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

Chao Ni, Xin Xia, David Lo, Xiang Chen, *Member, IEEE*, 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], [2], [3], [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], [13], [14], [15], [16], [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

- *Chao Ni is with the School of Software Technology, Zhejiang University, Ningbo, Zhejiang 315048, China, and with the Ningbo Research Institute, Zhejiang University, Ningbo, Zhejiang, China, and also with the PengCheng Laboratory, Shenzhen, Guangdong 518066, China. E-mail: jacknichao920209@gmail.com.*
- *Xin Xia is with the Faculty of Information Technology, Monash University, Melbourne, VIC 3000, Australia. E-mail: xin.xia@monash.edu.*
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore188065. E-mail: davidlo@smu.edu.sg.*
- *Xiang Chen is with the School of Information Science and Technology Science, Nantong University, Nantong, Jiangsu 226000, China, and also with the Nanjing University, Nanjing, Jiangsu 210094, China. E-mail: xchencs@ntu.edu.cn.*
- *Qing Gu is with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu 210023, China. E-mail: guq@nju.edu.cn.*

Manuscript received 24 Sept. 2019; revised 8 June 2020; accepted 9 June 2020. Date of publication 11 June 2020; date of current version 15 Mar. 2022. (Corresponding author: Xin Xia.)
Recommended for acceptance by Y. Brun.
Digital Object Identifier no. 10.1109/TSE.2020.3001739

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 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], [22], [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], [34], [35]) and *(2) effort-aware performance measures* (i.e., IFA, $PII@L$, $CostEffort@L$ and P_{opt} [27], [36], [37], [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 versus 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.

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

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], [43], [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, $mixed\text{-}project$ defect prediction, and $pair\text{-}wise$ 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].
- *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], [47], [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 calculating 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], [31], [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], [34], [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:

$F1\text{-}score$: 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:

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

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

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

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:

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

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

$$CostEffort@2000 = \frac{n}{N},\ where\ L\ equals\ to\ 2000\ LOC. \tag{6}$$

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 Fig. 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 percent of the LOCs are inspected.

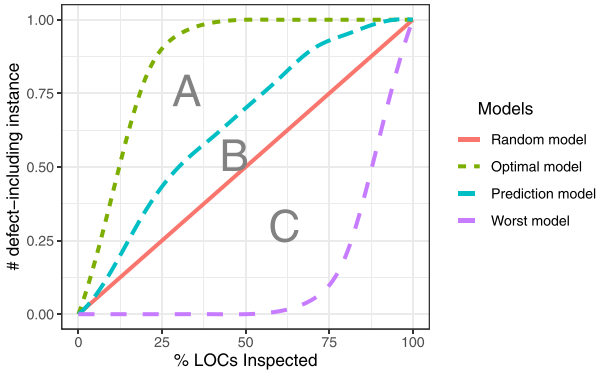


Fig. 1. An example of the relationship between the number of defective instances and the inspection cost for different prediction models.

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 first 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*, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2020.3001739, [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], [22], [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*, available online, [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.

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*, available online, [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], [47], [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?

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

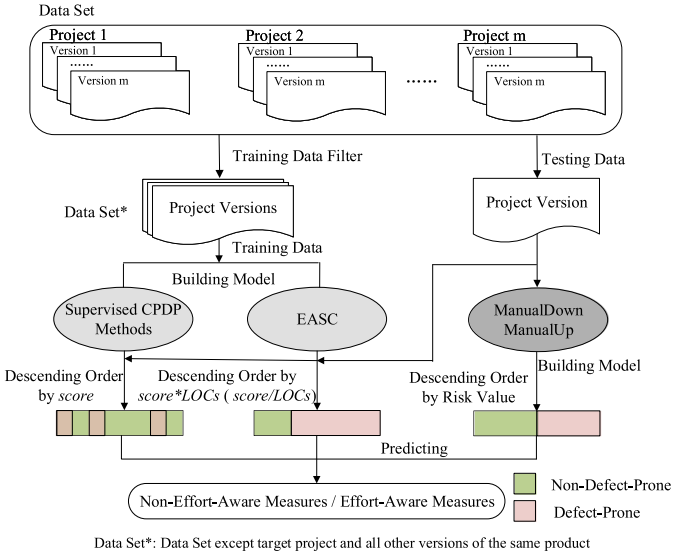


Fig. 2. The workflow of supervised methods and unsupervised methods in CPDP scenario.

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 percent, 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 Fig. 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*, available online, [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, Tables 1 and 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 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*, available online, [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 $F1$-$score$ 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.

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

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'.

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

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

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 percent 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?

First, 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 percent as the classification threshold. We list the average results of five different thresholds (i.e., 10, 20, 30, 40 and 50 percent). 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 percent), 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 percent instances, the majority of defective instances (i.e., at least *more than 70 percent*) will be ranked at the top. As for ManualDown method, the classification threshold is set as 50 percent, which means the top 50 percent 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 percent, ManualDown will obtain a higher $Recall$ (i.e., at least 70 percent 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 percent, then the majority of non-defective instances are ranked in the rest 50 percent. ManualDown classifies the instances in the top 50 percent and the instances in the last 50 percent as defect-prone and non-defect-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 percent instances, the total inspection effort accounts for the majority of all inspection effort (i.e., *at least 86 percent 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*, available online, [58].

Second, 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 percent instances will consume a few of the total inspection effort. For example, inspecting the top 50 percent instances of AEEEM, NASA, PROMISE and RELINK needs to consume only 8, 14, 10 and 7 percent of the total inspection effort, respectively. In other words, inspecting instances with 20 percent of the total inspection effort will

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

inspect much more than 50 percent 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.

When considering NPMs, the unsupervised CPDP method ManualDown performs significantly better than supervised methods on most performance measures (i.e., *F1-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], [47], [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*, available online, [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 first 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*, available online, [58].

Results 1: Comparison between EASC and ManualDown/ ManualUp.

Tables 5 and 6 present the average results and the statistical test results comparing EASC and ManualDown (ManualUp). In Tables 5 and 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 Tables 5 and 6.

Non-effort-aware Performance Comparison. From the results shown in Table 5, in terms of *F1-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 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.

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'.

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

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'.

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

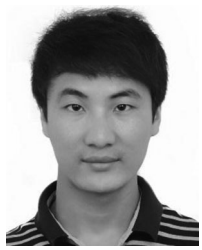
ACKNOWLEDGMENTS

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 *Proc. 23rd Softw. Anal. Evol. Reengineering*, 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 Trans. Softw. Eng.*, early access, 2018, doi: 10.1109/TSE.2018.2877678.
[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," *J. Comput. Sci. Technol.*, vol. 32, no. 6, pp. 1090–1107, 2017.
[4] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," in *Proc. 11th Joint Meeting Foundations Softw. Eng.*, 2017, pp. 72–83.
[5] Y. Zhou *et al.*, "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 Trans. Softw. Eng.*, vol. 42, no. 10, pp. 977–998, Oct. 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 *Proc. 26th Int. Conf. Softw. Anal. Evol. Reengineering*, 2019, pp. 298–309.
[8] X. Yang, K. Tang, and X. Yao, "A learning-to-rank approach to software defect prediction," *IEEE Trans. Rel.*, vol. 64, no. 1, pp. 234–246, Mar. 2015.
[9] S. Kim, E. J. Whitehead Jr, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, Mar. 2008.
[10] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan 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 Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Nov. 2012.
[12] A. Boucher and M. Badri, "Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison," *Inf. Softw. Technol.*, vol. 96, pp. 38–67, 2018.
[13] F. Wu *et al.*, "Cross-project and within-project semisupervised software defect prediction: A unified approach," *IEEE Trans. Rel.*, vol. 67, no. 2, pp. 581–597, Jun. 2018.
[14] R. Krishna and T. Menzies, "Bellwethers: A baseline method for transfer learning," *IEEE Trans. Softw. Eng.*, vol. 45, no. 11, pp. 1081–1105, Nov. 2019.
[15] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," *IEEE Trans. Softw. Eng.*, vol. 45, no. 2, pp. 111–147, Feb. 2019.
[16] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 874–896, Sep. 2018.
[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 Trans. Softw. Eng.*, vol. 43, no. 4, pp. 321–339, Apr. 2017.
[18] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 382–391.
[19] J. Nam and S. Kim, "Clami: Defect prediction on unlabeled datasets," in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2015, pp. 452–463.
[20] C. Tantithamthavorn, "Towards a better understanding of the impact of experimental components on defect prediction modelling," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 867–870.
[21] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in *Proc. 10th Working Conf. Mining Softw. Repositories*, 2013, pp. 409–418.
[22] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Inf. Softw. Technol.*, 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," *Empir. Softw. Eng.*, 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 *Proc. 37th EUROMICRO Conf. Softw. Eng. Advanced Appl.*, 2011, pp. 396–403.
[25] D. Ryu and J. Baik, "Effective multi-objective naïve bayes learning for cross-project defect prediction," *Appl. Soft Comput.*, 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 *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 159–170.
[27] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empir. Softw. Eng.*, vol. 24, pp. 2823–2862, 2019.
[28] S. Herbold, A. Trautsch, and J. Grabowski, "A comparative study to benchmark cross-project defect prediction approaches," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 811–833, Sep. 2018.
[29] S. Herbold, "Benchmarking cross-project defect prediction approaches with costs metrics," pp. 1–12, 2018. [Online]. Available: https://arxiv.org/abs/1801.04107
[30] J. Stuckman, J. Walden, and R. Scandariato, "The effect of dimensionality reduction on software vulnerability prediction models," *IEEE Trans. Rel.*, vol. 66, no. 1, pp. 17–37, Mar. 2017.
[31] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," *Autom. Softw. Eng.*, vol. 19, no. 2, pp. 167–199, 2012.
[32] S. Herbold, A. Trautsch, and J. Grabowski, "Global versus local models for cross-project defect prediction," *Empir. Softw. Eng.*, 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," *Empir. Softw. Eng.*, 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 *Proc. 4th Int. Workshop Predictor Models Softw. Eng.*, 2008, pp. 47–54.
[35] W. Fu, T. Menzies, and X. Shen, "Tuning for software analytics: Is it really necessary?" *Inf. Softw. Technol.*, vol. 76, pp. 135–146, 2016.
[36] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," in *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2014, pp. 19–29.
[37] Y. Yang *et al.*, "Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models," in *Proc. 24th ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2016, pp. 157–168.
[38] Y. Kamei *et al.*, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, Jun. 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 *Proc. 40th Int. Conf. Softw. Eng.*, 2018, Art. no. 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 *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 1050–1061.
[41] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. 31st Int. Conf. Softw. Eng.*, 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 *Proc. 41st Annu. Comput. Softw. Appl. Conf.*, 2017, pp. 51–56.
[43] X. Chen, D. Zhang, Y. Zhao, Z. Cui, and C. Ni, "Software defect number prediction: Unsupervised vs supervised methods," *Inf. Softw. Technol.*, vol. 106, pp. 161–181, 2019.
[44] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the imprecision of cross-project defect prediction," in *Proc. ACM SIGSOFT 20th Int. Symp. Foundations Softw. Eng.*, 2012, Art. no. 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," *Autom. Softw. Eng.*, 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," *Empir. Softw. Eng.*, vol. 13, no. 5, 2008, Art. no. 473.
[47] G. Koru, H. Liu, D. Zhang, and K. El Emam, "Testing the theory of relative defect proneness for closed-source software," *Empir. Softw. Eng.*, 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 Trans. Softw. Eng.*, vol. 35, no. 2, pp. 293–304, Mar./Apr. 2009.
[49] Y. Jiang, B. Cukic, and T. Menzies, "Fault prediction using early lifecycle data," in *Proc. 18th IEEE Int. Symp. Softw. Rel.*, 2007, pp. 237–246.
[50] C. Tantithamthavorn and A. E. Hassan, "An experience report on defect modelling in practice: Pitfalls and challenges," in *Proc. 40th Int. Conf. Softw. Eng.: Softw. Eng. Practice*, 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 *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 199–209.
[53] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 165–176.
[54] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Proc. 14th Eur. Conf. Softw. Maintenance Reengineering*, 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," *J. Syst. Softw.*, vol. 83, no. 1, pp. 2–17, 2010.
[56] Y. Yang *et al.*, "An empirical study on dependence clusters for effort-aware fault-proneness prediction," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 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 *Proc. IEEE Int. Conf. Softw. Maintenance*, 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 *Proc. 7th Mining Softw. Repositories*, 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 Trans. Softw. Eng.*, vol. 39, no. 9, pp. 1208–1215, Sep. 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 *Proc. Eval. Assessment Softw. Eng.*, 2011, pp. 96–103.
[62] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proc. Int. Conf. Predictive MODELS Softw. Eng.*, 2010, pp. 1–10.
[63] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Foundations Softw. Eng.*, 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 *Proc. 3rd Int. Symp. Empir. Softw. Eng. Meas.*, 2009, pp. 460–463.
[65] T. Menzies, A. Butcher, A. Marcus, and D. Zimmermann, Thomas a nd Cok, "Local versus global models for effort estimation and defect prediction," in *Proc. 26th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2011, pp. 343–351.
[66] S. Watanabe, H. Kaiya, and K. Kaijiri, "Adapting a fault prediction model to allow inter languagereuse," in *Proc. 4th Int. Workshop Predictor Models Softw. Eng.*, 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*. London, U.K. : Psychology Press, 2014.
[69] S. Amasaki, "Cross-version defect prediction using cross-project defect prediction approaches: Does it work?" in *Proc. 14th Int. Conf. Predictive Models Data Analytics Softw. Eng.*, 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. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Jul./Aug. 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 Newslett.*, 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 versus domain versus process," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Foundations Softw. Eng.*, 2009, pp. 91–100.
[73] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 297–308.
[74] S. J. Pan, I. W. Tsang, J. T. Kwok, and Q. Yang, "Domain adaptation via transfer component analysis," *IEEE Trans. Neural Netw.*, vol. 22, no. 2, pp. 199–210, Feb. 2011.
[75] Y. Zhang, D. Lo, X. Xia, and J. Sun, "Combined classifier for cross-project defect prediction: An extended empirical study," *Frontiers Comput. Sci.*, 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 *Proc. 10th Joint Meeting Foundations Softw. Eng.*, 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 *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 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 Trans. Softw. Eng.*, vol. 45, no. 4, pp. 391–411, Apr. 2019.
[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," *Autom. Softw. Eng.*, 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 *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 309–320.



Chao Ni received the BSc degree in computer science from Nantong University, Nantong, in 2014, and the MSc and PhD degrees in computer software and theory from Nanjing University, China, in 2017 and 2020, respectively. His research interest includes software engineering. In particular, he is interested in mining software repositories, empirical software engineering, software maintenance, and software defect prediction.

Xin Xia received the PhD degree in computer science from Zhejiang University, in 2014. He is an ARC DECRA fellow and a lecturer with the Faculty of Information Technology, Monash University, Australia. Prior to joining Monash University, he was a postdoctoral research fellow with the software practices lab at the University of British Columbia in Canada, and a research assistant professor at Zhejiang University in China. To help developers and testers improve their productivity, his current research focuses on mining and analyzing rich data in software repositories to uncover interesting and actionable information. For more information, please visit https://xin-xia.github.io/

David Lo received the PhD degree in computer science from the National University of Singapore, in 2008. He is a ACM distinguished member and an associate professor of information systems at Singapore Management University. His research interests include intersection of software engineering and data science, encompassing socio-technical aspects and analysis of different kinds of software artefacts, with the goal of improving software quality and developer productivity. His work has been published in premier and major conferences and journals in the area of software engineering, AI, and cybersecurity.

Xiang Chen (Member, IEEE) received the BSc degree from the School of Management, Xi'an Jiaotong University, China in 2002, and the MSc and PhD degrees in computer software and theory from Nanjing University, China, in 2008 and 2011, respectively. He is with the Department of Information Science and Technology, Nantong University as an associate professor. His research interests are mainly in software engineering. In particular, he is interested in empirical software engineering, mining software repositories, software maintenance, and software testing. In these areas, he has published more than 60 papers in refereed journals or conferences, such as the *IEEE Transactions on Software Engineering, Information and Software Technology*, the *Journal of Systems and Software*, the *IEEE Transactions on Reliability*, the *Journal of Software: Evolution and Process*, the *Software Quality Journal*, the *Journal of Computer Science and Technology*, ASE, ICSME, SANER, and COMPSAC. He is a senior member of CCF, China, and a member of ACM. For more information, please visit https://smartse.github.io/index.html

Qing Gu received the PhD degree in computer science from Nanjing University, Nanjing. He is a professor with the State Key Laboratory of Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing. His research interests include software testing, quality and process improvement, software maintenance and evolution, and complex network.