supervised and Unsupervised Methods for Effort-Aware Cross-Project Defect Prediction"

Chao Ni, Xin Xia, David Lo, Xiang Chen, and Qing Gu

APPENDIX A
AN EXAMPLE ON HOW TO CALCULATE EPMS

For a better understanding of how these recently proposed methods calculating EPMs (i.e., *IFA*, $PII@L$ and $CostEffort@L$), we describe the calculating process with an example shown in Table 1.

In Table 1, suppose we have a testing dataset, which contains eight instances. The first column represents the original order of these instances. These instances are numbered from 1 to 8 shown in the second column. The LOC of instances and the class label of instances are presented in the following two columns. Then, the predicted score and predicted results of each method are given in the next six columns. Finally, $score/LOC$ is listed in the last column.

- **Sorting strategy for state-of-the-art CPDP methods:** The state-of-the-art CPDP methods sort the testing instances in descending order of $score$ (i.e., the probability of defect-prone outputted by prediction model).
- **Sorting strategy for EASC method:** EASC uses different sorting strategies on testing instances when calculating different types of performance measures. In particular, when calculating EMPs, EASC firstly predicts testing instances, and divides all instances into two groups: Group_Defective and Group_Clean. For instances in Group_Defective, the probability of defect-proneness of each instance is larger than 0.5, while instances in Group_clean have less than 0.5 probability of defect-proneness. After that, all the instances in the two groups will be sorted by $score/LOC$, respectively. Finally, the two groups are combined together. In particular, these instances in Group_Clean will be appended at the end of Group_Defective. The results are shown in Figure 1.

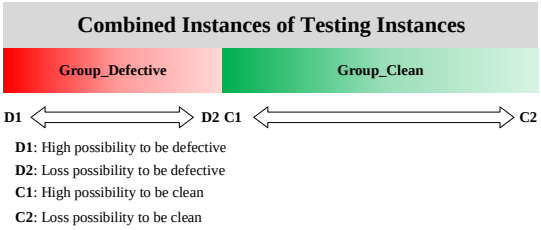


Fig. 1: Sorting Strategy for EASC.

In the above figure, different colors represent different types of instances: defective instance and clean instance. Besides, instances in each group have different color depth. The color depth indicates the defect density of each instance, which to some extent indicates the priority of inspecting these instances.

- **Sorting strategy for ManualDown/ManualUp methods:** Zhou et al. proposed two unsupervised methods in the scenario of cross project defect prediction: ManualDown and ManualUp. For the simplicity of presentation, let m be a module in the testing data, $SizeMetric$ be a module size metric, and $R(m)$ be the predicted risk value of the module m. Formally, the ManualDown method is $R(m) = SizeMetric(m)$, while the ManualUp method is $R(m) = 1/SizeMetric(m)$.

- *Chao Ni is with College of Software Technology, Zhejiang University, Ningbo, China and Ningbo Research Institute, Zhejiang University, Ningbo, China and PengCheng Laboratory, Shenzhen, China. E-mail: jacknichao920209@gmail.com.*
- *Xin Xia is with the Faculty of Information Technology, Monash University, Melbourne, Australia. E-mail: xin.xia@monash.edu*
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore. E-mail: davidlo@smu.edu.sg*
- *Xiang Chen is with the School of Information Science and Technology Science, Nantong University, China and Nanjing University, Nanjing, China. E-mail: xchencs@ntu.edu.cn*
- *Qing Gu is with State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China. E-mail: guq@nju.edu.cn*
- *Xin Xia and Qing Gu are the corresponding authors.*

Manuscript received ; revised.

TABLE 1: An example with 8 testing instances for calculating EPMs

OO	No.	LOC	Actual	Score			Predicted			Score / LOC
				CPDP	EASC	ManualUp	CPDP	EASC	ManualUp	
				Score 1	Score 2	Score 3	Predict 1	Predict 2	Predict 3	
1	①	50	0	0.43	0.22	1/50	0	0	1	4.40E-03
2	②	180	1	0.86	0.97	1/180	1	1	1	5.44E-03
3	③	100	0	0.495	0.12	1/100	0	0	1	1.20E-03
4	④	500	0	0.45	0.33	1/500	0	0	0	6.60E-04
5	⑤	758	1	0.34	0.89	1/758	0	1	0	1.17E-03
6	⑥	1000	0	0.33	0.12	1/1000	0	0	0	1.20E-04
7	⑦	80	1	0.78	0.78	1/80	1	1	1	9.75E-03
8	⑧	210	0	0.32	0.48	1/210	0	0	0	2.29E-03

Notes: (1) OO: Original Order; (2) No.: Instances Number.

For a given testing data, ManualDown considers a larger module as more defect-prone. However, ManualUp considers a smaller module as more defect-prone. Besides, in their work, they use LOC as the $SizeMetric$. Therefore, when calculating NPMs: ManualDown sorts all testing instances in descending order by LOC. Then, ManualDown classifies these instances sorted at the top 50% as defective ones, while ManualDown classifies these instances sorted at the bottom 50% as clean ones.

When calculating EPMs: ManualUp sorts all testing instances in descending order by $1/LOC$. Then, ManualUp classifies these instances sorted at the top 50% as defective ones, while ManualUp classifies these instances sorted at the bottom 50% as clean ones.

In this section, we pay more attention on ManualUp since Zhou et al. suggested ManualUp should be treated as base method when considering EPMs.

According to the sorting strategies of different methods, we give the re-sorted testing data for three methods independently.

- Order of CPDP methods:

Actual Results	①②③④⑤⑥⑦⑧
Sorted Instances order	②⑦③④①⑤⑥⑧
Prediction Results	①②③④⑤⑥⑦⑧
20% of Inspection Effort	②⑦③

- Order of EASC method:

Actual Results	①②③④⑤⑥⑦⑧
Sorted Instances order	⑦②⑤①⑧③④⑥
SCORE / LOC	9.75e-03,5.44e-03,1.17e-03, 4.40e-03,2.29e-03,1.20e-03, 6.60e-04, 1.20e-04
Prediction Results	①②③④⑤⑥⑦⑧
20% of Inspection Effort	⑦②

- Order of ManualUp method:

Actual Results	①②③④⑤⑥⑦⑧
Sorted Instances order	①⑦③②⑧④⑤⑥
1 / LOC	1/50,1/80,1/100,1/180, 1/210,1/500,1/758,1/1000
Prediction Results	①②③④⑤⑥⑦⑧
20% of Inspection Effort	①⑦③②

Therefore, based on the above results, the results of IFA and $PII@20\%$ for these methods are listed as follows:

EPMs	CPDP	EASC	ManualUp
IFA	1	0	1
PII@20%	3/8	2/8	4/8
CostEffort@20%	2/3	2/3	2/3

From the above tables, in terms of IFA, $PII@20\%$ and $CostEffort@20\%$, we find that EASC can achieve a good trade-off performance. In particular, when compared with CPDP and ManualUp, EASC can not only have less false alarm since its IFA is small, but it also has less context switch since its $PII@20\%$ is still small. Besides, all these methods achieve same performance of $CostEffort@20\%$.

APPENDIX B
EXPERIMENTAL SUBJECTS

In our experimental studies, we evaluate CPDP methods on four publicly available datasets, which are also used in [1]–[30]. Table 2 gives an overview of these datasets. The first column reports the name of the group. The second to sixth columns, respectively, report the target project, the project type, the programming language, the brief description, the number of instances contained (one instance corresponding to one module), and the percentage of defective instances. The last column reports the effort metric name in different datasets, which represents the proxy of effort when inspecting whether an instance has a defect or not. Note that, in our experiment, we only consider the **strict cross-project defect prediction scenario**. That means all other versions of a specific product will be excluded for training. Therefore, "Eclipse", investigated in [31], is not considered since the dataset only has one project with different versions.

NASA. The first dataset is the preprocessed version of the NASA Metrics Data Program data provided by Shepperd et al. [32]. The dataset contains information about 12 products from 6 projects. We use the preprocessed version since Shepperd et al. resolved the problems with the consistency of the originally published MDP data noted by Gray et al. [33]. There are 17 static source code metrics used in the dataset. However, information about how the defect labels were created is not available. In our experiment, we use all 12 products from this data set and refer to this dataset as NASA.

AEEEM. The second dataset was shared by D'Ambros et al. [34] and contained data about five Java products from different projects. Sixty one software metrics are considered, including static product metrics, process metrics like the number of defects in previous releases, the entropy of code

changes, and source code churn, as well as the weighted churn and of source code metrics (WCHU) and linearly decayed entropy of source code metrics (LDHH). The defect labels were extracted from the Issue Tracking System (ITS) of the projects. In our experiment, we use all five original products from this dataset and refer to this dataset as AEEEM.

RELINK. The third dataset was shared by Wu et al. [35] and contains defect information about three products from different projects. The dataset has 60 static product metrics and three different defect labels for each module: 1) golden, with manually verified and not automatically labelled defect labels; 2) relink, with defect labels generated with their proposed approach; and 3) traditional heuristic, with an SCM comment based labelling. In our experiment, we use all 3 products with the golden set labelling from this dataset and refer to this dataset as RELINK.

PROMISE. The fourth data set was shared by Jureczko and Madeyski [36]. In our experiment, we use 65 product versions of 32 projects which are provided by Herbold et at. [1]. As for metrics, they collected 20 static product metrics for Java classes, as well as the number of defects that were found in each class. According to [36], all the labels are extracted from the source code management system using a regular expression. Notice that in Herbold et al.'s work, the dataset is named as JURECZKO. In our experiment, we use all 32 products from this dataset and refer to this dataset as PROMISE since it is referred as such in previous works [2], [31], [37], [38].

TABLE 2: The Characteristics of Studied Datasets.

Dataset	Project	Lang.	Description	#Instances	%Defective	Effort Metric
NASA	CM1	C	A spacecraft instrument	344	12.21%	LOC_EXECUTABLE (The number of lines of executable code for a module)
	JM1		A real time C project	9593	18.34%	
	KC1	C++	A storage management system for ground data	2096	15.51%	
	KC3	Java		200	18.00%	
	MC1		A combustion experiment	9277	0.73%	
	MC2	C	A video guidance system	127	34.65%	
	MW1		A zero gravity experiment related to combustion	264	10.23%	
	PC1		A flight software from an earth orbiting satellite	759	8.04%	
	PC2		A dynamic simulator for attitude control systems	1585	1.01%	
	PC3		A flight software from an earth orbiting satellite	1125	12.44%	
	PC4			1399	12.72%	
	PC5			17001	2.96%	
AEEEM	equinox	Java	An OSGi R4 core framework implementation	324	39.81%	numberOfLinesOfCode (Number of lines of code)
	elipse jdt		The Java infrastructure of the Java IDE	997	20.66%	
	lucene		A text search engine library	691	9.26%	
	mylyn		A task management plugin	1862	13.16%	
	pde		A plug-in development environment	1497	13.96%	
RELINK	Apapce Httpd	Java	An open-source HTTP server	194	50.52%	CountLineCode (Number of lines of code)
	OpenIntents Safe		An open intents library	56	39.29%	
	Zxing		A barcode image-processing library	399	29.57%	

continued on the next page

APPENDIX C
BRIEF INTRODUCTION TO FOUR STATE-OF-THE-ART SUPERVISED METHODS

Cruz and Ochimizu [39] proposed to apply a power transformation to the metric data and then standardize it. The power transformation is based on the logarithm and the observation that software metrics, especially the size and complexity, often follow exponential distributions, which is the same as what Turhan et al. [40] do for the treatment of the data. Besides, they only consider training product as a reference. They conducted an experiment on seven Java projects and found that the CPDP methods with standardization can achieve better performance than the corresponding CPDP methods without standardization.

TABLE 2: The Characteristics of Studied Datasets. – continued from previous page

Dataset	Project	Lang.	Description	#Instances	%Defective	Effort Metric
PROMISE	ant 1.3, 1.4 1.5,1.6, 1.7	Java	A build management system	125-745	10.92%-26.21%	loc(Number of lines of code)
	arc		Academic software project developed by 8th or 9th semester computer science students	234	11.54%	
	berek			43	37.21%	
	camel 1.0, 1.2, 1.4, 1.6		A versatile integration framework	339-965	3.83%-35.53%	
	ckjm		A tool for collecting Chidamber and Kemerer metrics	10	50.00%	
	e-learning		Academic software project developed by 8th or 9th semester computer science students	64	7.81%	
	forrest 0.7			29	17.24%	
	ivy 1.1, 1.4, 2.0		A dependency manager	111-352	6.64%-56.76%	
	jedit 3.2, 4.0, 4.1, 4.2, 4.3		A text editor	272-492	2.24%-33.09%	
	kalkulator		Academic software project developed by 8th or 9th semester computer science students	27	22.22%	
	log4j 1.0, 1.1, 1.2		A logging utility	109-205	25.19%-92.20%	
	lucene 2.0, 2.2, 2.4		A text search engine library	195-340	46.67%-59.71%	
	nieruchomosci		Academic software project developed by 8th or 9th semester computer science students	27	37.04%	
	pbeans 1.0, 2.0		An object/relational database mapping framework	26-51	19.61%-76.92%	
	pdftranslator		Academic software project developed by 8th or 9th semester computer science students	33	45.45%	
	poi 1.5, 2.0, 2.5, 3.0		API for Office Open XML standards	237-442	11.78%-64.42%	
	redaktor		A decentralized content management system	176	15.34%	
	serapion		Academic software project developed by 8th or 9th semester computer science students	45	20.00%	
	skarbonka			45	20.00%	
	sklebagd			20	60.00%	
	synapse 1.0, 1.1, 1.2		A high-performance Enterprise Service Bus	157-256	10.19-33.59%	
	systemdata		Academic software project developed by 8th or 9th semester computer science students	65	13.85%	
	szybkafucha			25	56.00%	
	termoproject			42	30.95%	
	tomcat		A Web server	858	8.97%	
	velocity 1.4, 1.5, 1.6		A template language engine	196-229	34.06%-75.00%	
	workflow		Academic software project developed by 8th or 9th semester computer science students	39	51.28%	
	wspomaganiepi			18	66.67%	
	xalan 2.4, 2.5, 2.6, 2.7		An XSLT processor	723-909	15.21%-98.79%	
	xerces 1.0, 1.2, 1.3, 1.4		An XML processor	162-588	15.23%-74.32%	
	zuzel		Academic software project developed by 8th or 9th semester computer science students	29	44.83%	

Turhan et al. [40] proposed Burak filter to first transform the metric data with the logarithm and then applied a relevancy filter to the available training data based on the k (i.e., 10) nearest instances algorithm. Through the relevancy filter, the k nearest instances for each instance in the target data are selected. Notice that repeat selected instance will be used only once. They conducted an experiment on seven products from the NASA dataset and all three products from the SOFTLAB dataset and found that the Burak filter can achieve better performance on $recall$ and pf for both WPDP scenario and CPDP scenario.

Menzies et al. [41] created a local model through clustering of the training data with the WHERE algorithm and afterwards classification of the results with the WHICH rule learning algorithm. Separate WHICH rules are created for each cluster to create local models. In addition to WHICH, random forest is used in this paper due to its better performance. They conducted an experiment on seven products from the PROMISE dataset and observed a further gain in terms of the median over the method using all data.

Watanabe et al. [42] proposed to compensate differences between products through a standardization technique that rescales the data. In a scenario with only one candidate product as training data, they proposed to use this product as the reference for the standardization of the target data. This shall increase the homogeneity between the target product and the candidate product. As a formula for standardization, they proposed to multiply each metric value of the target product with the mean value of the candidate product and divided this by the mean of the target product itself. They conducted an experiment on two projects mined by themselves using seven metrics and bug labels based on the comments in the version control system logs. They observed a little improvement in $recall$ (e.g., 15%) and

$precision$ (e.g., 2%).

APPENDIX D
MORE STATISTICAL COMPARISON RESULT BETWEEN SUPERVISED AND UNSUPERVISED METHODS.

As suggested by Zhou et al., we should treat ManualDown (ManualUp) as baseline method when inspection effort is unlimited (limited). Therefore, for the convenience of reading, we present a part of statistical results between supervised methods and unsupervised methods in main article. For example, when considering NPMs, we only present the statistical test between ManualDown and supervised methods, and when considering EPMs, we only present the statistical test between ManualUp and supervised methods. In this section, for comprehensively understanding the difference between supervised and unsupervised methods, we list more statistical information.

TABLE 3: Comparisons between EASC and **ManualUp** on four datasets in terms of **Non-effort-aware Performance Measures**.

Measures	Datasets	EASC	ManualUp	ManualDown
$F1-score^{\uparrow}$	AEEEM	**0.32±0.02(L)***	013±0.00	0.39±0.03
	NASA	**0.26±0.01(L)****	009±0.01	0.27±0.02
	PROMISE	**0.28±0.02(S)***	022±0.03	0.50±0.03
	RELINK	0.67±0.02	024±0.00	0.64±0.01
$AUC^{\uparrow}$	AEEEM	**0.75±0.00(L)****	027±0.00	0.73±0.00
	NASA	**0.77±0.01(L)*****	026±0.01	0.74±0.01
	PROMISE	**0.73±0.01(L)*****	027±0.01	0.73±0.01
	RELINK	0.79±0.01	026±0.01	0.74±0.01
$False\ Alarm^{\downarrow}$	AEEEM	**0.07±0.01(L)****	056±0.00	0.43±0.00
	NASA	**0.07±0.00(L)*****	053±0.00	0.46±0.00
	PROMISE	**0.07±0.00(L)*****	061±0.01	0.38±0.01
	RELINK	0.23±0.05	064±0.00	0.33±0.01

Notes: (1) *** means $p < 0.001$, ** means $p < 0.01$, * means $p < 0.05$.
(2) L/M/S: Large/Medium/Small effect size according to Cliff's delta.
(3) '↓' indicates 'the smaller the better'; '↑' indicates 'the larger the better'.

Table 3 and Table 4 present the average results and statistical analysis results of supervised and unsupervised methods when NPMs and EPMs are considered, respectively. In both of the two tables, the first column lists the performance measures. The second column lists the datasets we experiment on. In the following column, the average performance values (i.e., the mean performance values) of EASC are given. Then we list the average performance of unsupervised methods. Notice that the results listed in the columns filled with grey is used for comparison. For example, Table 3 mainly lists the comparison of EASC and ManualUp in terms of NPMs. But for convenience, we also list the average result of ManualDown in the last column. Besides, for columns of supervised methods, we use different ways to present the statistical analysis results. In particular, the cells are in **bold** if the supervised method is significantly superior to the unsupervised method, the cells are in underline if the supervised method is significantly inferior to the unsupervised method. Furthermore, we use different number of symbol "*" to represent the level of p-value (i.e., *** means $p < 0.001$, ** means $p < 0.01$, * means $p < 0.05$). The effect sizes are also indicated using the "L/M/S" character, which correspondingly represents the Large/Medium/Small effect size according to Cliff's delta.

From the results shown in Table 3 and Table 4, we make the following observations:

(1) **NPMs**. EASC can statistically significantly outperform ManualUp with a large improvement for almost all cases in terms of $F1\text{-}score$, AUC and $False\ Alarm$.
(2) **EPMs**. ManualDown can statistically significantly outperform EASC in most cases in terms of $PII@L$, while EASC can statistically significantly outperform ManualDown for almost all cases in terms of $CostEffort@L\%$ and P_{opt}. In terms of IFA, ManualDown and EASC obtain similar performance without statistical significance except for PROMISE.

Besides, we also conduct experiments on the comparisons of the four state-of-the-art supervised methods and two unsupervised methods. The results are listed in the Table 5 and Table 6.

From Table 5 and Table 6, we find that:

(1) **NPMs**. The four state-of-the-art supervised methods can statistically significantly outperform ManualUp with a large improvement for almost all cases in terms of $F1\text{-}score$, AUC and $False\ Alarm$.
(2) **EPMs**. ManualDown can statistically significantly outperform the four state-of-the-art supervised methods in most cases in terms of IFA and $PII@L$. In terms of $CostEffort@L\%$ and P_{opt}, four state-of-the-art supervised methods statistically significantly outperform ManualDown in most cases.

APPENDIX E
THE DISTRIBUTION OF INSTANCE INSPECTION EFFORT AND INSTANCE QUALITY ON FOUR DATASETS

In Table 7 to Table 10, the first column lists the dataset name. The second column lists the name of project. In the following five columns, we list the percentage of defective instances in the top sorted instances based on inspection effort. In Zhou et al.'s method, they used 50% as the classification threshold. We list the results of five different thresholds (i.e., 10%, 20%, 30%, 40% and 50%). Next, we list the total number of defective instances in each project. In the following five columns, we list the percentage of effort in the top sorted instances based on inspection effort. Notice that the proxy of inspection effort can be found in **APPENDIX** B. Followed that, we list the total number of inspection and instances in each project respectively. The last column shows the distribution of defective modules by using ManualDown method via visualization technology. Notice that all the instances in each dataset are represented as stripes, and sorted by their inspection effort in descending order from the left side to the right side. In each distribution figure, the pink stripes represent the defective instances, and the green stripes represent the non-defective instances.

For each project, we find that the majority of defective instances (i.e., more than 70%) will be ranked on the top when sorting the instances according to its inspection effort in descending order and inspect the top 50%. In particular, we find that ManualDown can find at least 67%, 68%, 64% and 44% defective instances of total defective instances

TABLE 4: Comparisons between EASC and **ManualDown** on four datasets in terms of **Effort-aware Performance Measures**.

Measures	Dataset	EASC	ManualDown	ManualUp	Measures	Dataset	EASC	ManualDown	ManualUp
$IFA^{\downarrow}$	AEEEM	1±2	1±2	30±417	$P_{opt}@20\%^{\uparrow}$	AEEEM	**0.73±0.02(L)****	0.22±0.03	0.65±0.00
	NASA	5±50	1±16	1268±7251939		NASA	**0.62±0.02(L)***	0.41±0.04	0.49±0.04
	PROMISE	6±118(M)***	1±2	20±538		PROMISE	**0.66±0.08(L)*****	0.20±0.08	0.63±0.04
	RELINK	1±2	0±0	8±1		RELINK	0.74±0.02	0.32±0.08	0.63±0.00
$PII@20\%^{\downarrow}$	AEEEM	0.08±0.00(L)**	0.02±0.00	0.69±0.00	$CostEffort@20\%^{\uparrow}$	AEEEM	**0.18±0.01(L)****	0.05±0.00	0.26±0.00
	NASA	0.07±0.00(L)*	0.03±0.00	0.54±0.02		NASA	**0.20±0.01(L)***	0.10±0.00	0.18±0.02
	PROMISE	0.11±0.00(L)***	0.03±0.00	0.68±0.01		PROMISE	**0.17±0.01(L)*****	0.08±0.00	0.27±0.02
	RELINK	0.22±0.01	0.04±0.00	0.68±0.00		RELINK	0.33±0.00	0.09±0.00	0.28±0.00
$PII@1000^{\downarrow}$	AEEEM	0.02±0.00(L)**	0.00±0.00	0.19±0.02	$CostEffort@1000^{\uparrow}$	AEEEM	**0.04±0.00(L)***	0.00±0.00	0.08±0.00
	NASA	0.04±0.00	0.02±0.00	0.25±0.02		NASA	0.11±0.02	0.05±0.00	0.08±0.02
	PROMISE	0.06±0.01(L)***	0.03±0.00	0.35±0.04		PROMISE	0.05±0.00	0.05±0.01	0.15±0.01
	RELINK	0.12±0.00	0.02±0.00	0.48±0.05		RELINK	0.22±0.04	0.06±0.00	0.19±0.01
$PII@2000^{\downarrow}$	AEEEM	0.02±0.00(L)**	0.00±0.00	0.26±0.02	$CostEffort@2000^{\uparrow}$	AEEEM	**0.05±0.00(L)***	0.01±0.00	0.12±0.01
	NASA	0.07±0.01	0.05±0.01	0.39±0.05		NASA	0.16±0.02	0.11±0.02	0.11±0.02
	PROMISE	0.10±0.04(L)***	0.06±0.03	0.45±0.05		PROMISE	0.07±0.01(S)*	0.08±0.02	0.18±0.02
	RELINK	0.18±0.00	0.05±0.00	0.61±0.07		RELINK	0.32±0.06	0.12±0.03	0.24±0.00

Notes: (1) *** means $p < 0.001$, ** means $p < 0.01$, * means $p < 0.05$.
(2) L/M/S: Large/Medium/Small effect size according to Cliff's delta.
(3) '↓' indicates 'the smaller the better'; '↑' indicates 'the larger the better'.

TABLE 5: Comparisons among supervised methods and **ManualUp** on four datasets in terms of non-effort-aware performance measures.

Measures	Dataset	CamargoCruz09-DT	Menzies11-RF	Turhan09-DT	Watanabe08-DT	ManualUp	ManualDown
$F1-score^{\uparrow}$	AEEEM	**0.31±0.01(L)***	**0.27±0.02(L)***	**0.27±0.00(L)***	**0.30±0.00(L)****	0.13±0.00	0.39±0.03
	NASA	0.09±0.01	0.12±0.01	0.16±0.01	0.11±0.00	0.09±0.01	0.27±0.02
	PROMISE	**0.37±0.02(L)*****	**0.33±0.03(M)*****	**0.36±0.03(M)*****	**0.37±0.01(L)*****	0.22±0.03	0.50±0.03
	RELINK	0.54±0.00	0.59±0.03	0.53±0.03	0.49±0.03	0.24±0.00	0.64±0.01
$AUC^{\uparrow}$	AEEEM	**0.60±0.01(L)****	**0.58±0.00(L)****	**0.53±0.00(L)****	**0.59±0.00(L)****	0.27±0.00	0.73±0.00
	NASA	**0.70±0.01(L)*****	**0.53±0.00(L)*****	**0.62±0.00(L)*****	**0.67±0.01(L)*****	0.26±0.01	0.74±0.01
	PROMISE	**0.58±0.01(L)*****	**0.59±0.01(L)*****	**0.59±0.01(L)*****	**0.59±0.01(L)*****	0.27±0.01	0.73±0.01
	RELINK	0.65±0.00	0.68±0.01	0.63±0.01	0.60±0.02	0.26±0.01	0.74±0.01
$False\ Alarm^{\uparrow}$	AEEEM	**0.06±0.00(L)****	**0.04±0.00(L)****	**0.13±0.01(L)****	**0.11±0.00(L)****	0.56±0.00	0.43±0.00
	NASA	**0.01±0.00(L)*****	**0.03±0.00(L)*****	**0.05±0.00(L)*****	**0.02±0.00(L)*****	0.53±0.00	0.46±0.00
	PROMISE	**0.20±0.01(L)*****	**0.13±0.01(L)*****	**0.18±0.01(L)*****	**0.26±0.02(L)*****	0.61±0.01	0.38±0.01
	RELINK	0.21±0.01	0.20±0.00	0.17±0.02	0.17±0.01	0.64±0.00	0.33±0.01

Notes: (1) *** means $p < 0.001$, ** means $p < 0.01$, * means $p < 0.05$.
(2) L/M/S: Large/Medium/Small effect size according to Cliff's delta.
(3) '↓' indicates 'the smaller the better'; '↑' indicates 'the larger the better'.

on AEEEM, RELINK, NASA and PROMISE respectively. However, we need to spend 87%, 91%, 77% and 80% of total inspection effort on AEEEM, RELINK, NASA and PROMISE respectively. Through the fringe patterns, we can easily illustrate the distribution of defective and non-defective instances. Thus, it is not hard to find that the unsupervised method obtains better performance in terms of NPMs at the cost of higher inspection efforts.

APPENDIX F
THE INFLUENCE OF CLASSIFIER ON EASC AND TUNEDMANUALUP

F.1 The Influence of Classifier on EASC

EASC, a supervised method proposed in this paper, builds a prediction model on the basis of state-of-the-art supervised classifiers and then resorts the testing instances for different practical usages. Therefore, in this section, to investigate the effect of the choice of EASC's underlying classifier, we make a comparison among six commonly used classifiers: Decision Tree, Random Forest, Logistic Regression, Naive Bayes, RBF Network and Support Vector Machine. Table 11 lists the introduction to all basic classifiers used in the paper. The first to third columns are the name of classifier, the abbreviation and brief description respectively. These classifier are widely used in CPDP scenario [7], [49]–[52].

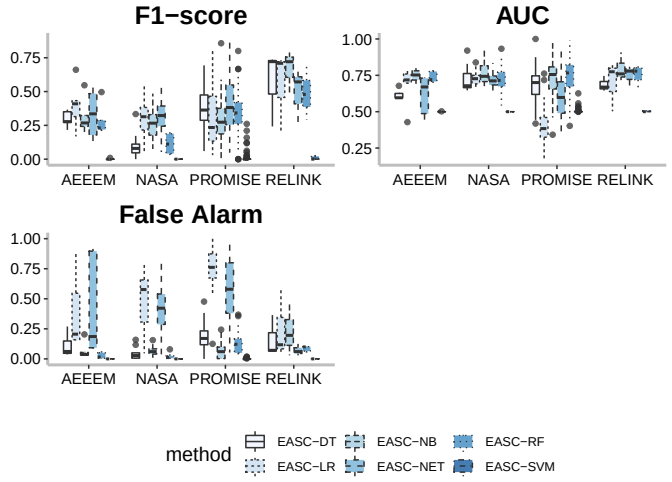


Fig. 2: Comparison of Different Classifiers in EASC in Terms of Non-Effort-Aware Performance Measures.

Figure 2 and Figure 3 show the overall performance of EASC built with different classifiers on different datasets.

Figure 2 and Figure 3 present the value of EASC in terms of 11 performance measures when using six different basic classifiers. Therefore, there are 11 sub-figures and each one illustrates the performance of EASC in terms of a specific

TABLE 6: Comparisons among supervised methods and **ManualDown** on four datasets in terms of **non-effort-aware performance measures**.

Measures	Datasets	CamargoCruz09-DT	Menzies11-RF	Turhan09-DT	Watanabe08-DT	ManualDown	ManualUp
$IFA^{\downarrow}$	AEEEM	1±1	0±0	2±5	1±1	1±2	30±417
	NASA	33±6341	28±5637(M)*	5±43(L)*	4±39	1±16	1268±7251939
	PROMISE	3±114(S)***	5±194(M)***	6±205(S)***	3±50(S)**	1±2	19±473
	RELINK	0±0	0±0	0±0	1±0	0±0	8±1
$PII@20\%^{\downarrow}$	AEEEM	0.22±0.00(L)**	0.22±0.00(L)**	0.22±0.00(L)**	0.22±0.00(L)**	0.02±0.00	0.69±0.00
	NASA	0.18±0.00(L)***	0.18±0.00(L)***	0.18±0.00(L)***	0.18±0.00(L)***	0.03±0.00	0.54±0.02
	PROMISE	0.22±0.00(L)***	0.22±0.00(L)***	0.22±0.00(L)***	0.22±0.00(L)***	0.03±0.00	0.68±0.01
	RELINK	0.17±0.00	0.17±0.00	0.17±0.00	0.17±0.00	0.04±0.00	0.68±0.00
$PII@1000^{\downarrow}$	AEEEM	0.02±0.00(L)**	0.02±0.00(L)**	0.02±0.00(L)**	0.02±0.00(L)**	0.00±0.00	0.19±0.02
	NASA	0.09±0.01(L)*	0.09±0.01(L)*	0.09±0.01(L)*	0.09±0.01(L)*	0.02±0.00	0.25±0.02
	PROMISE	0.08±0.02(L)***	0.08±0.02(L)***	0.08±0.02(L)***	0.08±0.02(L)***	0.03±0.00	0.35±0.04
	RELINK	0.08±0.00	0.08±0.00	0.08±0.00	0.08±0.00	0.02±0.00	0.48±0.05
$PII@2000^{\downarrow}$	AEEEM	0.02±0.00(L)**	0.02±0.00(L)**	0.02±0.00(L)**	0.02±0.00(L)**	0.00±0.00	0.26±0.02
	NASA	0.18±0.05	0.18±0.05	0.18±0.05	0.18±0.05	0.05±0.01	0.39±0.05
	PROMISE	0.14±0.05(L)***	0.14±0.05(L)***	0.14±0.05(L)***	0.14±0.05(L)***	0.06±0.03	0.45±0.05
	RELINK	0.18±0.04	0.18±0.04	0.18±0.04	0.18±0.04	0.05±0.00	0.61±0.07
$CostEffort@20\%^{\uparrow}$	AEEEM	**0.26±0.00(L)****	**0.20±0.03(L)***	0.24±0.01(L)**	**0.30±0.01(L)****	0.05±0.00	0.26±0.00
	NASA	0.07±0.01	0.09±0.00	0.13±0.01	0.11±0.02	0.10±0.00	0.18±0.02
	PROMISE	**0.29±0.02(L)*****	**0.27±0.02(L)*****	**0.28±0.02(L)*****	**0.26±0.01(L)*****	0.08±0.00	0.27±0.02
	RELINK	0.29±0.00	0.31±0.00	0.26±0.02	0.22±0.00	0.09±0.00	0.28±0.00
$CostEffort@1000^{\uparrow}$	AEEEM	**0.04±0.00(L)***	**0.05±0.00(L)***	**0.03±0.00**	**0.04±0.00(L)***	0.00±0.00	0.08±0.00
	NASA	0.06±0.00	0.06±0.00	0.08±0.01	0.05±0.00	0.05±0.00	0.08±0.02
	PROMISE	**0.09±0.02(S)***	0.06±0.01	0.06±0.01	**0.09±0.02(S)***	0.05±0.01	0.15±0.01
	RELINK	0.16±0.02	0.15±0.02	0.14±0.02	0.11±0.01	0.06±0.00	0.19±0.01
$CostEffort@2000^{\uparrow}$	AEEEM	**0.05±0.00(L)***	**0.07±0.00(L)***	**0.05±0.00(L)***	**0.04±0.00(L)***	0.01±0.00	0.12±0.01
	NASA	0.06±0.00	0.08±0.00	0.10±0.01	0.06±0.00	0.11±0.02	0.11±0.02
	PROMISE	**0.12±0.02(S)****	0.10±0.02	0.10±0.03	**0.13±0.03(S)****	0.08±0.02	0.18±0.02
	RELINK	0.21±0.05	0.29±0.10	0.24±0.09	0.29±0.12	0.12±0.03	0.24±0.00
$P_{opt}@20\%^{\uparrow}$	AEEEM	**0.49±0.01(L)***	0.49±0.02	0.40±0.00	**0.51±0.02(L)***	0.22±0.03	0.65±0.00
	NASA	0.34±0.04	0.36±0.04	0.34±0.04	0.35±0.03	0.41±0.04	0.49±0.04
	PROMISE	**0.45±0.04(L)*****	**0.39±0.03(L)*****	0.39±0.04(L)***	**0.43±0.05(L)*****	0.20±0.08	0.63±0.04
	RELINK	0.51±0.13	0.46±0.04	0.50±0.12	0.62±0.12	0.32±0.08	0.63±0.00

Notes: (1) *** means $p < 0.001$, ** means $p < 0.01$, * means $p < 0.05$.
(2) L/M/S: Large/Medium/Small effect size according to Cliff's delta.
(3) '↓' indicates 'the smaller the better'; '↑' indicates 'the larger the better'.

TABLE 7: The Distribution of Instance Inspection Effort and Instance Quality on AEEEM

Dataset	Project	Percentage of Defects					# Defect	Percentage of Efforts					# Total Effort	# Instances	Distribution
		10%	20%	30%	40%	50%		10%	20%	30%	40%	50%			Larger effort⟸⟹Lower effort
AEEEM	eclipse	0.35	0.54	0.68	0.76	0.79	206	0.59	0.74	0.83	0.89	0.93	224,055	997	
	equinox	0.21	0.36	0.53	0.63	0.73	129	0.55	0.72	0.83	0.90	0.95	39,534	324	
	lucene	0.20	0.39	0.50	0.58	0.67	64	0.58	0.73	0.82	0.88	0.92	73,184	691	
	mylyn	0.29	0.45	0.58	0.68	0.73	245	0.52	0.68	0.78	0.86	0.91	156,102	1,862	
	pde	0.25	0.46	0.59	0.70	0.78	209	0.42	0.60	0.72	0.81	0.87	146,952	1,497	

TABLE 8: The Distribution of Instance Inspection Effort and Instance Quality on NASA

Dataset	Project	Percentage of Defects					# Defect	Percentage of Efforts					# Total Effort	# Instances	Distribution
		10%	20%	30%	40%	50%		10%	20%	30%	40%	50%			Larger effort⟸⟹Lower effort
NASA	CM1	0.29	0.48	0.60	0.69	0.76	42	0.38	0.55	0.66	0.75	0.81	13,626	344	
	JM1	0.25	0.41	0.55	0.65	0.72	1,759	0.47	0.64	0.75	0.82	0.88	272,370	9,593	
	KC1	0.29	0.48	0.64	0.75	0.86	325	0.50	0.72	0.84	0.91	0.96	30,631	2,096	
	KC3	0.28	0.39	0.53	0.58	0.64	36	0.35	0.54	0.66	0.75	0.82	6,354	200	
	MC1	0.65	0.84	0.88	0.88	1.00	68	0.65	0.84	0.93	0.97	1.00	59,159	9,277	
	MC2	0.23	0.34	0.45	0.57	0.66	44	0.41	0.58	0.70	0.79	0.86	5,205	127	
	MW1	0.44	0.59	0.63	0.74	0.78	27	0.28	0.45	0.59	0.69	0.78	6,838	264	
	PC1	0.30	0.46	0.61	0.75	0.82	61	0.37	0.55	0.66	0.75	0.82	22,038	759	
	PC2	0.13	0.19	0.44	0.44	0.69	16	0.31	0.47	0.59	0.69	0.77	3,170	1,585	
	PC3	0.18	0.37	0.52	0.66	0.74	140	0.41	0.57	0.68	0.76	0.83	31,040	1,125	
	PC4	0.17	0.39	0.57	0.69	0.75	178	0.38	0.56	0.68	0.78	0.84	26,980	1,399	
	PC5	0.91	0.94	0.96	0.96	0.96	503	0.80	0.85	0.88	0.90	0.92	153,501	17,001	

TABLE 9: The Distribution of Instance Inspection Effort and Instance Quality on RELINK

Dataset	Project	Percentage of Defects					# Defect	Percentage of Efforts					# Total Effort	# Instances	Distribution
		10%	20%	30%	40%	50%		10%	20%	30%	40%	50%			Larger effort⟸⟹Lower effort
RELINK	Apache2.0	0.16	0.31	0.48	0.62	0.71	98	0.41	0.63	0.78	0.86	0.92	59,879	194	
	openintents	0.27	0.50	0.59	0.73	0.77	22	0.46	0.69	0.80	0.90	0.95	4,121	56	
	zxing1.6	0.16	0.28	0.43	0.57	0.68	118	0.47	0.68	0.79	0.86	0.91	12,811	399	

TABLE 10: The Distribution of Instance Inspection Effort and Instance Quality on PROMISE

Dataset	Project	Percentage of Defects					# Defect	Percentage of Efforts					# Total Effort	# Instances	Distribution
		10%	20%	30%	40%	50%		10%	20%	30%	40%	50%			Larger effort⟸⟹Lower effort
PROMISE	ant-1.3	0.40	0.55	0.65	0.80	0.90	20	0.36	0.55	0.70	0.79	0.87	37,699	125	
	ant-1.4	0.10	0.25	0.40	0.50	0.63	40	0.38	0.59	0.73	0.83	0.89	54,195	178	
	ant-1.5	0.34	0.59	0.69	0.72	0.81	32	0.45	0.65	0.77	0.85	0.90	87,047	293	
	ant-1.6	0.26	0.51	0.70	0.82	0.88	92	0.44	0.64	0.76	0.84	0.90	113,246	351	
	ant-1.7	0.33	0.57	0.69	0.78	0.87	166	0.45	0.64	0.75	0.84	0.90	208,653	745	
	arc	0.33	0.44	0.52	0.59	0.63	27	0.55	0.70	0.81	0.89	0.94	31,342	234	
	berek	0.31	0.56	0.81	0.94	1.00	16	0.64	0.75	0.84	0.91	0.94	32,320	43	
	camel-1.0	0.31	0.31	0.54	0.69	0.77	13	0.41	0.59	0.71	0.81	0.88	33,721	339	
	camel-1.2	0.14	0.24	0.35	0.45	0.54	216	0.41	0.61	0.75	0.84	0.90	66,302	608	
	camel-1.4	0.23	0.39	0.48	0.61	0.69	145	0.43	0.63	0.76	0.84	0.91	98,080	872	
	camel-1.6	0.19	0.31	0.43	0.51	0.62	188	0.45	0.64	0.76	0.85	0.91	113,055	965	
	ckjm	0.20	0.40	0.40	0.60	0.80	5	0.29	0.43	0.58	0.72	0.82	1,469	10	
	e-learning	0.60	0.60	1.00	1.00	1.00	5	0.42	0.59	0.72	0.80	0.87	3,639	64	
	forrest-0.7	0.20	0.60	0.60	0.60	0.80	5	0.26	0.45	0.61	0.74	0.83	4,930	29	
	ivy-1.1	0.14	0.29	0.44	0.59	0.67	63	0.55	0.72	0.81	0.88	0.93	27,292	111	
	ivy-1.4	0.44	0.56	0.69	0.75	0.88	16	0.58	0.75	0.84	0.90	0.94	59,286	241	
	ivy-2.0	0.45	0.65	0.75	0.83	0.88	40	0.52	0.73	0.83	0.90	0.94	87,769	352	
	jedit-3.2	0.20	0.42	0.53	0.68	0.78	90	0.63	0.75	0.83	0.89	0.93	128,883	272	
	jedit-4.0	0.25	0.48	0.67	0.72	0.75	75	0.62	0.76	0.83	0.89	0.93	144,803	306	
	jedit-4.1	0.28	0.51	0.63	0.73	0.81	79	0.61	0.74	0.83	0.89	0.93	153,087	312	
	jedit-4.2	0.38	0.60	0.73	0.83	0.92	48	0.56	0.70	0.80	0.87	0.92	170,683	367	
	jedit-4.3	0.36	0.55	0.55	0.64	0.64	11	0.54	0.69	0.80	0.88	0.93	202,363	492	
	kalkulator	0.17	0.33	0.33	0.33	0.83	6	0.32	0.50	0.64	0.71	0.82	4,022	27	
	log4j-1.0	0.29	0.47	0.62	0.76	0.79	34	0.38	0.57	0.70	0.80	0.87	21,549	135	
	log4j-1.1	0.22	0.43	0.59	0.73	0.84	37	0.35	0.53	0.68	0.78	0.86	19,938	109	
	log4j-1.2	0.11	0.20	0.30	0.39	0.50	189	0.40	0.58	0.71	0.80	0.88	38,191	205	
	lucene-2.0	0.19	0.34	0.46	0.57	0.67	91	0.47	0.63	0.76	0.84	0.90	50,596	195	
	lucene-2.2	0.13	0.26	0.39	0.47	0.57	144	0.48	0.65	0.77	0.85	0.91	63,571	247	
	lucene-2.4	0.14	0.27	0.38	0.49	0.58	203	0.52	0.69	0.80	0.87	0.92	102,859	340	
	nieruchomosci	0.30	0.60	0.70	0.80	0.90	10	0.35	0.56	0.70	0.78	0.87	4,754	27	
	pbeans1	0.15	0.30	0.40	0.50	0.60	20	0.76	0.87	0.90	0.95	0.97	5,572	26	
	pbeans2	0.30	0.50	0.60	0.60	0.80	10	0.73	0.83	0.90	0.95	0.98	15,125	51	
	pdftranslator	0.27	0.40	0.53	0.67	0.80	15	0.56	0.71	0.81	0.90	0.94	6,318	33	
	poi-1.5	0.12	0.28	0.40	0.53	0.63	141	0.40	0.57	0.70	0.79	0.86	55,428	237	
	poi-2.0	0.30	0.41	0.51	0.62	0.62	37	0.48	0.63	0.75	0.83	0.88	93,171	314	
	poi-2.5	0.13	0.25	0.38	0.47	0.57	248	0.49	0.64	0.75	0.83	0.89	119,731	385	
	poi-3.0	0.14	0.27	0.41	0.53	0.64	281	0.49	0.66	0.77	0.85	0.90	129,327	442	

continued on next page

TABLE 10: The Distribution of Instance Inspection Effort and Instance Quality on PROMISE. – continued from previous page

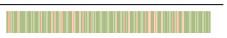
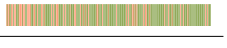
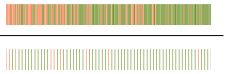
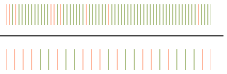
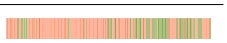
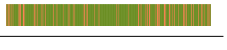
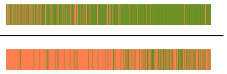
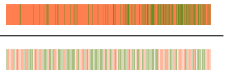
Dataset	Project	Percentage of Defects					# Defect	Percentage of Efforts					# Total Effort	# Instances	Distribution
		10%	20%	30%	40%	50%		10%	20%	30%	40%	50%			Larger effort⟺Lower effort
PROMISE	redaktor	0.11	0.15	0.30	0.44	0.48	27	0.34	0.51	0.64	0.74	0.83	59,280	176	
	serapion	0.33	0.67	0.67	0.67	0.78	9	0.53	0.67	0.78	0.83	0.88	10,505	45	
	skarbonka	0.22	0.33	0.44	0.78	0.89	9	0.43	0.62	0.78	0.87	0.94	15,029	45	
	sklebagd	0.17	0.33	0.50	0.58	0.67	12	0.39	0.54	0.65	0.74	0.83	9,602	20	
	synapse-1.0	0.44	0.69	0.88	0.88	0.88	16	0.32	0.52	0.65	0.75	0.84	28,806	157	
	synapse-1.1	0.22	0.38	0.50	0.60	0.68	60	0.35	0.55	0.69	0.78	0.86	42,302	222	
	synapse-1.2	0.21	0.42	0.57	0.65	0.73	86	0.36	0.58	0.70	0.80	0.87	53,500	256	
	systemdata	0.33	0.33	0.67	0.78	0.89	9	0.55	0.74	0.84	0.90	0.94	15,441	65	
	szybkafucha	0.21	0.29	0.36	0.43	0.57	14	0.35	0.49	0.65	0.74	0.85	1,910	25	
	termoproject	0.38	0.54	0.69	0.69	0.85	13	0.57	0.72	0.82	0.89	0.94	8,239	42	
	tomcat	0.39	0.65	0.75	0.82	0.87	77	0.54	0.74	0.84	0.91	0.95	300,674	858	
	velocity-1.4	0.10	0.18	0.27	0.38	0.47	147	0.62	0.75	0.84	0.90	0.94	51,713	196	
	velocity-1.5	0.11	0.24	0.39	0.52	0.64	142	0.63	0.76	0.84	0.90	0.95	53,141	214	
	velocity-1.6	0.13	0.32	0.47	0.60	0.68	78	0.62	0.76	0.85	0.91	0.95	57,012	229	
	workflow	0.20	0.30	0.45	0.55	0.65	20	0.46	0.61	0.72	0.81	0.87	4,125	39	
	wspomaganiepi	0.17	0.25	0.42	0.58	0.67	12	0.31	0.50	0.64	0.75	0.80	5,685	18	
	xalan-2.4	0.30	0.51	0.71	0.83	0.88	110	0.51	0.69	0.80	0.88	0.93	225,088	723	
	xalan-2.5	0.15	0.28	0.40	0.51	0.62	387	0.54	0.72	0.83	0.90	0.94	304,860	803	
	xalan-2.6	0.21	0.38	0.50	0.61	0.73	411	0.53	0.72	0.83	0.90	0.94	411,737	885	
	xalan-2.7	0.10	0.20	0.30	0.40	0.51	898	0.52	0.71	0.82	0.89	0.94	428,555	909	
	xerces-1.2	0.18	0.28	0.32	0.44	0.46	71	0.73	0.88	0.94	0.97	0.98	159,254	440	
	xerces-1.3	0.26	0.46	0.57	0.75	0.83	69	0.73	0.88	0.94	0.97	0.98	167,095	453	
	xerces-1.4	0.13	0.25	0.37	0.49	0.61	437	0.68	0.84	0.91	0.95	0.97	141,180	588	
	xerces-init	0.16	0.23	0.31	0.39	0.44	77	0.73	0.88	0.94	0.97	0.98	90,718	162	
	zuzel	0.23	0.46	0.62	0.69	0.85	13	0.38	0.65	0.82	0.89	0.94	14,421	29	

TABLE 11: The basic classifiers used in our experiment

Classifier	Abbr.	Brief Description
C4.5 Decision Tree [43]	DT	A tree structure consists of nodes and leafs. Each node of the tree represents a logical decision based on a single metric, while each leaf of the tree defines the classification. Information gain decides which attribute should be used for decision.
Random Forest [44]	RF	A random forest is composed of many random trees. Each random tree is a decision tree (e.g., C4.5).
Logistic Regression [45]	LR	A linear regression model estimates the likelihood of a classification with logistic function. For classification issues, logistic regression chooses the class with the highest likelihood.
Naive Bayes [46]	NB	The method estimates a score for each class based on a simplification of Bayes law.
RBF Network [47]	NET	One of the types of artificial neural network with Radial Basis Functions (RBFs) as neurons.
Support Vector Machine [48]	SVM	It determines a hyperplane that separates the positive from the negative samples. The hyperplane is determined in the kernel space of the data, i.e., a transformation of the data in a higher dimensional space using a kernel function. We use RBFs as kernel function in this paper.

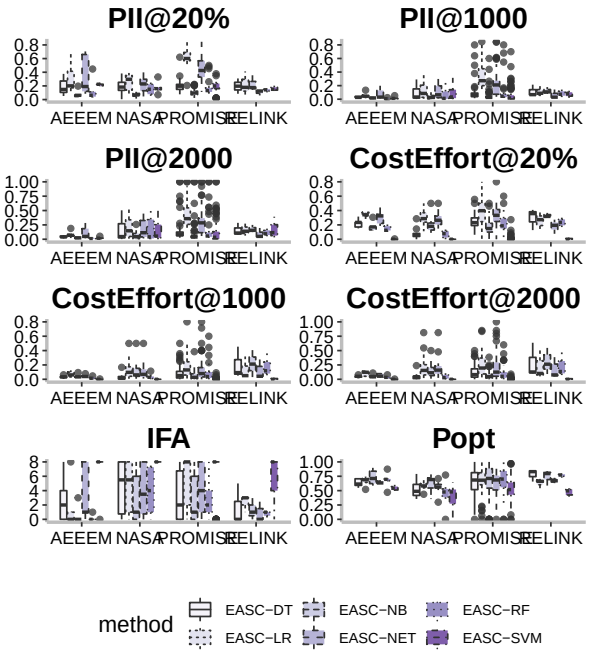


Fig. 3: Comparison of Different Classifiers in EASC in Terms of Effort-Aware Performance Measures.

performance measure. Besides, for every sub-figure, the x-axis represents different datasets while the y-axis represents the corresponding performance values. Notice since the large range of *IFA* value (i.e., **0**∼**17001**), we truncate those values which are large than 22 and make them equal to 22 for better illustration presentation. We choose 22 as the cut-point since the number of values which are larger than 22 occupies a small proportion (i.e., 121/492) and the number 22 is statistically the upper quartile value for *IFA*.

As shown in Figure 2 and Figure 3, when EASC built on different basic classifiers, the performances of EASC appear similarly in terms of different performance measures. Through observing this figure, we can obtain the following two conclusions:

- the six state-of-the-art classifiers have similar prediction ability since they can achieve similar performance in terms of NPMs and EPMs;
- EASC does not rely on the classifiers used, at least on the four datasets. However, through further analysis of the results, we find that EASC using Naive Bayes wins more times than any other classifiers considering the average performances.

We list the total numbers of wins and ties for each classifier on different projects in Table 12. As shown in Table 12, the first column represents classifiers, the following 11 columns represent different performance measures used in this paper and the last column lists the sum of each classifier. We sum up the number of cases that EASC built on a special classifier obtains the best performance or obtains the same performance with others which obtain the best performance. At last, we sort classifiers in descending order according to total win/tied numbers and find that NB ranks first. Therefore, we use NB as the default basic classifier in EASC.

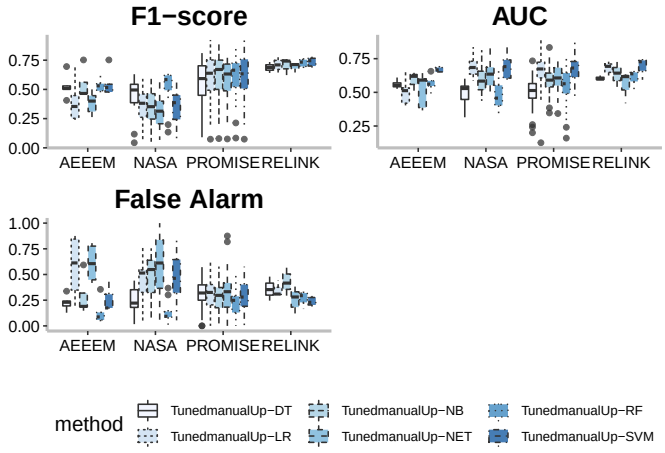


Fig. 4: Comparison of Different Classifiers in $TunedmanualUp$ in Terms of Non-effort-aware Performance Measures.

> EASC is insensitive to basic classifiers and can obtain stable performances on various projects using different classifiers.

F.2 The Influence of Classifier on TunedmanualUp

Menzies et al. [53] found that manualUp tuned with a defect predictor could achieve better performance. In particular, in the phase of model building, a defect predictor should be trained on training instances. In the phase of model applying, the defect predictor firstly makes a binary decision (e.g., defective or clean) on testing instances. Then, all instances identified as defective are sorted in ascending order by LOC. Therefore, the choice of $TunedmanualUp$ underlying classifier can infect the sorting order of testing instances.

Therefore, in this section, to investigate the effect of the choice of $TunuedmanualUp$'s underlying classifier, we make a comparison among six commonly used classifiers: Decision Tree, Random Forest, Logistic Regression, Naive Bayes, RBF Network and Support Vector Machine. The introduction to all basic classifiers can be found in Table 11.

Figure 4 and Figure 5 present the overall performance of $TunedmanualUp$ in terms of 11 performance measures when using six different basic classifiers. Notice since the large range of *IFA* value (i.e., 0∼435), we truncate those values which are large than 8 and make them equal to 8 for better illustration presentation. We choose 8 as the cut-point since the number of values which are larger than 8 occupies a small proportion (i.e., 109/492) and the number 8 is statistically the upper quartile value for *IFA*.

As shown in Figure 4 and Figure 5, when $TunedmanualUp$ built on different basic classifiers, $TunedmanualUp$ achieves similar performances in terms of different performance measures. Through observing this figure, we can find that the six state-of-the-art classifiers

TABLE 12: The Win/Tie Numbers of Different Classifiers in EASC on Four Datasets

Classifiers	F1-score	AUC	False Alarm	IFA	PII			CostEffort			Popt	Total
					20%	1000	2000	20%	1000	2000		
EASC-NB	18	32	16	41	56	27	33	10	11	8	30	282
EASC-SVM	1	2	82	14	7	44	39	0	1	1	5	196
EASC-LR	14	0	0	16	0	0	2	40	52	53	14	191
EASC-RF	8	37	7	29	15	13	13	10	9	9	12	162
EASC-NET	24	4	0	21	4	2	5	28	30	31	9	158
EASC-DT	18	8	5	32	3	1	4	11	13	14	16	125

TABLE 13: The Win/Tie Numbers of Different Classifiers in $TunedmaunalUp$ on Four Datasets

Classifiers	F1-score	AUC	False Alarm	IFA	PII			CostEffort			Popt	Total
					20%	1000	2000	20%	1000	2000		
TunedmanualUp-NB	37	4	58	27	60	28	35	15	28	22	32	346
TunedmanualUp-SVM	18	42	10	52	14	29	28	20	22	19	15	269
TunedmanualUp-LR	15	21	6	40	3	15	10	20	31	28	16	205
TunedmanualUp-RF	13	6	10	38	17	29	27	14	24	14	11	203
TunedmanualUp-NET	8	8	11	29	12	20	12	14	21	18	9	162
TunedmanualUp-DT	8	3	8	19	3	1	3	22	30	30	18	145

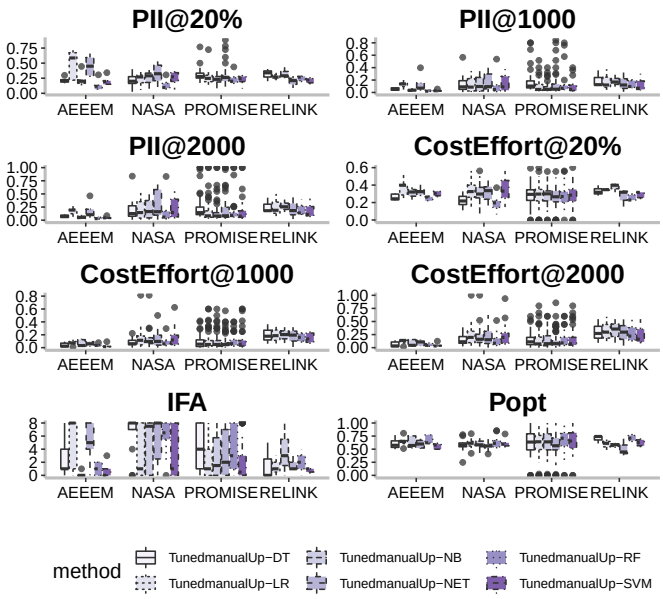


Fig. 5: Comparison of Different Classifiers in $TunedmanualUp$ in Terms of Effort-Aware Performance Measures.

have similar prediction ability since they can achieve similar performance in terms of NPMs and EPMs.

We list the total numbers of wins and ties for each classifier on different projects in Table 13. As shown in Table 13, we find that $TunedmanualUp$ using Naive Bayes wins more times than any other classifiers considering the average performances. Therefore, we use NB as the default basic classifier in $TunedmanualUp$.

APPENDIX G
THE CORRELATION BETWEEN PERFORMANCE MEASURES.

In this paper, we totally consider 11 performance measures, which can be divided into two groups: non-effort-aware performance measures (NPMs) and effort-aware performance measures (EPMs). In particular, as for NPMs, three performance measures are considered which are widely used in the scenario of software defect prediction. As for EPMs, eight performance measures from four different types are considered which are most recently proposed and are not considered in Zhou et al.'s work.

Different software modules may have different sizes. That is, the lines of code in software modules may vary from hundreds of LOC to thousands of LOC. Therefore, to comprehensively investigate $PII@L$ and $CostEffort@L$, two kinds of $PII@L$ and $CostEffort@L$ are considered: 1) relative LOC of PII and CostEffort (e.g., 20%); 2) absolute LOC of PII and CostEffort (e.g., 1000, 2000). These performance measures consider costs from different perspectives: inspection costs, revenue costs, and impact on developers.

We make a correlation analysis among the all performance measures on all projects and all methods. The results are shown in Figure 6:

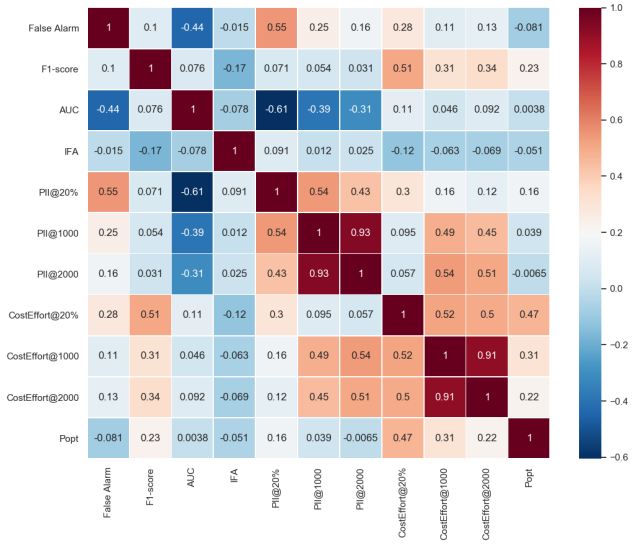


Fig. 6: Correlation analysis among all performance measures on all projects and all methods.

In the above figure, different colors indicate different degrees of correlation between two performance measures.

From this figure, we make the following observations: (1) $PII@1000$ ($CosfEffort@1000$) has a high correlation with $PII@2000$ ($CosfEffot@2000$), which is also obvious since they has same definition with different inspection effort. (2) $F1\text{-}score$ has a 51% correlation with $CostEffort@20\%$. The reason behinds this observation is that $CostEffort@L$ has the same meaning with $Recall$ and $CostEffort@L$ can be treated as the effort-aware version of $Recall$. Besides, $F1\text{-}score$ is highly related to $Recall$. (3) $PII@20\%$ has a 55% correlation with $False\ Alarm$. It is obvious that if there are many false alarm in modeling application, then it requires developers to switch the context more frequently. Therefore, $False\ Alarm$ has a relationship with $PII@20\%$. (4) For other performance measures, there is not obvious correlation between two performance measures.

According to above analysis, in this revision, we keep $PII@1000$ and $PII@2000$ since they are the absolute version of $PII@L$. We also keep $CostEffort@L$ for the same reason. Besides, we also keep $F1\text{-}score$, $CostEffort@20\%$, $PII@20\%$ and $False\ Alarm$ for two reasons: 1) the correlation between each pair of performance measures is not large; 2) these performance measures are proposed for different goals (i.e., effort-aware and non-effort-aware).

REFERENCES

[1] S. Herbold, A. Trautsch, and J. Grabowski, "A comparative study to benchmark cross-project defect prediction approaches," IEEE Trans. Software Eng., vol. 44, no. 9, pp. 811–833, 2018.

[2] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "Hydra: Massively compositional model for cross-project defect prediction," IEEE Transactions on Software Engineering, vol. 42, no. 10, pp. 977–998, 2016.

[3] D. Ryu, O. Choi, and J. Baik, "Value-cognitive boosting with a support vector machine for cross-project defect prediction," Empirical Software Engineering, vol. 21, no. 1, pp. 43–71, 2016.

[4] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in Proceedings of the 10th Working Conference on Mining Software Repositories, 2013, pp. 409–418.

[5] D. Ryu and J. Baik, "Effective multi-objective naïve bayes learning for cross-project defect prediction," Applied Soft Computing, vol. 49, pp. 1062–1077, 2016.

[6] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," Information and Software Technology, vol. 54, no. 3, pp. 248–256, 2012.

[7] X. Chen, D. Zhang, Y. Zhao, Z. Cui, and C. Ni, "Software defect number prediction: Unsupervised vs supervised methods," Information and Software Technology, vol. 106, pp. 161–181, 2019.

[8] Z. Xu, S. Li, Y. Tang, X. Luo, T. Zhang, J. Liu, and J. Xu, "Cross version defect prediction with representative data via sparse subset selection," in Proceedings of the 26th Conference on Program Comprehension, ser. ICPC '18. New York, NY, USA: ACM, 2018, pp. 132–143. [Online]. Available: http://doi.acm.org/10.1145/3196321.3196331

[9] C. Liu, D. Yang, X. Xia, M. Yan, and X. Zhang, "A two-phase transfer learning model for cross-project defect prediction," Information and Software Technology, 2018.

[10] Y. Zhang, D. Lo, X. Xia, and J. Sun, "Combined classifier for cross-project defect prediction: an extended empirical study," Frontiers of Computer Science, vol. 12, no. 2, pp. 280–296, 2018.

[11] Z. Xu, J. Liu, X. Luo, and T. Zhang, "Cross-version defect prediction via hybrid active learning with kernel principal component analysis," in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018, pp. 209–220.

[12] Y. Liu, Y. Li, J. Guo, Y. Zhou, and B. Xu, "Connecting software metrics across versions to predict defects," in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018, pp. 232–243.

[13] S. Hosseini, B. Turhan, and M. Mäntylä, "A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction," Information and Software Technology, vol. 95, pp. 296–312, 2018.

[14] A. Boucher and M. Badri, "Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison," Information and Software Technology, vol. 96, pp. 38–67, 2018.

[15] B. Turhan, A. T. Mısırlı, and A. Bener, "Empirical evaluation of the effects of mixed project data on learning defect predictors," Information and Software Technology, vol. 55, no. 6, pp. 1101–1118, 2013.

[16] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, "An empirical study of just-in-time defect prediction using cross-project models," in Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, 2014, pp. 172–181.

[17] C. Ni, W. Liu, Q. Gu, X. Chen, and D. Chen, "Fesch: A feature selection method using clusters of hybrid-data for cross-project defect prediction," in Proceedings of the 41st Annual Computer Software and Applications Conference (COMPSAC). IEEE, 2017, pp. 51–56.

[18] S. Amasaki, "Cross-version defect prediction using cross-project defect prediction approaches: Does it work?" in Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering. ACM, 2018, pp. 32–41.

[19] N. Limsettho, K. E. Bennin, J. W. Keung, H. Hata, and K. Matsumoto, "Cross project defect prediction using class distribution estimation and oversampling," Information and Software Technology, vol. 100, pp. 87–102, 2018.

[20] Z. Li, X.-Y. Jing, X. Zhu, and H. Zhang, "Heterogeneous defect prediction through multiple kernel learning and ensemble learning," in Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2017, pp. 91–102.

[21] D. Ryu, J.-I. Jang, and J. Baik, "A transfer cost-sensitive boosting approach for cross-project defect prediction," Software Quality Journal, vol. 25, no. 1, pp. 235–272, 2017.

[22] Z. Li, X.-Y. Jing, F. Wu, X. Zhu, B. Xu, and S. Ying, "Cost-sensitive transfer kernel canonical correlation analysis for heterogeneous defect prediction," Automated Software Engineering, vol. 25, no. 2, pp. 201–245, 2018.

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

[24] P. S. Kochhar, X. Xia, D. Lo, and 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.

[25] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in Proceedings of the international symposium on software testing and analysis. ACM, 2011, pp. 199–209.

[26] J. H. Feng Wang and Y. Ma, "A top-k learning to rank approach to cross-project software defect prediction," in Proceedings of 25th Asia-Pacific Software Engineering Conference, 2018.

[27] X. Yang and W. Wen, "Ridge and lasso regression models for cross-version defect prediction," IEEE Transactions on Reliability, vol. 67, no. 3, pp. 885–896, 2018.

[28] W. N. Poon, K. E. Bennin, J. Huang, P. Phannachitta, and J. W. Keung, "Cross-project defect prediction using a credibility theory based naive bayes classifier," in Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on. IEEE, 2017, pp. 434–441.

[29] Z. Li, X.-Y. Jing, X. Zhu, H. Zhang, B. Xu, and S. Ying, "On the multiple sources and privacy preservation issues for heterogeneous defect prediction," IEEE Transactions on Software Engineering (TSE), 2017.

[30] Y. Fan, C. Lv, X. Zhang, G. Zhou, and Y. Zhou, "The utility challenge of privacy-preserving data-sharing in cross-company defect prediction: An empirical study of the cliff&morph algorithm," in 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2017, pp. 80–90.

[31] Y. Zhou, Y. Yang, H. Lu, L. Chen, Y. Li, Y. Zhao, J. Qian, and B. Xu, "How far we have progressed in the journey? an examination of cross-project defect prediction," ACM Trans. Softw. Eng. Methodol., vol. 27, no. 1, pp. 1:1–1:51, 2018.

[32] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the nasa software defect datasets," IEEE

Transactions on Software Engineering, vol. 39, no. 9, pp. 1208–1215, 2013.
[33] D. Gray, D. Bowes, N. Davey, and Y. Sun, "The misuse of the nasa metrics data program data sets for automated software defect prediction," in Proceedings of Evaluation & Assessment in Software Engineering, 2011, pp. 96–103.
[34] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in Proceedings of the 7th Mining Software Repositories (MSR). IEEE, 2010, pp. 31–41.
[35] R. Wu, H. Zhang, S. Kim, and 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.
[36] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in Proceeding of International Conference on Predictive MODELS in Software Engineering, 2010, pp. 1–10.
[37] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan, "The promise repository of empirical software engineering data," 2012.
[38] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," IEEE Transactions on Software Engineering, vol. 45, pp. 683–711, 2019.
[39] A. E. Camargo Cruz and K. Ochimizu, "Towards logistic regression models for predicting fault-prone code across software projects," in Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE Computer Society, 2009, pp. 460–463.
[40] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," Empirical Software Engineering, vol. 14, no. 5, pp. 540–578, 2009.
[41] T. Menzies, A. Butcher, A. Marcus, and D. Zimmermann, Thomas a nd Cok, "Local vs. global models for effort estimation and defect prediction," in Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, 2011, pp. 343–351.
[42] S. Watanabe, H. Kaiya, and K. Kaijiri, "Adapting a fault prediction model to allow inter languagereuse," in Proceedings of the 4th international workshop on Predictor models in software engineering. ACM, 2008, pp. 19–24.
[43] J. R. Quinlan, C4.5: programs for machine learning. Elsevier, 2014.
[44] L. Breiman, "Random forests," Machine learning, vol. 45, no. 1, pp. 5–32, 2001.
[45] D. R. Cox, "The regression analysis of binary sequences," Journal of the Royal Statistical Society. Series B (Methodological), pp. 215–242, 1958.
[46] S. J. Russell and P. Norvig, Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited,, 2016.
[47] D. S. Broomhead and D. Lowe, "Radial basis functions, multivariable functional interpolation and adaptive networks," Royal Signals and Radar Establishment Malvern (United Kingdom), Tech. Rep., 1988.
[48] M. A. Hearst, S. T. Dumais, E. Osuna, J. Platt, and B. Scholkopf, "Support vector machines," IEEE Intelligent Systems and their applications, vol. 13, no. 4, pp. 18–28, 1998.
[49] X. Xia, D. Lo, X. Wang, and X. Yang, "Collective personalized change classification with multiobjective search," IEEE Transactions on Reliability, vol. 65, no. 4, pp. 1810–1829, 2016.
[50] C. Ni, W.-S. Liu, X. Chen, Q. Gu, D.-X. Chen, and Q.-G. Huang, "A cluster based feature selection method for cross-project software defect prediction," Journal of Computer Science and Technology, vol. 32, no. 6, pp. 1090–1107, 2017.
[51] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," in Proceedings of the 23rd international conference on Machine learning. ACM, 2006, pp. 161–168.
[52] T. Van Gestel, J. A. Suykens, B. Baesens, S. Viaene, J. Vanthienen, G. Dedene, B. De Moor, and J. Vandewalle, "Benchmarking least squares support vector machine classifiers," Machine learning, vol. 54, no. 1, pp. 5–32, 2004.
[53] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," Automated Software Engineering, vol. 17, no. 4, pp. 375–407, 2010.