12-2016

Collective personalized change classification with multiobjective search

Xin XIA

David LO
Singapore Management University, davidlo@smu.edu.sg

Xinyu WANG

Xiaohu YANG

DOI: https://doi.org/10.1109/TR.2016.2588139

Citation

XIA, Xin; David LO; WANG, Xinyu; and YANG, Xiaohu. Collective personalized change classification with multiobjective search. (2016). *IEEE Transactions on Reliability*. 65, (4), 1810-1829. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/3610

Published in IEEE Transactions on Reliability, December 2016, Volume 65, Issue 4, Pages 1810-1829.
http://doi.org/10.1109/TR.2016.2588139

Collective Personalized Change Classification With Multiobjective Search

Xin Xia, *Member, IEEE*, David Lo, *Member, IEEE*, Xinyu Wang, *Member, IEEE*, and Xiaohu Yang

Abstract—**Many change classification techniques have been proposed to identify defect-prone changes. These techniques consider all developers' historical change data to build a global prediction model. In practice, since developers have their own coding preferences and behavioral patterns, which causes different defect patterns, a separate change classification model for each developer can help to improve performance. Jiang, Tan, and Kim refer to this problem as *personalized change classification*, and they propose PCC+ to solve this problem. A software project has a number of developers; for a developer, building a prediction model not only based on his/her change data, but also on other *relevant* developers' change data can further improve the performance of change classification. In this paper, we propose a more accurate technique named *collective personalized change classification* (CPCC), which leverages a multiobjective genetic algorithm. For a project, CPCC first builds a personalized prediction model for each developer based on his/her historical data. Next, for each developer, CPCC combines these models by assigning different weights to these models with the purpose of maximizing two objective functions (i.e., F1-scores and cost effectiveness). To further improve the prediction accuracy, we propose CPCC+ by combining CPCC with PCC proposed by Jiang, Tan, and Kim To evaluate the benefits of CPCC+ and CPCC, we perform experiments on six large software projects from different communities: Eclipse JDT, Jackrabbit, Linux kernel, Lucene, PostgreSQL, and Xorg. The experiment results show that CPCC+ can discover up to 245 more bugs than PCC+ (468 versus 223 for PostgreSQL) if developers inspect the top 20% lines of code that are predicted buggy. In addition, CPCC+ can achieve F1-scores of 0.60–0.75, which are statistically significantly higher than those of PCC+ on all of the six projects.**

Index Terms—**Cost effectiveness, developer, machine learning, multiobjective genetic algorithm, personalized change classification (PCC).**

Acronyms and Abbreviations

CC	Change classification.
PCC	Personalized change classification.
CPCC	Collective personalized change classification.
GA	Genetic algorithm.
LOC	Lines of code.

Manuscript received September 1, 2014; revised May 18, 2015, November 20, 2015, and April 18, 2016; accepted July 2, 2016. Date of publication July 21, 2016; date of current version November 29, 2016. This work was supported by the National Basic Research Program of China (the 973 Program) under Grant 2015CB352201, the National Natural Science Foundation of China Program (No.61572426), and National Key Technology R&D Program of the Ministry of Science and Technology of China under Grant 2015BAH17F01. Associate Editor: W. E. Wong. *(Corresponding author: Xinyu Wang.)*

X. Xia, X. Wang, and X. Yang are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China (e-mail: xxia@zju.edu.cn; wangxinyu@zju.edu.cn; yangxh@zju.edu.cn).

D. Lo is with the School of Information Systems, Singapore Management University, Singapore 188065 (e-mail: davidlo@smu.edu.sg).

Notation

Dev_i	ith source developer.
S_i	Prediction model built on the historical change data of Dev_i.
T	Prediction model built on target developers historical data.
$\mathrm{CPCC}(j)$	The composite confidence score of CPCC for instance j to be buggy.
$\mathrm{CPCC}+(j)$	The composite confidence score of CPCC+ for instance j to be buggy.
threshold	Boundary used to decide whether an instance is buggy or not.

I. Introduction

Change classification (CC) aims to precisely identify the existence of bugs in an individual file-level software change to help in allocating limited test resources. Kim, Whitehead, and Zhang define a *change* as a set of line ranges that are added/deleted/modified in one file in a software version control system commit [1]. A *change* could be *clean* if it contains no bug, and *buggy* if it contains one or more bugs. A number of CC methods based on machine learning techniques have been proposed to build a prediction model from historical change data stored in software repositories [1]–[6]. These methods have achieved significant advances in CC by proposing various features that can be used to better detect if a change is buggy or not. These traditional CC techniques combine *all* developers' change data to build a global model.

Different developers have their own coding preferences and behavioral patterns, which cause different defect patterns [5], [7]. For example, in our Eclipse JDT dataset, 39% of one developer's changes related to Boolean assignment are buggy, while the percentage is only 7% for another developer. If we consider all developers' change data and combine them to build a global prediction model, due to the different defect patterns among different developers, the performance of CC would be hurt. Thus, it is necessary to build PCC models.

To address the above need, Jiang, Tan, and Kim propose *personalized change classification* (PCC), which builds separate prediction models for different developers to predict defects [7]. To solve this problem, Jiang, Tan, and Kim propose PCC, weighted PCC, and PCC+. For each developer, PCC builds a separate prediction model based on the developer's historical data. Weighted PCC also builds a separate prediction model for each developer; however, the model is built from a training data that consist of the developer's own historical data (50%) and other developer historical data (50%). PCC+ is a metaclassifier that selects either PCC, weighted PCC, or CC to predict if a change is buggy or not. PCC+ has been shown to outperform PCC, weighted PCC, and CC.

Considering that a typical software project has a number of developers, for a developer, building a prediction model not only based on his/her change data, but also on other *relevant* developers' change data can improve performance. One difficulty is how other developers' change data can be used. Traditional CC methods build a single model using all developers' change data, which result in poorer performance since developers have different defect patterns. On the other hand, PCC only uses each developer's own change data without incorporating other developers' data. Weighted PCC merges each developer's own change data with other developers' change data to train a model. However, the other developers' change data are chosen randomly, and thus, irrelevant data can be included, which can degrade the quality of the learned model.

In this paper, we propose a new technique named *collective personalized change classification* (CPCC). CPCC builds collective models that combine personalized models built from different developers' change data using multiobjective GA. More specifically, CPCC first builds a personalized prediction model for each developer based on his/her own change data. Next, for each developer, CPCC combines the personalized models into a collective model by assigning different weights to the models with the purpose of maximizing two objective functions (i.e., F1-scores and cost effectiveness[1]). Each of these weights measures the *relevancy* of a developer data to the target developer. CPCC uses multiobjective GA to learn a good set of personalized weights for each developer. Finally, weighted predictions from these models are aggregated together to identify potentially buggy changes. To further improve classification performance, we propose CPCC+, which combines CPCC and PCC.

To evaluate CPCC+ and CPCC, we use two widely used metrics that were also used to evaluate PCC [7]: cost effectiveness [8]–[15] and F1-score [1], [8], [13]–[17]. Cost effectiveness evaluates prediction performance given a certain cost threshold, e.g., a certain percentage of code to inspect. In our case, we inspect a certain percentage of the number of LOCs in changes. For example, when a team has limited resource to inspect potentially buggy LOC, it is crucial that manually inspecting the top percentages of lines that are likely to be buggy can help developers discover as many bugs as possible. We use the same cost effectiveness setting as Jiang, Tan, and Kim [7], which measures the number of bugs that can be discovered by inspecting the top 20% LOCs based on the confidence levels that a CC technique outputs (NofB20). In addition, we also evaluate our method using the F1-score [1], [8], [16], [17], which is a summary measure that combines both precision and recall. F1-score is a good evaluation metrics when there is enough test resource to inspect all predicted buggy changes. A higher F1-score means that a method can detect more buggy changes (true positives) and reduce the time wasted on inspecting clean changes.

We perform experiments on six large software projects from different communities: Eclipse JDT, Jackrabbit, Linux kernel, Lucene, PostgreSQL, and Xorg. The experiment results show that CPCC+ can discover up to 245 more bugs than PCC+ (468 versus 223 for PostgreSQL) if developers inspect the top 20% LOC that are predicted buggy. On average across the six datasets, CPCC+ can discover 122 more bugs than PCC+ (371 versus 249 on average), which improves PCC+ by 49.0%. In addition, our approach improves the F1-scores by 0.02–0.04 compared to PCC+, which are statistically significantly higher than those of PCC+ on all of the six projects. We address the following research questions:

RQ1: How effective is CPCC+ and CPCC? How much improvement can they achieve over the state-of-the-art method?
On average across the six projects, CPCC+ improves PCC+ by 0.02 and 122 in terms of F1 and NofB20 scores, respectively. In most cases, the improvements are statistically significant.

RQ2: How effective are CPCC+, CPCC, and PCC+ when different percentages and number of LOC are inspected?
We find that CPCC+ detects more defects than PCC+ for a wide range of percentages of LOC to inspect.

RQ3: How effective are CPCC+, CPCC, and PCC+ when different underlying classifiers are used?

We find that CPCC+ outperforms PCC+ in ADTree, Naive Bayes, and Logistic Regression. Also, CPCC+ with ADTree as the underly classifier achieves the best performance.

RQ4: How much time does it take for CPCC+ and CPCC to run?

We find that the model building and prediction time for CPCC+ and CPCC are reasonable. On average, CPCC+ and CPCC need about 4.449 and 4.200 s to train a model, and 0.011 and 0.010 s to predict the label of an instance using the model.

The main contributions of this paper are as follows:

1) We propose CPCC, which utilizes the advantages of multiobjective GAs to combine different developers' change data. Based on CPCC, we propose CPCC+ to further improve the performance.
2) We compare our method with PCC+ on 6 large software projects. The experiment results show that our method can achieve significant improvement over PCC+.

The remainder of this paper is organized as follows. We describe some preliminary materials on CC and a motivating example in Section II. We describe the high-level architecture of CPCC in Section III. We elaborate the CPCC approach details in Section IV. We elaborate CPCC+ approach details in Section V. We present our experiment setup and results in Sections VI and VII. We discuss additional points on the benefits and limitations of our approach in Section VIII. We present related work in Section IX. We conclude and mention future work in Section X.

II. Preliminaries

In this section, we first introduce the basic concepts of CC in Section II-A. Next, we describe the feature used in this paper in Section II-B. Then, we present the usage scenario of our proposed tool in Section II-C. Finally, we present the motivation of building a compositional model in Section II-D.

[1] For more details of cost effectiveness, see Section IV.

A. Change Classification

CC aims to predict if a particular file involved in a commit (i.e., a change) is buggy or not. Traditional CC techniques typically follow the following steps.

1) *Training Data Extraction:* For each developer's change, label it as buggy or clean by mining a project's revision history and bug tracking system [7], [18]. Buggy change means the change contains bugs (one or more), while clean change means the change has no bug. To identify which change is buggy, we follow the methods proposed by previous works [7], [18]. We first identify bug-fixing changes from the commit logs by searching the commit logs with the keyword "fix." We assume the lines which are modified in the bug-fixing changes are the locations of a bug. Next, we use the command git blame, which annotates each line in a source code file with the most recent change that modified that line. We then intersect the locations of the bug with the *git blame* annotations to find the bug-introducing changes (aka., buggy changes).
2) *Feature Extraction:* Extract the values of various features from each change. Many different features have been used in past CC studies.
3) *Model Learning:* Build a model by using a classification algorithm based on the labeled changes and their corresponding features. In this paper, by default, we use ADTree [19] to construct the prediction model.
4) *Model Application:* For a new change, extract the values of various metrics. Input these values to the learned model to predict whether the change is buggy or clean.

PCC modifies the above process by constructing not only one model during the model learning step, but rather a number of personalized models each trained using a developer historical data. In this work, similar to PCC, we still create multiple models each for a particular developer. However, each model is now a collective one and takes into consideration not only the developer historical data, but also the historical data of other relevant developers.

B. Features

In this paper, we consider three types of features, i.e., characteristic vector, word vector, and metadata. These features were used by Jiang, Tan, and Kim in their PCC work [7].

1) Characteristic Vector: A characteristic vector stores the number of times each node type appears in the abstract syntax tree (AST) of a code. Given a change, we build two characteristic vectors: one for the source code file before the change and one for the source code file after the change. We take the difference in the characteristic vector of the code prior to the change and the code after the change. Each element in the characteristic vector is a feature. Notice that the number of features in the characteristic vector are large because there are a large number of nodes in the AST.

```
for(int i = 0; i < 100; i++)
    for(int index = 0;index < i; index++)
        sum+ = array[i][j];
```

Fig. 1. Example of source code before the change.

```
for(int i = 0;i < 100; i++)
    for(int index = 0; index < i; index++)
        if(array[i][j] > 0) sum+ = array[i][j];
```

Fig. 2. Example of source code after the change.

Figs. 1 and 2 present an example of source code before and after the change. Suppose we only use if, for, and while node types for characteristic vectors. The characteristic vector of the code shown in Fig. 1 is $\langle 0, 2, 0\rangle$. And the characteristic vector of the code shown in Fig. 2 is $\langle 1, 2, 0\rangle$. We then subtract the two characteristic vectors to obtain the difference, i.e., the difference for the code before and after the change is $\langle 1, 0, 0\rangle$.

2) Word Vector: Multiset of word tokens that appear in the commit message and source code of a change. Each word token in the multiset is a feature. We use the Snowball[2] stemmer in Weka to convert the strings into word vectors and represent them as the form of bags of words. Notice that the number of word features are large due to the large number of terms in the commit message.

3) Metadata: In addition to characteristic vector and word vector, we also use a number of metadata features: commit hour (0, 1, 2, ..., 23), commit day, cumulative change count, cumulative buggy change count, source code file/path names, and file age (in days).

C. Usage Scenario

In this paper, we aim to build a prediction tool which identifies the buggy changes early on. The goal is to highlight defect-prone changes to improve the quality of source code. In practice, due to limited time budget and tight project schedule, developers are likely to inspect only a limited number of potentially buggy LOC to identify defect-prone changes. In such a case, it is crucial that manually inspecting the top percentages of lines that are likely to be buggy can help developers discover as many bugs as possible.

The following scenarios illustrate the benefits of our tool. *Scenario 1—Without Tool:* Xin joins a new software project team as a developer. The release date for the project is approaching, so the project manager asks Xin to inspect the code to identify as many buggy code as possible. One problem for Xin is that he does not know where to begin to inspect the code, and he decides to inspect the code from the first file to the last file in the project. When Xin has inspected just about 20% of the source code, the project needs to be released. Xin only finds 20 bugs, and 180 bugs remain in the released product. Most customers encounter many bugs when using various features of the product and are extremely unhappy.

Scenario 2—With Tool: Xin joins a new software project team as a developer. The release date for the project is approaching, so the project manager asks Xin to inspect the code to identify as many buggy code as possible. Xin first mines historical data from git, uses our tool to identify likely buggy changes, ranks

[2] http://snowball.tartarus.org/

TABLE I
F1-Score and Cost Effectiveness (NofB20) Scores of Various Models for the Two Target Developers (Dev 1 and Dev 2) From Eclipse JDT

Models	Target Dev 1		Target Dev 2	
	F1-score	NofB20	F1-score	NofB20
P	0.55	44	0.63	39
O_1	0.48	23	0.60	33
O_2	0.43	**61**	0.46	33
O_3	0.29	25	0.49	**46**
O_4	0.19	35	0.28	26
O_5	0.49	38	0.26	32
O_6	0.13	40	0.26	29
O_7	0.26	56	0.14	24
O_8	0.26	51	0	14
All	0.54	35	0.43	26

The first row corresponds to the performance of the model built using the target developer's own change data. The remaining nine rows corresponds to the performance of the model built using the remaining eight developers' change data.

the changes from high to low confidence scores, and inspects the code in the changes one by one. When Xin inspects about 20% of the source code, the project needs to be released. Xin finds 180 bugs, and 20 remains in the released product. Only a minority of the customers encounter one or a few bug when using several features of the product, and only a few are mildly unhappy.

D. Motivating Example

The effectiveness of our collective personalized CC technique relies on one primary hypothesis:

> *A model built from a relevant developer's historical data can potentially be used to predict buggy changes of a target developer.*

Here, we try to validate this hypothesis. To do so, we select ten developers from Eclipse JDT. We pick two developers as target developers (Dev 1 and Dev 2), and we investigate whether a personalized model learned from each of the remaining eight developers (we refer them as source developers) can be used to predict buggy changes made by the target developers. We use ADTree to learn models from each of the eight developers—we denote these models as $O_1, O_2, \ldots, O_8$. We also combine all of the eight developers' data to train a global model All. Next, we use the same algorithm to learn a model from the target developer own data—we denote this model as P. We then test the models on the changes made by Dev 1 and Dev 2, and we measure the quality of these models by computing F1-score and cost effectiveness using tenfold cross-validation setup. More specifically, for each target developer, we divide the change data of the target developer into ten equal-sized folds, and we choose nine folds of the data to train a classifier and evaluate the performance of a model in the remaining fold; the above process iterates ten times and the aggregate scores across the ten iterations are reported.

Table I presents the F1-score and cost effectiveness (NofB20) of various models for the two target developers. We notice that the global model All does not perform well; its NofB20 scores are lower than the other models. For Dev 1, the F1-score and NofB20 are 0.55 and 44, respectively, when the model built using his/her own change data (i.e., P) is used. Interestingly, we notice that other models can achieve similar or even better performance than P. In particular, using model O_2, the F1-score and cost effectiveness are 0.43 and 61, respectively. Notice that O_2's cost effectiveness score is even higher than that of the target classifier. For Dev 2, we also observe a similar result. For example, when O_3 is used the cost effectiveness score is 46, which is higher than that of the target classifier P. Thus, if we find a way to better use other developer data, the performance of CC can potentially be further improved.

From Table I, we also notice that the same model (i.e., O_1–O_8) has different performance when evaluated on different target developers' change data. For example, the F1-score and cost effectiveness for O_3 are 0.29 and 25 when evaluated on target developer 1's change data, which are lower than many other models. We note that the F1-score and cost effectiveness for O_8 are 0 and 14, respectively, when it is evaluated on target developer 2's change data, which are much lower than the other models. We manually check the results, and we find that when we build a prediction model by using developer 8's change data, it will predict all the changes in the target developer 2's change data as clean C since it could not identify any true positive, its precision, recall, and F1-score are all zeroes. However, the same classifier could achieve F1-score and cost effectiveness of up to 0.49 and 46 when evaluates on target developer 2's change data, which are higher than many other models. We refer to this phenomenon as *target difference*.

Moreover, in Table I, we notice that even for the same target developer, different models exhibit different performance. For example, for target developer 1, O_2, O_7, and O_8 achieve much better cost effectiveness scores than the remaining models. For target developer 2, O_1, O_2, O_3, and O_5 achieve much better cost effectiveness scores than the remaining models. We refer to this phenomenon as *source difference*.

We also check the F1-score and cost effectiveness for the other eight developers by using the same setting as we do for the target developers 1 and 2. And we find the phenomenon of *target difference* and *source difference* still exist. For example, when we predict the buggy changes for developer 4, we find the prediction model built on developer 2's change data (i.e., O_2) shows good performance, i.e., it achieves an F1-score and cost effectiveness of 0.54 and 57, respectively. But the prediction model built on developer 5's change data (i.e., O_5) shows bad performance, i.e., it achieves an F1-score and cost effectiveness of 0.21 and 27, respectively. Also, when we predict the buggy changes for developer 4, prediction models built on developer 1, 2, or 6's change data (i.e., O_1, O_2, or O_6) achieve much better performance than the other models. And when we predict the buggy changes for developer 7, prediction models built on developer 3 or 8's change data (i.e., O_3 or O_8) achieve much better performance than the other models.

Due to the phenomenon of *target difference* and *source difference*, typically no single model always perform best on different target developers' change data. Thus, a collective model which utilizes the advantages of different models can help to reduce the effect of *target difference* and *source difference* and further improve the performance of personalized defect prediction.

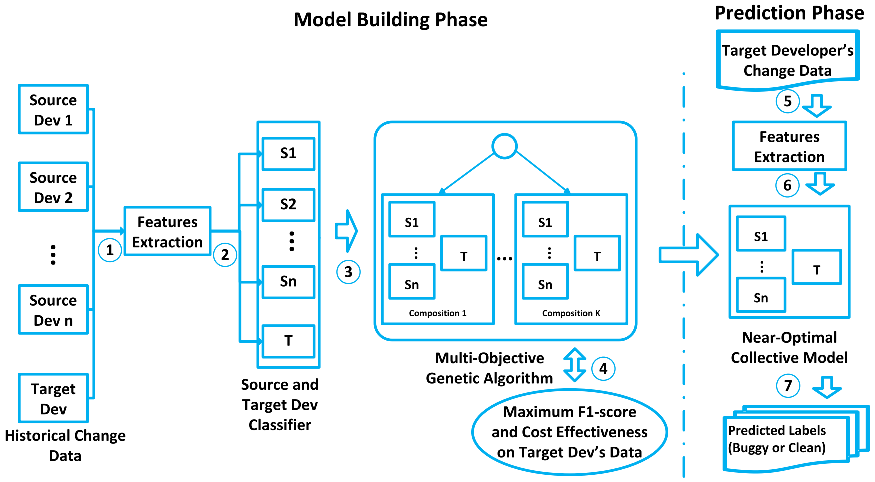

Model Building Phase
Prediction Phase
Source Dev 1
Source Dev 2
Source Dev n
Target Dev
Historical Change Data
1
Features Extraction
2
S1
S2
Sn
T
Source and Target Dev Classifier
3
S1
Sn
T
Composition 1
S1
Sn
T
Composition K
Multi-Objective Genetic Algorithm
4
Maximum F1-score and Cost Effectiveness on Target Dev's Data
Target Developer's Change Data
5
Features Extraction
6
S1
Sn
T
Near-Optimal Collective Model
7
Predicted Labels (Buggy or Clean)

Fig. 3. Overall architecture of CPCC.

To achieve this goal, in this paper, we build a collective model using our proposed technique CPCC.

III. CPCC Architecture

Fig. 3 presents the architecture of our CPCC framework. CPCC contains two phases: model building phase and prediction phase. In the model building phase, CPCC builds a collective model for each target developer, which is learned from historical change data of source developers and the target developer. In the prediction phase, we apply this model to predict if each of the target developer's new changes is defective or not.

Our framework takes as input historical changes from various source developers and the target developer with known labels (i.e., *buggy* or *clean*). Next, it extracts the values of various features from these changes (Step 1). In this paper, we use the features as shown in Section II-B. These features were also previously used by Jiang, Tan, and Kim for PCC+ [7]. Then, our framework builds multiple personalized prediction models based on the extracted feature values (Step 2). In total, we have $(n+1)$ prediction models, where the first n models are built from the source developers' changes, and the final model is built from the target developer's changes. Next, CPCC searches for the near-optimal composition of these models by leveraging a multiobjective GA (Step 3). The algorithm picks a composite (or collective) model that maximizes F1-score and cost effectiveness (NofB20) when it is used to predict the labels of the target developer's historical changes (Step 4). The near-optimal composition model is a machine learning classifier which assigns labels (in our case: *buggy* or *clean*) to a change based on its feature values.

After the model is constructed, in the prediction step, it is then used to predict whether each of the target developer's new changes is defective or not. For each new change, we first extract the values of the same set of features as those considered in the model building step (Step 5). We then input the values of these features into the learned model (Step 6). It will output a prediction result which is one of the following labels: *buggy* or *clean* (Step 7).

IV. CPCC Approach

In this section, we elaborate how CPCC builds a collective model for a target developer in a software project which corresponds to Steps 2–4 in Fig. 3. We also elaborate how the model can be used to predict if an unknown change is buggy or clean (Step 5 in Fig. 3).

Assuming n source developers (i.e., other developers in the project), CPCC first builds a total of $(n+1)$ prediction models (or classifiers[3]): for a source developer $Dev_i, \{1 \leq i \leq n\}$, it builds a model S_i from his/her historical change data; for the target developer $Target$, it builds the $(n+1)$th model T from $Target$'s historical change data. CPCC then searches for a near-optimal collective model containing these $(n+1)$ models along with a set of weights and a threshold to decide if a change is buggy or not. We refer to this collective model as the CPCC *classifier*. A multiobjective GA is used to learn the best values for these weights and the threshold. It will search for a solution (i.e., a collective model) in a search space based on a set of objective functions. The best solution is then outputted.

In Section IV-A, we formally define the CPCC classifier and how it can be used to predict if a change is clean or buggy. We formally present the search space of all possible combinations of $n+1$ classifiers to construct the CPCC classifier in Section IV-B. Section IV-B also introduces the objective functions. We elaborate the detailed procedure of how multiobjective GA is used to learn the CPCC classifier in Section IV-C.

A. *CPCC Classifier*

A CPCC classifier integrates the $n+1$ prediction models (classifiers): $S_1, \ldots, S_n$ and T. Given a change j, a model S_i outputs a confidence score that indicates the likelihood of j to be buggy, which is denoted as $\text{Score}_i(j)$. Similarly, we denote the confidence score of the target classifier T to j as $\text{Score}_t(j)$. The range of a confidence score is from 0 to 1. A CPCC classifier computes a weighted sum of all confidence scores assigned by

[3] We use the term model and classifier interchangably in this paper.

the $(n+1)$ classifiers and predicts whether j is buggy or not based on a threshold score. Definition 1 provides a mathematical definition of the CPCC classifier.

Definition 1 (CPCC Classifier): Consider n source classifiers $\{S_1, S_2, \ldots, S_n\}$ and a target classifier T. A CPCC classifier composes these $(n+1)$ classifiers and assigns a label to an instance j, denoted as Label(j), as follows:

$$\text{Label}(j) = \begin{cases} 1 \text{ (i.e., buggy)}, & if\ CPCC(j) \geq \text{threshold} \\ 0 \text{ (i.e., clean)}, & \text{Otherwise} \end{cases}$$

where

$$\text{CPCC}(j) = \frac{\sum_{i=1}^{n} \alpha_i \times \text{Score}_i(j) + \alpha_t \times \text{Score}_t(j)}{\text{LOC}(j)}. \quad (1)$$

In the above equation, Score$_i(j)$ is the confidence score outputted by the ith source classifier for instance j to be buggy, α_1–α_n are the weights of the n source classifiers, Score$_t(j)$ is the confidence score outputted by the target classifier T for instance j to be buggy, α_t is the weight of the target classifier, threshold is the boundary used to decide whether an instance is buggy or not, and LOC(j) is the number of LOCs for instance j. CPCC(j) is the composite confidence score for instance j to be buggy, and we set it as a linear combination of the confidence scores outputted by the n source classifiers and the target classifier T. Instance j is classified as buggy if its composite confidence score CPCC(j) is larger than or equal to threshold; otherwise, it is classified as clean. Note that α_1–α_n, α_t, and threshold are the parameters of a CPCC classifier. Thus, we denote a CPCC classifier as $(\bigcup i = 1^n \{(\alpha_i, S_i)\}$, (α_t, T), threshold), where each S_i is a source classifier, α_i is the weight of S_i, T is a target classifier, α_t is the weight of T, and threshold is the defect boundary.

We include LOC in (1) to maximize the number of buggy changes found given a budget (e.g., inspecting only 20% of the number of LOC). If two changes have equal likelihood to be buggy and one of them has a higher LOC, to find as many bugs as possible within the budget, we need to pick the change with the lower LOC.

Note that the CPCC score may be larger than 1—since the numerator of formula (1) may be larger than 1, and the denominator of formula (1) may be equal to 1. This may make the output of CPCC noninterpretable since the range of values that it can take is very large. Still, since the CPCC score is only an intermediary score, we believe it is okay for the score to be noninterpretable. If an interpretable score is needed, we can normalize the output of the CPCC formula by dividing it with the sum of the weights (i.e., $\alpha_1 + \cdots + \alpha_n + \alpha_t$). If we also normalize the threshold in the same way, this modification will not affect the inferred labels.

Example: Consider a change j which has 100 LOCs. Suppose we have three source classifiers and one target classifier. Let the confidence scores of the three source classifiers and the target classifier for j be 0.4, 0.5, 0.8, and 0.9, respectively. Also, let the weights of the source classifiers and the target classifiers be 0.3, 0.7, 0.7, and 0.8, respectively. From the above, the composite confidence score for j is

$$\frac{0.3 \times 0.4 + 0.7 \times 0.5 + 0.7 \times 0.8 + 0.8 \times 0.9}{100} = 0.0175.$$

If the bug boundary is less than or equal to 0.0175, then CPCC predicts j as buggy, else it predicts it as clean.

B. Search Space and Objective Functions

1) Search Space: The search space of all possible compositions corresponds to the various assignments of values to the weights $\alpha_1, \alpha_2, \ldots, \alpha_n$ of the source classifiers, the weight α_t of the target classifier, and the bug boundary threshold. Each weight is a real number from zero to one and threshold is a positive real number. We refer to each composition as a solution in the search space, and it comprises of a set of parameters Par $= \{\alpha_1, \alpha_2, \ldots, \alpha_n, \alpha_t, \text{threshold}\}$.

2) Objective Functions: An objective function measures the quality of a solution in a search space. We have two objective functions that we try to maximize:

$$F1(\text{Par})$$
$$\text{cost(Par)}. \quad (2)$$

In the above equation, $F1$(Par) and cost(Par) are the F1-score and the cost effectiveness (NofB20) achieved by a CPCC classifier with parameters Par on a training data. The details on how F1-score and cost effectiveness are computed are given in Section VI-C. Notice that we choose the objective functions as F1 and cost effectiveness, since these two evaluation metrics are widely used in defect prediction literature [1], [8]–[12], [16], [17], and we would like our constructed model to not only achieve good F1-score but also have good cost effectiveness score.

Since we have two different objective functions to be maximized, we find a set of Pareto optimal solutions (cf., [17], [20]) defined in Definition 2.

Definition 2 (Dominance and Pareto optimal solutions): A set of parameters Par$_i$ dominates another set of parameters Par$_j$ if and only if the values of the two objective functions (i.e., F1-score and cost effectiveness) satisfy

$$\text{F1}(\text{Par}_i) > \text{F1}(\text{Par}_j) \text{ and cost}(\text{Par}_i) \geq \text{cost}(\text{Par}_j)$$
$$\text{or}$$
$$\text{F1}(\text{Par}_i) \geq \text{F1}(\text{Par}_j) \text{ and cost}(\text{Par}_i) > \text{cost}(\text{Par}_j).$$

A set of parameters Par$_i$ is Pareto optimal if and only if no other set of parameters dominates it in the feasible region, i.e., no other set of parameters Par$_j$ exists that would improve the F1-score, without worsening cost effectiveness scores, and vice versa.

There would be a number of solutions which are Pareto optimal. We further reduce these solutions by selecting the following subset

$$\text{ParSet}_{\text{selected}} = \underset{\text{Par}_i \in \text{Pareto}}{\text{argmax}}\ F1(\text{Par}_i) \times \text{cost}(\text{Par}_i). \quad (3)$$

In other words, we select solutions from the Pareto optimal set which has the highest product of F1-score and cost effectiveness scores. We randomly pick a solution from this set as the near-optimal solution.

C. Detailed Procedure

We employ a multiobjective GA [20] to learn the weights (α_1–α_n and α_t) and the threshold (threshold). In the multiobjective GA, solutions in a search space are modeled as chromosomes. A chromosome contains a set of genes where a gene corresponds to a part of a solution (i.e., a value of a weight, or a value of the threshold, in our setting). The multiobjective GA begins with an initial set of random chromosomes, referred to as the initial population. Next, the population evolves in a number of iterations, and in each iteration, it generates subsequent generations, where each generation is a population of chromosomes. To create the next generation, multiobjective GA will do three operations: selection, crossover, and mutation. Selection operation refers to the process that selects parent chromosomes according to their objective function scores (F1-score and cost effectiveness); thus, the parent chromosomes that have high objective function scores have high probability to survive in the next generation. Crossover operation refers to the process that merges the genes of two parent chromosomes to form offspring chromosomes according to a given probability. Mutation refers to the process that the genes of offspring chromosomes would be modified according to a given probability. More details about multiobjective GA can be found in [20].

There are many variants of the multiobjective GA; in this paper, we use an algorithm named vNSGA-II, which is proposed by Deb, Pratap, Agarwal, and Meyarivan [21]. In our setting, each chromosome is represented as an array of $(n+2)$ double; $(n+1)$ double represent the weights (α_1–α_n and α_t), and the last double represents the threshold. We use the Roulette wheel selection procedure [22], [23] as the selection operator. It assigns a high probability to a chromosome with higher objective function scores. For the crossover operator, we use the simulated binary crossover (SBX) operator [24]. For the mutation operator, we use the polynomial mutation (PM) operator [25]. SBX and PM attempt to simulate the offspring distribution of binary-encoded bit-flip crossover and mutation on real-valued variables, respectively.

Algorithm 1 presents the detailed steps to train a CPCC classifier. For the ith source developer's historical data D_i, we first build a classifier S_i based on instances in D_i (lines 9–11). Similarly, we build a classifier T using the target developer's data D_t alone (line 12). Then, we create an initial population (i.e., P) containing PopSize chromosomes (i.e., solutions) that are chosen randomly, and we record the set of nondominated solutions (i.e., the solutions with nondominated F1-score and cost effectiveness scores on D_t) among the solutions in P (lines 13 and 14). Remember that each solution in P is a set of weights $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, α_t, and a threshold. Next, we evolve the population in MaxGen iterations; for each iteration, we perform the selection, crossover, and mutation operations on the current population and record the set of nondominated solutions so far (lines 16–22). After the iterations complete, we find the set of Pareto optimal solutions from the set of nondominated solutions (line 23). The algorithm returns a set of $\alpha_1, \alpha_2, \ldots, \alpha_n$, α_t, and threshold values, which achieves the highest product of F1-score and cost effectiveness in the set of Pareto optimal solutions. If there are more than one of such near-optimal set, we randomly pick one of them.

Algorithm 1: The Model Building Phase of CPCC.

```
 1: CPCC($\{D_1, D_2, \ldots, D_n\}$, $D_t$, PopSize, MaxGen)
 2: Input:
 3: $\{D_1, D_2, \ldots, D_n\}$: Historical data of source
    developers
 4: $D_t$: Historical data of the target developer
 5: $PopSize$: Number of chromosomes in a population
 6: $MaxGen$: Maximum number of generations
 7: Output: Composite CPCC Classifier ($\sum_{i=1}^{n} \alpha_i S_i, \alpha_t T$,
    threshold).
 8: Method:
 9: for all $D_i \subseteq \{D_1, D_2, \ldots, D_n\}$ do
10:    Build a classifier $S_i$ by using $i$th source developer's
       historical data;
11: end for
12: Build a classifier $T$ by using the target developer's
    historical data $D_t$;
13: Let $P =$ Initial population with $PopSize$ members;
14: Evaluate $P$ and record the set of nondominated
    solutions (i.e., the solutions with nondominated
    F1-score and cost effectiveness scores on $D_t$) so far;
15: Let $curGen = 0$
16: while $curGen < MaxGen$ do
17:    Let $P^{'} = select(P)$;
18:    $P^{'} = crossover(P^{'})$;
19:    $P^{'} = mutation(P^{'})$;
20:    Evaluate $P^{'}$ and record the set of non dominated
       solutions so far;
21:    $curGen = curGen$ + 1;
22: end while
23: Find the set of Pareto optimal solutions;
24: Output ($\sum_{i=1}^{n} \alpha_i S_i$, $\alpha_t T$, $threshold$) which achieves
    the highest product of F1-score and cost effectiveness
    scores in the set of Pareto optimal solutions.
```

D. Complexity Analysis

In our CPCC, we use a multiobjective GA (i.e., vNSGA-II [20], [26]–[28]) to find good values of some parameters (i.e., the weights and the threshold), and we use ADTree [19] as the default underlying classifier. Suppose we have M objective functions and a population size of N; then, the time complexity of vNSGA-II is $O(M \times N^2)$. Suppose our dataset have P features, and Q number of instances, the time complexity of ADTree [19] is $O(P \times Q^2)$. Thus, the total time complexity of CPCC is $O(M \times N^2 + P \times Q^2)$.

V. CPCC+: A Composite Approach

To further improve classification performance, we propose CPCC+ which combines CPCC and PCC. Similar to CPCC,

CPCC+ also has two phases: model building phase and prediction phase. In the model building phase, for each target developer, CPCC+ builds a prediction model using CPCC and another one using PCC. In the prediction phase, for a new unlabeled change, CPCC+ predicts the change as clean or buggy using the two prediction models built in the model building phase. It first obtains the confidence scores of an instance to be buggy that are outputted by the two prediction models. It then normalizes the scores produced by CPCC and PCC so that their scores are comparable. It does this by adjusting the CPCC confidence score for instance j to be buggy as follows:

$$\mathrm{CPCC}^{\mathrm{Adj}}(j) = \frac{\mathrm{CPCC}(j)}{\mathrm{CPCC}(j) + \mathrm{threshold}}. \tag{4}$$

Note that we only normalize the CPCC scores [using (4)]. We do not normalize the PCC scores since they are already normalized. They are probabilities outputted by a classification algorithm and their scores are already between zero and one. For CPCC scores, they can be larger than one, and thus, we normalize them.

Next, based on the adjusted CPCC confidence score and the PCC confidence score, it chooses the prediction with the highest confidence as the final prediction.

More formally, for a new change, new, let us denote the larger of $\mathrm{CPCC}^{\mathrm{Adj}}$'s and PCC's confidence of new to be buggy as $\mathrm{Score}^{\max}_{\mathrm{buggy}}(\mathrm{new})$, and the larger of $\mathrm{CPCC}^{\mathrm{Adj}}$'s and PCC's confidence of new to be clean[4] as $\mathrm{Score}^{\max}_{\mathrm{clean}}(\mathrm{new})$. Based on these notations, CPCC+ assigns a label to new as follows:

$$\mathrm{Label}(\mathrm{new}) = \begin{cases} 1 \text{ (i.e., buggy)}, & \text{if CPCC+(new)} \geq 0 \\ 0 \text{ (i.e., clean)}, & \text{Otherwise} \end{cases}$$

where

$$\text{CPCC+(new)} = \frac{\mathrm{Score}^{\max}_{\mathrm{buggy}}(\mathrm{new}) - \mathrm{Score}^{\max}_{\mathrm{clean}}(\mathrm{new})}{\mathrm{LOC}(\mathrm{new})}. \tag{5}$$

In the above equation, $\mathrm{CPCC} + (\mathrm{new})$ is CPCC+'s confidence that new is buggy, and $\mathrm{LOC}(\mathrm{new})$ is the number of LOCs in change new. The LOC does not determine if a change is predicted as buggy or not. We use LOC to help rank the predicted buggy changes. For two changes with similar likelihood to be buggy, we will rank the one with less LOC first so as to maximize the number of bugs found given an inspection budget.

A. Complexity Analysis

As shown in Section IV-D, the total time complexity of CPCC is $O(M \times N^2 + P \times Q^2)$. In CPCC+, we combine CPCC and PCC. Suppose our dataset have P features and Q number of instances, the time complexity of PCC using ADTree is $O(P \times Q^2)$. Thus, the total time complexity of CPCC+ is $O(M \times N^2 + P \times Q^2)$.

TABLE II
Statistics of Collected Data

Project	Language	LOC	Time	% Buggy
Eclipse	Java	1.5 M	2006-06-07 – 2006-01-24	23.0%
Jackrabbit	Java	589 K	2007-09-13 – 2009-09-15	46.4%
Linux	C	7.3 M	2008-01-23 – 2008-07-15	21.0%
Lucene	Java	828 K	2010-09-17 – 2011-06-30	31.0%
PostgreSQL	C	289 K	1998-07-08 – 2010-02-14	40.9%
Xorg	C	1.1 M	2003-07-02 – 2009-07-24	23.1%

VI. Experiment Setup

In this section, we describe the settings that we follow to evaluate the effectiveness of our proposed CPCC and CPCC+. We first present our dataset, cross-validation settings, and experimental environment. We then present our parameter setting and evaluation metrics.

A. Datasets, Cross-Validation Settings, and Environment

We use the same datasets as Jiang, Tan, and Kim [7], which contain changes from six open-source software projects: Eclipse JDT, Jackrabbit, Linux, Lucene, PostgreSQL, and Xorg. Table II presents the statistics of Jiang, Tan, and Kim's data. The columns correspond to the programming language (Language), LOCs (LOC), the time period of collected changes (Time), and the percentage of buggy changes (% Buggy Changes). Jiang, Tan, and Kim selected top ten developers from each project. For each of them, they collected 100 consecutive changes from the time period listed in Table II. Thus, in total, there are 1000 changes for each project.

Jiang, Tan, and Kim have shown that PCC+ outperforms CC, PCC, and weighted PCC; thus, in this work, since we use the same dataset, we only need to compare our technique CPCC+ and CPCC with PCC+. We perform tenfold cross validation 100 times and take the average performance to reduce the training set selection bias. For each tenfold cross validation, we randomly split the data into ten subsets. The random splitting is performed differently for each of the tenfold cross validations. Cross validation is a standard evaluation setting, which is widely used in software engineering studies; cf., [1], [4], [7], [9], [29]–[32].

Note that our CPCC+ and CPCC use a multiobjective GA (i.e., vNSGA-II) to optimize the values of some parameters (e.g., the weights and the threshold). To do this step, we create a validation set from a training data. For each training fold and each target developer j, we separate j's change data in the training data of that fold to create a validation set. Next, we use the multiobjective GA to optimize the parameters for this validation set. Furthermore, since multiobjective GA exhibits some randomness [20], [21], for each of the 100 tenfold cross validations, we run the multiobjective GA ten times and use the average performance score. This follows the recommendation made by Liu, Khoshgoftaar, and Seliya [33] and Canfora *et al.* [17] to evaluate random algorithms in software engineering context. PCC+ is a deterministic algorithm and thus there is no need to perform these repetitions.

[4] The confidence of new to be clean is one minus the confidence of new to be buggy.

The experimental environment is a Windows 8, 64-bit, Intel(R) Core(TM) 1.60-GHz laptop with 4-GB RAM.

B. Parameter Setting

Our multiobjective GA vNSGA-II [34] accepts some parameters. The parameters of vNSGA-II that we use in this study are as follows.

1) *Population size:* We set a moderate population size with PopSize $= 200$.
2) *Number of generations:* We set the maximum number of generations MaxGen $= 1000$.
3) *Crossover operator:* We use an SBX operator with probability $p_c = 0.6$.
4) *Mutation operator:* We use a PM operator with probability $p_m = 0.05$.

CPCC+, CPCC, and PCC+ use an underlying classifier. By default, we use the same classifier that was used by Jiang, Tan, and Kim to evaluate PCC+ [7], namely ADTree [19]. We use the implementation of ADTree that comes with Weka [35].

C. Evaluation Metrics

We use two evaluation metrics: cost effectiveness and F1-score. These two measures are useful in different situations. Cost effectiveness is useful when there are limited resources to inspect a limited amount of code due to a hectic schedule of development. F1-score is useful when there is sufficient resource to inspect all of the predicted buggy changes.

1) Cost Effectiveness: Cost effectiveness is a widely used evaluation metric for defect prediction [7]–[12], which evaluates prediction performance given a cost limit. In our setting, the cost is the LOC to inspect. We use the same cost effectiveness setup as the one used by Jiang, Tan, and Kim [7]. They measure the number of buggy changes that a developer can identify by inspecting the top 20% LOC with the highest confidence to be buggy. They refer to this number as NofB20. Consider different projects have different numbers of bugs; we also compute a percentage of the total number of bugs (PofB20) by normalizing the NofB20 scores .

To compute NofB20 and PofB20, we sort changes in the test data based on the confidence levels that a CC technique outputs for each of them. A change with a higher confidence level is deemed to be more likely to be buggy by the CC technique. We then simulate a developer that inspects these potentially buggy changes one at a time. As the changes are inspected one at a time, we accumulate the number of LOC that are inspected and the number of buggy changes identified. We stop the process when 20% of the LOC have been inspected and output the number of buggy changes that are identified. This number is the NofB20 score. A higher cost effectiveness score represents that a developer can detect more bugs when inspecting a limited number of LOC.

2) F1-Score: There are four possible outcomes for a change in the test data: A change can be classified as buggy when it truly is buggy (true positive, TP); it can be classified as buggy when it is actually clean (false positive, FP); it can be classified as clean when it is actually buggy (false negative, FN); or it can be classified as clean and it truly is clean (true negative, TN). Based on these possible outcomes, precision, recall and F1-score are defined as follows.

Precision: The proportion of changes that are correctly labeled as buggy among those labeled as buggy:

$$P = \text{TP}/(\text{TP} + \text{FP}). \quad (6)$$

Recall: The proportion of buggy changes that are correctly labeled:

$$R = \text{TP}/(\text{TP} + \text{FN}). \quad (7)$$

F1-score: A summary measure that combines both precision and recall—it evaluates if an increase in precision (recall) outweighs a reduction in recall (precision):

$$F = (2 \times P \times R)/(P + R). \quad (8)$$

There is a tradeoff between precision and recall. One can increase precision by sacrificing recall (and vice versa). In our framework, we can sacrifice precision (recall) to increase recall (precision), by manually lowering (increasing) the value of the threshold parameter in Definition 1. The tradeoff causes difficulties to compare the performance of several prediction models by using only precision or recall alone [36]. For this reason, we compare the prediction results using F1-score, which is a harmonic mean of precision and recall. This follows the setting used in many CC and defect prediction studies [1], [7], [16] and various software analytics studies [37], [38].

VII. Experiment Results

In this section, we present our experiment results which answer a number of research questions. We present these questions and their answers in the following subsections.

A. RQ1: How Effective is CPCC+ and CPCC? How Much Improvement Can They Achieve Over the State-of-the-Art Method?

Motivation: We need to compare CPCC+ and CPCC with the state-of-the-art CC method. Answer to this research question would shed light to the extent CPCC+ and CPCC advances the state of the art.

Approach: In a recent work, Jiang, Tan, and Kim have demonstrated that PCC+ outperforms a number of other CC methods [7]. Thus, to answer this research question, we compare CPCC+ and CPCC with PCC+ and CC. We compute NofB20, PofB20, and F1-score to evaluate the performance of these two approaches on the six projects. Also, in this paper, we use 100 times tenfold cross validation to evaluate each of the methods, and for each approach (i.e., CPCC+, CPCC, PCC+, and CC), at the end of each of the tenfold cross validation, it will output an F1 and PofB20 score. We apply Wilcoxon signed-rank test [40] on the 100 paired data to test whether the improvement of CPCC+ and CPCC over PCC+ and CC is significant. We also use Bonferroni correction [41] to counteract the results of multiple comparisons. We consider that the improvement is statistically significant if the p-value is less than 0.05 (i.e., at the confidence level of 95%) after Bonferroni correction.

TABLE III
CLIFF'S DELTA AND THE EFFECTIVENESS LEVEL [9]

Cliff's Delta ($\|\delta\|$)	Effectiveness Level
$\|\delta\| < 0.147$	Negligible
$0.147 \leq \|\delta\| < 0.33$	Small
$0.33 \leq \|\delta\| < 0.474$	Medium
$\|\delta\| \geq 0.474$	Large

We also use Cliff's delta (δ) [39], which is a nonparametric effect size measure that quantifies the amount of difference between two groups. In our context, we use Cliff's delta to compare CPCC+ and CPCC to the baseline approaches. For example, when compared the F1-score of CPCC+ with PCC+, the two groups are the F1-scores of CPCC+ and PCC+. The delta values range from -1 to 1, where $\delta = -1$ or 1 indicates the absence of overlap between two approaches (i.e., all values of one group are higher than the values of the other group, and vice versa), while $\delta = 0$ indicates that the two approaches are completely overlapping. Table III describes the meaning of different Cliff's delta values and their corresponding effectiveness level [39]. Notice that different from Wilcoxon signed-rank test which is used to test whether the improvements of CPCC+ and CPCC over the baseline approaches are statistically significant, Cliff's delta is used to test whether the improvements of CPCC+ and CPCC over the baseline approaches are of substantially large amount. If the effect size is medium or large, we consider the improvement to be substantial.

Results: Table IV presents the experiment results comparing CPCC+ and CPCC with PCC+ and CC. The experiment results are a little different than what were reported in [7]. This is the case as the tenfold cross validation used in our experiments randomly partitions the dataset into ten sets. Due to the randomness in the process, the resultant sets are different than those produced by the random partitioning performed in Jiang, Tan, and Kim's experiments. Also, differently from Jiang, Tan, and Kim's experiment setup, we run tenfold cross validation 100 times and record the average experiment results.

From Table IV, we can note that the F1, NofB20, and PofB20 scores of CPCC vary from 0.58 to 0.74, 197 to 651, and 48% to 63%, respectively. On average, across the six projects, CPCC can achieve F1, NofB20, and PofB20 scores of 0.64, 353, and 56%, respectively. In terms of NofB20, CPCC improves PCC+ and CC by 20–229 and 34–284, respectively. In terms of PofB20, CPCC improves PCC+ and CC by 6–32% and 11–40%, respectively. In terms of F1 score, CPCC improves PCC+ and CC by 0.01–0.02 and 0.00–0.13, respectively. On average, CPCC improves PCC+ in terms of NofB20, PofB20, and F1 score by 104, 15%, and 0.01 and improves CC in terms of NofB20, PofB20, and F1 score by 139, 21%, and 0.08, respectively.

We also notice that the F1, NofB20, and PofB20 scores of CPCC+ vary from 0.60 to 0.75, 205 to 678, 52% to 66%, respectively. On average, across the six projects, CPCC can achieve F1, NofB20, and PofB20 scores of 0.65, 371, and 59%, respectively. In terms of NofB20, CPCC+ improves CPCC, PCC+, and CC by 5–37, 20–229, and 44–300, respectively. In terms of PofB20, CPCC+ improves CPCC, PCC+, and CC by 1–4%, 9–34%, and 14–42%, respectively. In terms of F1 score, CPCC+ improves CPCC, PCC+, and CC by 0.01–0.02, 0.02–0.04, and 0.01–0.15, respectively. On average, CPCC+ improves PCC+ in terms of NofB20, PofB20, and F1 score by 122, 24%, and 0.02, respectively, and improves CC in terms of NofB20, PofB20, and F1 score by 157, 24%, and 0.09, respectively. Moreover, we notice that for NofB20 and PofB20, the performance gap between CC and PCC+ is smaller than the performance gap between CPCC+ and PCC+. On the other hand, for F1 score, the performance gap between CC and PCC+ is larger than the performance gap between CPCC+ and PCC+.

Moreover, we notice CPCC+ and CPCC are more stable than PCC+ and CC. From Table IV, we notice that the standard deviations for CPCC+ and CPCC are from 0.00 to 0.01 and 0% to 1% in terms of F1 and PofB20 scores, while these values for PCC+ and CC are from 0.01 to 0.03 and 3% to 4% in terms of F1 and PofB20 scores, respectively.

To investigate whether the improvements of CPCC+ over CPCC, PCC+, and CC are statistically significant and the effective sizes are large, we also compute the p-value in Wilcoxon signed-rank test and Cliff's Delta. Table V presents the p-values and Cliff's delta of comparing CPCC+ results with those of PCC+. Notice that in our study, we use Bonferroni correction to counteract the results of multiple comparisons; thus, the p-values are adjusted. We notice that the improvements of CPCC+ over PCC+ are statistically significant on all of the six projects in terms of F1-score and PofB20 at the 95% confidence level. Moreover, the effect sizes are mostly large.

Table VI presents the p-values and Cliff's delta of comparing CPCC+ results with those of CC. The p-values are adjusted by using Bonferroni correction. We notice that the improvements of CPCC+ over CC are statistically significant on all of the six projects in terms of F1-score and PofB20 at the 95% confidence level. Moreover, the effect sizes are mostly large.

Table VII presents the p-values and Cliff's delta of comparing CPCC+ results with those of CPCC. The p-values are adjusted by using Bonferroni correction. We notice that in terms of PofB20, the improvements of CPCC+ over CPCC are statistically significant at 95% confidence level. In terms of F1-score, except for project Eclipse, CPCC+ statistically significantly outperforms CPCC. Moreover, the effect sizes are mostly large.

Table VIII presents the p-values and Cliff's delta of comparing CPCC results with those of PCC+. The p-values are adjusted by using Bonferroni correction. We notice that the improvements of CPCC over PCC+ are statistically significant on all of the six projects in term of F1-score and PofB20 at the 95% confidence level. Moreover, the effect sizes are mostly large.

Table IX presents the p-values and Cliff's delta of comparing CPCC results with those of CC. The p-values are adjusted by using Bonferroni correction. We notice that the improvements of CPCC over CC are statistically significant on all of the six projects in term of F1-score and PofB20 at the 95% confidence level. Moreover, the effect sizes are mostly large.

On average across the six projects, CPCC+ improves PCC+ by 0.02 and 122 in terms of F1 and NofB20 scores, respectively.

TABLE IV
EXPERIMENT RESULTS OF CPCC+ AND CPCC COMPARED WITH PCC+ AND CC

Project	Method	Precision	Recall	F1-score	NofB20	PofB20
Eclipse	CPCC+	0.52±0.01	0.76±0.00	**0.62±0.01**	**299±6**	**52%±0%**
	CPCC	0.57±0.00	0.66±0.01	0.61±0.01	294±7	51%±0%
	PCC+	0.52±0.01	0.74±0.01	0.60±0.01	222±26	39%±4%
	CC	0.43±0.02	0.74±0.02	*0.51±0.02*	*185±26*	*32%±4%*
Jackrabbit	CPCC+	0.64±0.00	0.91±0.00	**0.75±0.00**	**678±5**	**60%±0%**
	CPCC	0.68±0.00	0.83±0.00	*0.74±0.00*	651±5	58%±0%
	PCC+	0.64±0.01	0.91±0.01	**0.75±0.01**	452±33	40%±3%
	CC	0.71±0.00	0.77±0.01	*0.74±0.01*	*434±30*	*39%±3%*
Linux	CPCC+	0.52±0.01	0.71±0.01	**0.60±0.01**	**205±3**	**66%±1%**
	CPCC	0.53±0.01	0.64±0.01	0.58±0.01	197±3	63%±1%
	PCC+	0.48±0.02	0.70±0.02	0.56±0.02	177±13	57%±4%
	CC	0.57±0.04	0.37±0.03	*0.45±0.03*	*161±12*	*52%±4%*
Lucene	CPCC+	0.48±0.01	0.79±0.01	**0.60±0.01**	**341±5**	**54%±1%**
	CPCC	0.52±0.00	0.68±0.01	0.59±0.00	304±5	48%±1%
	PCC+	0.51±0.01	0.71±0.01	0.58±0.01	255±21	40%±3%
	CC	0.67±0.02	0.37±0.03	*0.48±0.02*	*199±20*	*31%±3%*
PostgreSQL	CPCC+	0.54±0.01	0.82±0.01	**0.66±0.01**	**468±8**	**65%±1%**
	CPCC	0.56±0.00	0.77±0.01	0.64±0.00	452±8	63%±1%
	PCC+	0.54±0.01	0.78±0.01	0.63±0.01	223±30	31%±4%
	CC	0.68±0.02	0.51±0.02	*0.58±0.02*	*168±26*	*23%±3%*
Xorg	CPCC+	0.52±0.00	0.79±0.00	**0.66±0.00**	**235±3**	**57%±1%**
	CPCC	0.57±0.00	0.75±0.00	0.65±0.00	220±3	53%±1%
	PCC+	0.64±0.01	0.57±0.01	0.64±0.01	157±14	38%±3%
	CC	0.72±0.02	0.53±0.02	*0.61±0.02*	*136±14*	*33%±3%*
Average.	CPCC+		**0.65**		**371**	**59%**
	CPCC		0.64		353	56%
	PCC+		0.63		249	41%
	CC		*0.56*		*214*	*35%*

The results are in the form of mean±standard deviation (std). The highest and lowest results for each dataset are in bold and italic, respectively.

TABLE V
P-VALUE AND CLIFF'S DELTA OF COMPARING CPCC+ RESULTS WITH THOSE OF PCC+

Projects	F1-score			PofB20		
	p-value	Cliff's Delta	Effectiveness Level	p-value	Cliff's Delta	Effectiveness Level
Eclipse	$1.01e^{-5}$	0.37	Medium	$1.32e^{-15}$	1	Large
Jackrabbit	$1.32e^{-15}$	0.98	Large	$1.32e^{-15}$	1	Large
Linux	$5.10e^{-15}$	0.75	Large	$5.10e^{-15}$	0.75	Large
Lucene	$2.35e^{-13}$	0.76	Large	$1.32e^{-15}$	1	Large
PostgreSQL	$1.32e^{-15}$	1	Large	$1.32e^{-15}$	1	Large
Xorg	$5.84e^{-11}$	0.6014	Large	$1.32e^{-15}$	1	Large

In most cases, the improvements are statistically significant and the effective sizes are large.

B. RQ2: How Effective Are CPCC+, CPCC, and PCC+ When Different Percentages and Number of LOC Are Inspected?

Motivation: By default, we set the percentage of LOC to inspect as 20%. In this RQ, we also investigate the performance of CPCC+, CPCC, and PCC+ when different percentages of LOC are inspected. Additionally, we also investigate the effectiveness of CPCC+, CPCC, and PCC+ when a fixed budget is specified, i.e., an absolute number of LOC to inspect. Answer to this research question can verify whether CPCC+ and CPCC still improve PCC+ for other cost settings.

Approach: To answer this research question, we plot cost effectiveness graphs that show the percentages of bugs that can be detected by inspecting different percentages of LOC. Moreover, we also compute the area under the curve (AUC) in these graphs. In our graphs, we denote the percentage of LOC to inspect as p, and the PofB score as PofB; then, the AUC value is computed by

$$\text{AUC} = \int \text{PofB} \times p \, dp. \tag{9}$$

Results: Fig. 4 presents the cost effectiveness graphs for Eclipse JDT, Jackrabbit, Linux, Lucene, PostgreSQL, and Xorg. We can note that inspecting 20% of the LOC, CPCC+ could identify more than 50% of the defects in all of the six projects. We also notice that CPCC+ is better than PCC+ for a wide range of percentages of LOC to inspect. For Eclipse JDT and Xorg, the percentage range for which CPCC+ achieves the best performance are wide, i.e., from around 1% to 95%. For PostgreSQL, the percentage range for which CPCC+ achieves

TABLE VI
P-VALUE AND CLIFF'S DELTA OF COMPARING CPCC+ RESULTS WITH THOSE OF CC

Projects	F1-score			PofB20		
	p-value	Cliff's Delta	Effectiveness Level	p-value	Cliff's Delta	Effectiveness Level
Eclipse	$1.32e^{-15}$	1	Large	$1.32e^{-15}$	1	Large
Jackrabbit	$1.22e^{-12}$	0.66	Large	$1.32e^{-15}$	1	Large
Linux	$1.32e^{-15}$	1	Large	$1.32e^{-15}$	1	Large
Lucene	$1.32e^{-15}$	1	Large	$1.32e^{-15}$	1	Large
PostgreSQL	$1.32e^{-15}$	1	Large	$1.32e^{-15}$	1	Large
Xorg	$1.32e^{-15}$	0.96	Large	$1.32e^{-15}$	1	Large

TABLE VII
P-VALUE AND CLIFF'S DELTA OF COMPARING CPCC+ RESULTS WITH THOSE OF CPCC

Projects	F1-score			PofB20		
	p-value	Cliff's Delta	Effectiveness Level	p-value	Cliff's Delta	Effectiveness Level
Eclipse	1	0.11	Negligible	$1.32e^{-15}$	1	Large
Jackrabbit	$1.32e^{-15}$	0.85	Large	$1.32e^{-15}$	1	Large
Linux	$1.50e^{-6}$	0.31	Small	$1.32e^{-15}$	1	Large
Lucene	$1.32e^{-15}$	0.75	Large	$1.32e^{-15}$	1	Large
PostgreSQL	$1.32e^{-15}$	1	Large	$1.32e^{-15}$	1	Large
Xorg	0.2852	0.15	Negligible	$1.32e^{-15}$	1	Large

TABLE VIII
P-VALUE AND CLIFF'S DELTA OF COMPARING CPCC RESULTS WITH THOSE OF PCC+

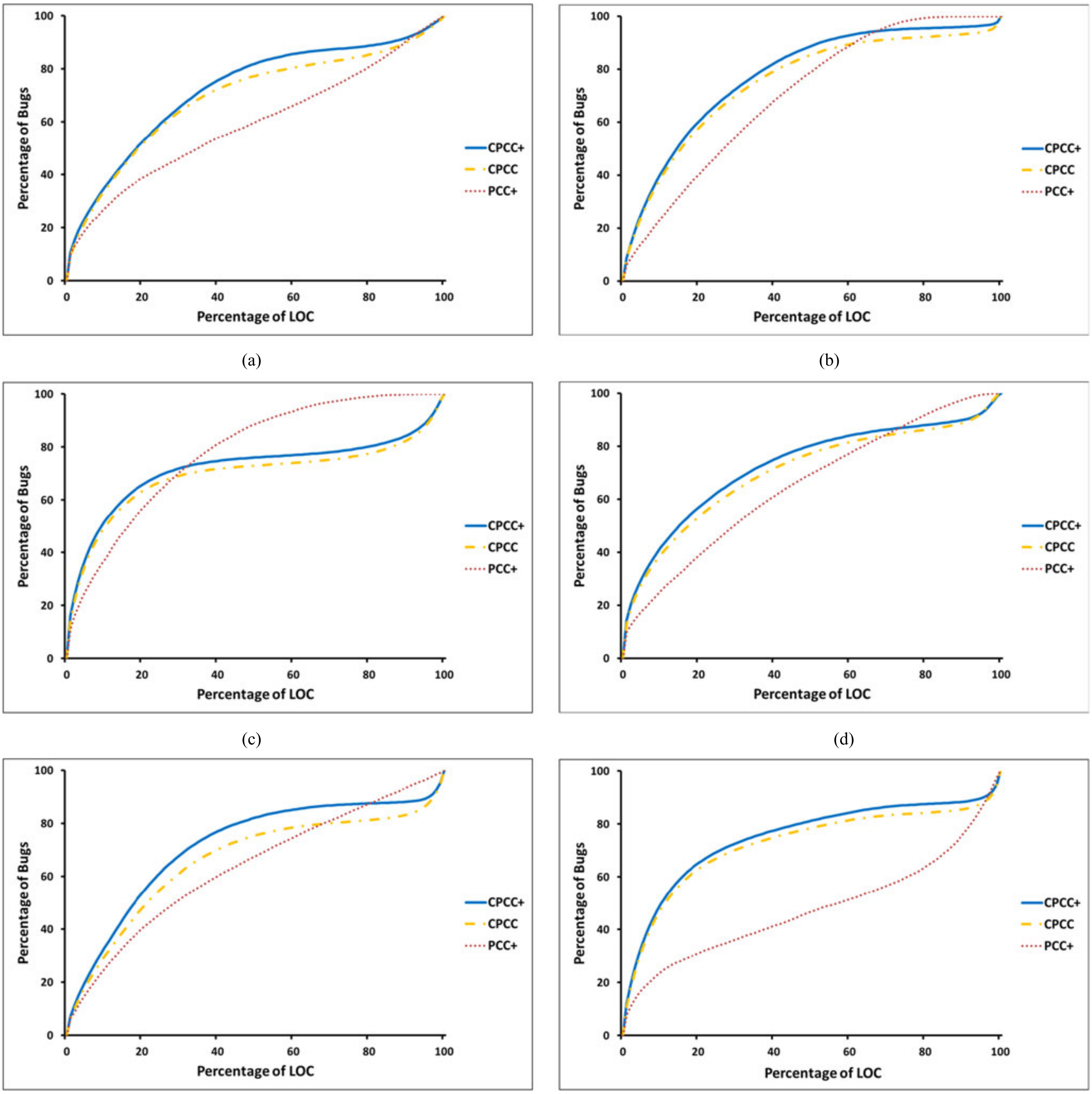
Projects	F1-score			PofB20		
	p-value	Cliff's Delta	Effectiveness Level	p-value	Cliff's Delta	Effectiveness Level
Eclipse	$2.15e^{-6}$	0.39	Medium	$1.32e^{-15}$	1	Large
Jackrabbit	$1.2e^{-14}$	0.78	Large	$1.32e^{-15}$	1	Large
Linux	$4.73e{-}13$	0.67	Large	$1.32e^{-15}$	1	Large
Lucene	$6.30e^{-6}$	0.40	Medium	$1.32e^{-15}$	1	Large
PostgreSQL	$1.32e^{-15}$	0.77	Large	$1.32e^{-15}$	1	Large
Xorg	$1.06e^{-9}$	0.55	Large	$1.32e^{-15}$	1	Large

TABLE IX
P-VALUE AND CLIFF'S DELTA OF COMPARING CPCC RESULTS WITH THOSE OF CC

Projects	F1-score			PofB20		
	p-value	Cliff's Delta	Effectiveness Level	p-value	Cliff's Delta	Effectiveness Level
Eclipse	$1.32e^{-15}$	1	Large	$1.32e^{-15}$	1	Large
Jackrabbit	$5.69e^{-4}$	0.30	Small	$1.32e^{-15}$	1	Large
Linux	$1.32e^{-15}$	1	Large	$1.32e^{-15}$	1	Large
Lucene	$1.32e^{-15}$	1	Large	$1.32e^{-15}$	1	Large
PostgreSQL	$1.32e^{-15}$	1	Large	$1.32e^{-15}$	1	Large
Xorg	$1.32e^{-15}$	0.96	Large	$1.32e^{-15}$	1	Large

the best performance are wide, i.e., from around 1% to 80%. For Jackrabbit and Lucene, the percentage range for which CPCC+ achieves the best performance are from around 1% to 70%. For Linux, the percentage range for which CPCC+ achieves the best performance is narrower, i.e., from around 1% to 34%.

Our proposed approaches (i.e., CPCC and CPCC+) and PCC+ show similar performance in the lower percentages of LOC to inspect (i.e., less than 5%), and PCC+ performs better than both CPCC+ and CPCC when inspecting more than 95% of the number of LOC. The reasons of these observations are: 1) Buggy changes that can be identified when inspecting very low percentages of LOCs (e.g., less than 5%) are likely to be easy cases that can be detected equally well by CPCC, CPCC+, and PCC+. 2) Also, we note that as we increase the amount of LOC inspected, the rate of performance improvement that CPCC and CPCC+ gain decreases. This is the case since both CPCC and



(a)

(b)

(c)

(d)

(e)

(f)

Fig. 4. Cost effectiveness graphs for (a) Eclipse JDT, (b) Jackrabbit, (c) Linux, (d) Lucene, (e) PostgreSQL, and (f) Xorg.

CPCC+ penalize large changes that cover many LOCs [see the denominators of (1) and (7)]. These large changes are likely to be listed last (if they have the same confidence scores as other small changes). To include these large changes into the list of successfully detected buggy changes, we need to spend much inspection budget. This causes the performance improvement of CPCC at the latter end of the curves to taper off. In practice, developers would not inspect more than 95% of the number of LOC due to limited project budget and tight project schedule, and both CPCC and CPCC+ perform better than PCC+ in a wide range of percentages of LOC to inspect.

Table X presents the area under PofB curve (AUC) values for CPCC+ and CPCC compared with PCC+. On average across the six projects, CPCC+ achieves AUC value to 0.74, which improves CPCC and PCC+ by 0.04 and 0.10, respectively. Moreover, CPCC+ shows better AUC values than PCC+ on five out of six projects. And in Linux, we notice that the AUC value for PCC+ (0.78) is better than CPCC+ (0.72). As shown in Fig. 4, in Linux, PCC+ achieves better performance than CPCC+ when developers inspect more than 35% of LOCs.

CPCC+ detect more defects than PCC+ for a wide range of percentages of LOC to inspect.

TABLE X
AUC SCORES FOR CPCC+ AND CPCC COMPARED TO THOSE OF PCC+

Projects	CPCC+	CPCC	PCC+
Eclipse	**0.72**	0.69	*0.59*
Jackrabbit	**0.78**	0.75	*0.70*
Linux	0.72	*0.69*	**0.78**
Lucene	**0.71**	0.66	*0.63*
PostgreSQL	**0.75**	0.72	*0.48*
Xorg	**0.73**	0.70	*0.65*
Average.	**0.74**	0.70	*0.64*

The highest and lowest AUC scores for each dataset are in bold and italic, respectively.

TABLE XI
F1 AND POFB20 SCORES FOR DIFFERENT UNDERLYING CLASSIFIERS

Project	Methods	F1-score			PofB20		
		AD.	NB	LR	AD.	NB	LR
Eclipse	CPCC+	**0.62**	**0.52**	**0.56**	**52%**	**49%**	**49%**
	CPCC	0.61	**0.52**	0.55	51%	45%	45%
	PCC+	0.60	0.47	0.54	39%	37%	35%
Jackrabbit	CPCC+	**0.75**	**0.71**	0.70	**60%**	**57%**	**53%**
	CPCC	0.74	0.70	**0.72**	58%	55%	53%
	PCC+	**0.75**	0.69	0.71	40%	31%	23%
Linux	CPCC+	**0.60**	**0.44**	**0.48**	**66%**	**65%**	**56%**
	CPCC	0.58	0.42	0.46	63%	59%	52%
	PCC+	0.56	0.43	**0.48**	57%	55%	54%
Lucene	CPCC+	**0.60**	**0.47**	0.52	**54%**	**49%**	**48%**
	CPCC	0.59	0.45	**0.56**	48%	43%	42%
	PCC+	0.58	0.44	0.53	40%	19%	22%
PostgreSQL	CPCC+	**0.66**	**0.57**	**0.60**	**65%**	**69%**	**73%**
	CPCC	0.64	**0.57**	0.58	63%	65%	69%
	PCC+	0.63	**0.57**	0.59	31%	40%	43%
Xorg	CPCC+	**0.66**	**0.59**	**0.63**	**57%**	**56%**	**57%**
	CPCC	0.65	0.59	0.60	53%	51%	50%
	PCC+	0.64	0.54	0.59	38%	34%	31%

AD = ADTree, NB = Naive Bayes, LR = Logistic Regression. The best results for each underlying classifier are highlighted in bold.

TABLE XII
MODEL BUILDING TIME AND PREDICTION TIME FOR CPCC+, CPCC, AND PCC+ (IN SECONDS)

Projects	Model Building			Prediction		
	CPCC+	CPCC	PCC+	CPCC+	CPCC	PCC+
Eclipse	4.552	4.107	9.390	0.013	0.011	0.012
Jackrabbit	3.541	3.143	6.448	0.013	0.011	0.010
Linux	5.296	5.130	13.287	0.011	0.010	0.021
Lucene	3.241	3.097	5.817	0.009	0.009	0.008
PostgreSQL	6.038	5.908	12.598	0.009	0.010	0.008
Xorg	4.026	3.817	9.457	0.009	0.010	0.008
Average.	**4.449**	**4.200**	**9.500**	**0.011**	**0.010**	**0.011**

C. RQ3: How Effective Are CPCC+, CPCC, and PCC+ When Different Underlying Classifiers Are Used?

Motivation: By default, we set the underlying classifier of CPCC+, CPCC, and PCC+ as ADTree. In this RQ, we investigate the performance of CPCC+, CPCC, and PCC+ with other underlying classifiers. Answer to this research question can verify whether CPCC+ and CPCC still improve PCC+ for different underlying classifiers.

Approach: To answer this research question, we compare the performance of CPCC and PCC+ with two other underlying classifiers, i.e., Naive Bayes and Logistic Regression [36]. These two underlying classifiers are also widely used in the software engineering literature [1], [7], [8], [16], [17], [42], [43]. We also apply Wilcoxon signed-rank test [40] on the paired data to test whether the improvements of CPCC+ over CPCC and PCC+ with different underlying classifiers are significant. We also use Bonferroni correction [41] to counteract the effect of multiple comparisons.

Results: Table XI presents the experiment results of CPCC+, CPCC, and PCC+ with ADTree, Naive Bayes, and Logistic Regression. The results demonstrate that CPCC+ outperforms CPCC and PCC+ for each of the three underlying classifiers. In other words, the benefits of CPCC+ are not limited to a specific underlying classifier. Bonferroni correction test over the multiple p-values shows the improvements of CPCC+ over CPCC and PCC+ using Naive Bayes and Logistic Regression, as underlying classifiers are statistically significant at 95% confidence level in terms of F1-score and PofB20.

Also, comparing across the three underlying classifiers, we find that CPCC+ using ADTree as the underlying classifier achieves the best performance. We also investigate whether the improvements of CPCC+ using ADTree as the underlying classifier are significant over the other variants of CPCC+, CPCC, and PCC+ (e.g., CPCC+ using Naive Bayes as the underlying classifier, CPCC using Logistic Regression as underlying classifier), and Bonferroni correction test over the multiple p-values shows that the improvements of CPCC+ using ADTree as the underlying classifier over other variants of CPCC+, CPCC, and PCC+ are statistically significant at 95% confidence level in terms of F1-score and PofB20.

CPCC+ outperforms PCC+ in ADTree, Naive Bayes, and Logistic Regression. Also, CPCC+ with ADTree as the underly classifier achieves the best performance.

D. RQ4: How Much Time Does It Take for CPCC+ and CPCC to Run?

Motivation: The efficiency of CPCC+ and CPCC would affect its practical usage. Thus, in this research question, we investigate the time efficiency of CPCC+ and CPCC.

Motivation: To address this research question, we report the model building and prediction time. Model building time refers to the time it takes to convert a training data into a CPCC+ or CPCC classifier. Prediction time refers to the time it takes for a CPCC+ or CPCC classifier to predict the label of a change. We compare the model building and prediction time of CPCC+ and CPCC with those of PCC+.

Results: Table XII presents the model building and prediction time for each of the six projects. We notice that the model building and prediction time of CPCC+ and CPCC are reasonable, e.g., on average, we need about 4.449 and 4.200 s to train a model and 0.011 and 0.010 s to predict the label of an instance using the model. Notice that the model does not need to be updated all the time. A trained model can be used to label many changes.

The training time of CPCC+ is shorter than that of PCC+. The prediction time of CPCC+ is almost the same as PCC+.

From Table VII, we notice that the model building time for CPCC+ is less than PCC+. CPCC+ combines CPCC and PCC, while PCC+ combines PCC and CC (i.e., a global model built on all of the changes in the training set). The difference of the model building time of CPCC+ and PCC+ must then be caused by the difference in the model building time of CPCC and CC.

The average model building time for CC across the six projects is 9.186 s, which is twice the model building time for CPCC (i.e., 4.200 s). In CC, a global ADTree classifier is built by analyzing all of the changes in the training set. In CPCC, for each developer, we build an ADTree classifier by analyzing only the developer's own changes in the training set. The model building time for constructing multiple ADTree classifiers built on subsets of the training set (which are of small sizes) is less than the time needed to build an ADTree classifier from all changes in the training set.

The model building and prediction time for CPCC+ and CPCC are reasonable. Also, the model building time for CPCC+ and CPCC are less than that of PCC+.

VIII. Discussion

A. Precision Versus Recall

Our approach is meant to be a recommendation tool. The goal is to highlight defect-prone changes to improve the quality of source code. For such setting, recall (its ability to find defect-prone changes) is more important than precision. In terms of precision, for three out of the six datasets, our CPCC+ does not perform as well as PCC+ proposed by Jiang, Tan, and Kim; however, in terms of recall, CPCC+ perform either better than (for five datasets) or as good as (for one dataset) PCC+. Neither our approach nor PCC+ is ready for full automation yet (i.e., automatic defect-prone change prediction without human intervention) since the precisions of these approaches are still relatively low.

CPCC+ takes the maximum adjusted confidence scores outputted by CPCC and PCC and use it as a final score. With this optimistic heuristic, more changes will be labeled as buggy. This strategy sacrifices a little precision to gain more recall. The sacrifice in precision is substantially less than the gain in recall, i.e., CPCC+ average precision is 3.24% lower than that of PCC+, and CPCC+ average recall is 8.44% higher than that of PCC+.

Since in our study, recall is more important than precision. In this section, we also consider other evaluation metrics such as F2 and F4 scores. We define the F_β score as

$$F_\beta = (1+\beta^2) \times \frac{\text{precision} \times \text{recall}}{\beta^2 \times \text{precision} + \text{recall}}. \quad (10)$$

We set $\beta = 2$ and 4 to assign more weights to recall. Table XIII presents the F2 and F4 scores for CPCC+ and CPCC compared with PCC+. We notice on average across the six projects, CPCC+ achieves the F2 and F4 scores of 0.73 and 0.78, respectively, which outperforms CPCC by 0.04 and 0.07, and PCC+ by 0.04 and 0.06 in terms of F2 and F4 scores, respectively.

TABLE XIII
F2 and F4 Scores for CPCC+ and CPCC Compared With PCC+

Projects	Methods	F2	F4
Eclipse	CPCC+	**0.70**	**0.74**
	CPCC	0.64	0.65
	PCC+	*0.68*	*0.72*
Jackrabbit	CPCC+	**0.84**	**0.89**
	CPCC	0.79	0.82
	PCC+	*0.84*	*0.89*
Linux	CPCC+	**0.66**	**0.70**
	CPCC	0.61	0.63
	PCC+	*0.64*	*0.68*
Lucene	CPCC+	**0.70**	**0.76**
	CPCC	*0.64*	*0.67*
	PCC+	0.66	0.69
PostgreSQL	CPCC+	**0.74**	**0.80**
	CPCC	*0.72*	*0.75*
	PCC+	*0.72*	0.76
Xorg	CPCC+	**0.72**	**0.77**
	CPCC	0.71	0.74
	PCC+	*0.58*	*0.57*
Average.	CPCC+	**0.73**	**0.78**
	CPCC	*0.69*	*0.71*
	PCC+	*0.69*	0.72

The highest and lowest results for each dataset are in bold and italic, respectively.

TABLE XIV
F1 and PofB20 Scores of CPCC+ Compared With Those of PCC+ for the 50-Fold Cross-Validation Setting

Projects	Methods	F1-score	PofB20
Eclipse	CPCC+	0.62±0.01	52%±1%
	PCC+	0.60±0.02	39%±3%
Jackrabbit	CPCC+	0.74±0.01	59%±1%
	PCC+	0.73±0.01	41%±0%
Linux	CPCC+	0.59±0.01	67%±1%
	PCC+	0.55±0.02	57%±3%
Lucene	CPCC+	0.60±0.01	55%±1%
	PCC+	0.58±0.01	41%±2%
PostgreSQL	CPCC+	0.66±0.02	66%±1%
	PCC+	0.63±0.01	31%±1%
Xorg	CPCC+	0.66±0.00	56%±1%
	PCC+	0.57±0.02	37%±2%
Average.	CPCC+	0.65	59%
	PCC+	0.61	41%

B. Fifty-Fold Cross Validation

In the previous section, we use tenfold cross validation to evaluate the performance of CPCC+. Here, we also investigate the performance of CPCC+ with 50-fold cross validation. Table XIV presents the F1 and PofB20 scores of CPCC+ compared with those of PCC+ following the 50-fold cross-validation setting. We notice that the F1 and PofB20 scores of CPCC+ and PCC+ in the 50-fold cross-validation setting are almost the same as the corresponding scores in the tenfold cross-validation setting. Table XV presents the p-values and Cliff's delta of comparing CPCC+ results and those of PCC+ for the 50-fold cross-validation setting. Notice that in our study, we use Bonferroni correction to counteract the results of multiple comparisons, and thus, the p-values are adjusted. From the results, we notice

TABLE XV
P-VALUES AND CLIFF'S DELTA OF COMPARING CPCC+ RESULTS WITH THOSE OF PCC+ FOR THE 50-FOLD CROSS-VALIDATION SETTING

Projects	F1-score			PofB20		
	p-value	Cliff's Delta	Effectiveness Level	p-value	Cliff's Delta	Effectiveness Level
Eclipse	$3.01e^{-5}$	0.39	Medium	$1.32e^{-15}$	1	Large
Jackrabbit	$1.32e^{-15}$	0.99	Large	$1.32e^{-15}$	1	Large
Linux	$5.68e^{-14}$	0.71	Large	$1.32e^{-15}$	0.82	Large
Lucene	$2.38e^{-14}$	0.72	Large	$1.32e^{-15}$	1	Large
PostgreSQL	$1.32e^{-15}$	1	Large	$1.32e^{-15}$	1	Large
Xorg	$4.12e^{-10}$	0.64	Large	$1.32e^{-15}$	1	Large

TABLE XVI
F1 AND POFB20 SCORES OF CPCC+ COMPARED WITH THOSE OF CPCC+$^{\mathrm{single}}$

Projects	Methods	F1-score	PofB20
Eclipse	CPCC+	0.62±0.01	52%±0%
	CPCC+$^{\mathrm{single}}$	0.61±0.01	50%±2%
Jackrabbit	CPCC+	0.75±0.00	60%±0%
	CPCC+$^{\mathrm{single}}$	0.72±0.02	56%±2%
Linux	CPCC+	0.60±0.01	66%±1%
	CPCC+$^{\mathrm{single}}$	0.58±0.02	64%±2%
Lucene	CPCC+	0.60±0.01	54%±1%
	CPCC+$^{\mathrm{single}}$	0.60±0.01	53%±1%
PostgreSQL	CPCC+	0.66±0.01	65%±1%
	CPCC+$^{\mathrm{single}}$	0.65±0.01	61%±1%
Xorg	CPCC+	0.66±0.00	57%±1%
	CPCC+$^{\mathrm{single}}$	0.64±0.01	55%±1%
Average.	CPCC+	0.65	59%
	CPCC+$^{\mathrm{single}}$	0.63	57%

that the improvement of CPCC+ over PCC+ is statistically significant, and in most of the cases, the effect size is large.

C. Multiobjective Versus Single Objective

Notice that in our CPCC+, we use a multiobjective GA (e.g., NSGA II) to tune the parameters, and we select the parameters from the set of Pareto optimal solutions. In this section, we also investigate the performance of CPCC+ by using a single-objective GA (i.e., a simple GA [22], [23]) to tune the parameters. The objective function for the single-objective GA is max $F1(\text{par}) \times \text{cost}(\text{Par})$, and we use the same settings (i.e., population size, number of generations, crossover operator, and mutation operator) as the settings of the multiobjective GA. We denote CPCC+ with single-objective GA as CPCC+$^{\mathrm{single}}$.

Table XVI presents the F1 and PofB20 scores of CPCC+ compared with those of CPCC+$^{\mathrm{single}}$. On average across the six projects, CPCC+$^{\mathrm{single}}$ achieves an F1 and PofB20 score of 0.63 and 57%, which are lower than those of CPCC+. Table XVII presents the p-value and Cliff's delta of comparing CPCC+ and CPCC+$^{\mathrm{single}}$. Notice the p-values are adjusted by using Bonferroni correction. In terms of F1-score, we notice that CPCC+ statistically significantly outperforms CPCC+$^{\mathrm{single}}$ on five out of the six projects, and the effect sizes are medium or large on five out of the six projects too. And in terms of PofB20 score, we notice that CPCC+ statistically significantly outperforms CPCC+$^{\mathrm{single}}$ on all the six projects, and the effect sizes are medium or large on all the six projects too.

D. Longitudinal Data Setup

To investigate whether our tool can be used to solve the problem in the same setting as the one in practice, we performed an experiment using a longitudinal data setup [44], [45]. We first sorted the changes in the order they are submitted and split them into 11 nonoverlapping windows of equal sizes. The process then proceeds as follows: First, in fold 1, we train using changes in window 0 and test the trained model using the changes in window 1. Then, in fold 2, we train using changes in windows 0 and 1 and test the trained model using the changes in window 2, and so on. We proceed in a similar manner for the next folds. In the final fold (i.e., fold 10), we train using changes in windows 0–9 and test using the changes in window 10. We record the average performance across the ten folds.

Table XVIII presents the F1 and PofB20 scores for CPCC+, CPCC, and PCC+ for the longitudinal data setup. On average, across the six projects, CPCC+ achieves an F1 and PofB20 score of 0.65 and 59% respectively, which improves CPCC and PCC+ by 0.02 and 0.05 in terms of F1-score, respectively, and 3% and 19% in terms of PofB20, respectively. Moreover, we find that the F1 and PofB20 scores of CPCC+ for the longitudinal data setup are similar to the corresponding scores for the tenfold cross-validation setup.

E. Threats to Validity

Threats to internal validity relates to errors in our code and experiment bias. We have double checked our code, still there could be errors that we did not notice. To reduce training set selection bias for the multiobjective GA vNSGA-II, we run tenfold cross validation 100 times and record the average performance. Also, we have only tried one set of parameter settings for the GA; it is unclear whether different parameter settings would impact the performance of our approach.

Another threats to internal validity relates to the data quality. In this paper, we rely on developers commit logs to identify bug-introducing changes. However, in practice, developers may not always write the bug identifier into the commit message [46], [47]. Thus, some buggy changes may get lost due to the missed links (false negatives). This threat is faced by many other studies that rely on commit logs to identify bug fixing commits [38], [48], [49].

Threats to external validity relates to the generalizability of our results. We have analyzed 5000 changes from six projects.

TABLE XVII
P-Value and Cliff's Delta of Comparing CPCC+ Results With Those of CPCC+$^{\text{single}}$

Projects	F1-score			PofB20		
	p-value	Cliff's Delta	Effectiveness Level	p-value	Cliff's Delta	Effectiveness Level
Eclipse	0.0001	0.38	Medium	$1.33e^{-10}$	0.61	Large
Jackrabbit	$1.32e^{-15}$	0.90	Large	$1.32e^{-15}$	1	Large
Linux	$5.12e^{-8}$	0.68	Large	$1.21e^{-6}$	0.71	Large
Lucene	0.1531	0.08	Negligible	0.0021	0.33	Medium
PostgreSQL	0.0005	0.36	Medium	$1.32e^{-15}$	1	Large
Xorg	$3.52e^{-5}$	0.51	Large	$1.26e^{-7}$	0.42	Medium

TABLE XVIII
F1 and PofB20 Scores for CPCC+, CPCC, and PCC+
in Longitudinal Data Setup

Projects	Methods	F1	PofB20
Eclipse	CPCC+	**0.61**	**51%**
	CPCC	0.59	49%
	PCC+	*0.55*	*37%*
Jackrabbit	CPCC+	**0.76**	**61%**
	CPCC	0.74	57%
	PCC+	*0.70*	*38%*
Linux	CPCC+	**0.59**	**64%**
	CPCC	*0.57*	62%
	PCC+	*0.57*	*58%*
Lucene	CPCC+	**0.62**	**56%**
	CPCC	0.61	52%
	PCC+	*0.55*	*38%*
PostgreSQL	CPCC+	**0.64**	**63%**
	CPCC	0.62	62%
	PCC+	*0.60*	*34%*
Xorg	CPCC+	**0.67**	**60%**
	CPCC	0.64	53%
	PCC+	*0.61*	*36%*
Average.	CPCC+	**0.65**	**59%**
	CPCC	0.63	56%
	PCC+	*0.60*	*40%*

The highest and lowest results for each dataset are in bold and italic, respectively.

These six projects include a wide variety of systems. For example, Eclipse is a famous Java IDE, Linux is a famous open-source operating system, and PostgreSQL is a popularly used database. Moreover, these six projects follow different development processes. For example, Linux and Eclipse use Bugzilla to manage their bug reports, while Lucene and Jackrabbit use JIRA to manage their bug reports. Moreover, Linux, Eclipse, Lucene, Jackrabbit, and PostgreSQL use Git as their version control systems, while Xorg use CVS as its version control system. In the future, we plan to reduce the threat to generalizability of our findings further by analyzing even more changes from additional software projects. Similar to other defect prediction studies, it is not possible to completely remove threats to external validity (which include threats to the generalizability of reported findings).

Threats to construct validity refers to the suitability of our evaluation measures. We use cost effectiveness and F1-score which are also used by past studies to evaluate the effectiveness of various CC and defect prediction techniques [1], [8]–[12], [16], [17], [42]. Moreover, in our study, we measure cost effectiveness by inspecting a certain percentage of the number of LOCs in changes, and we assume the effort that developers spend in inspecting one LOC is uniform. In practice, developers may check whether a change is buggy by not only inspecting the code in the change, but also the surrounding code. Furthermore, a one-line change in a complex function demands much more effort to inspect than a new ten-line initialization code. Unfortunately, quantifying the effort developers put in inspecting one particular line is very hard especially considering the size of the dataset that we consider.

IX. Related Work

In this section, we present related studies on CC, defect prediction, spectrum-based fault localization, and search-based software engineering.

A. *Change Classification*

There have been a number of studies on CC which analyze various metrics to build a global prediction model to identify buggy changes [1]–[6]. Kim, Whitehead, and Zhang propose the CC problem and use support vector machines to classify a change to be buggy or clean [1]. In a later work, Shivaji, Whitehead, Akella, and Kim apply feature selection algorithms to further improve the performance of Kim, Whitehead, and Zhang's work [2]. Ostrand, Weyuker, and Bell use negative binomial regression to build a prediction model using metadata and developer-specific features to predict buggy changes [5]. Lumpe, Vasa, Menzies, Rush, and Turhan investigate the effectiveness of activity-centric static code metrics [6]. Kamei *et al.* perform a large-scale empirical study on CC [50]. They choose 14 change measures and build logistic regression models to predict the buggy changes. In this paper, we collectively refer to these techniques as traditional CC techniques.

Jiang, Tan, and Kim propose PCC which constructs a prediction model for each developer [7]. They introduce three algorithms: PCC, weighted PCC, and PCC+. For each developer, PCC builds a separate prediction model based on the developer's historical data. Weighted PCC is similar as PCC except that the model is built from a training data that consist of the developer's historical data (50%) and other developers' historical data (50%). PCC+ is a metaalgorithm that selects either PCC, weighted PCC, or a traditional CC technique. Models built using PCC, weighted PCC, and CC each computes a confidence score for a change that represents the likelihood of the change to be buggy. PCC+ picks the model that outputs the highest confidence score. In this work, we propose a new PCC algorithm which outperforms PCC+.

B. *Defect Prediction*

There have been a number of studies on defect prediction which analyze different metrics such as code complexity, process metrics, and code locations to build prediction models to identify defective code elements (e.g., defective source code files) [9], [16], [17], [31], [42], [43], [51], [52], [52]–[55]. CC is closely related to and can be viewed as a special instance of defect prediction; defect prediction predicts defective code elements (e.g., files), while CC predicts defective changes.

Bettenburg, Nagappan, and Hassan [31] use an algorithm called MARS which is a global model that has local consideration to improve the performance of defect prediction. Jiang, Tan, and Kim have applied MARS to CC problem and show that PCC+ achieves a much better performance than MARS [7]. Rahman and Devanbu compare the effect of code metric and process metrics for defect prediction [9].

There are a number of defect prediction studies that employ ensemble learning [56], [57]. Seiffert, Khoshgoftaar, and Van Hulse apply five different data sampling approaches and ensemble learning technology to predict fault-prone modules in 15 projects [56]. Tosun, Turhan, and Bener use an ensemble of classifiers for defect prediction [57].

Recently, several studies investigate cross-project defect prediction, where defect data from other projects is used to improve defect prediction for a target project. Turhan, Menzies, Bener, and Di Stefano employ a k-nearest neighbor approach to select instances from various projects to be used as training data for a target project [52]. Nam, Pan, and Kim extend TCA which transforms data from two projects to a latent space where the two datasets are close to each other [16]. Liu, Khoshgoftaar, and Seliya propose a genetic programming-based approach (GP) which constructs a classification model in the form of a tree considering defect data from multiple software repositories [33]. Canfora *et al.* construct a classification model by using multiobjective GA for cross-project defect prediction [17]. Panichella, Oliveto, and De Lucia propose an approach named CODEP that uses a classification model to combine results of six classification algorithms (i.e., logistic regression, RBF network, multilayer perceptron, etc.) for cross-project defect prediction [55]. Turhan, Mısırlı, and Bener perform an empirical study on the effectiveness of the combination of within and cross (i.e., mixed) project data for binary defect prediction [58].

Our work is orthogonal to the above studies. In this paper, we mainly focus on the CC problem, which is a similar but a different problem from defect prediction. Moreover, Canfora *et al.* build the cross-project model based on multi-objective logistic regression model, and they merge all data from different projects to train the regression model. Our CPCC is technical different from their approach, we consider the target difference and source difference phenomenon and combine different prediction models by leveraging multiobjective GA. Also, the objective functions between CPCC and Canfora *et al.*'s model are different.

C. *Search-Based Software Engineering*

There have been a number of studies on search-based software engineering [59]–[62]. Harman and Jones propose the concept of search-based software engineering, and they demonstrate how to reformulate a SE problem as a search-based problem [59]. Later, Harman, Mansouri, and Zhang provide a review and classification of search-based software engineering techniques [60]. Recently, Bueno, Jino, and Wong use metaheuristic search algorithms (i.e., simulated annealing, GA, and simulated repulsion) to generate test cases that are diverse [63]. Panichella *et al.* propose a search-based GA which tunes latent Dirichlet allocation parameters and use the proposed algorithm for traceability link recovery, feature location, and software artifact labeling [61]. Le Goues, Nguyen, Forrest, and Weimer propose GenProg, which uses GA to automatically repair defects in software projects [62]. Wang, Harman, Jia, and Krinke propose a search-based approach for clone detection [64]. Lohar, Amornborvornwong, Zisman, and Cleland-Huang propose a search-based approach which identifies the best configuration for a trace retrieval technique that recovers traceability links between software artifacts (e.g., requirement to code, requirement to design, etc.) [65]. In this work, we also use a search-based technique to learn a semioptimal composition of classifiers. Different from the above-mentioned studies, we address a different problem namely PCC.

X. Conclusion and Future Work

In this paper, we propose a new PCC technique named CPCC. CPCC combine different personalized models trained from various developers' change data using multiobjective GA. CPCC first builds a separate prediction model for each developer based on his/her change data. Next, for each developer, CPCC combines these models by assigning different weights to the models with the purpose of maximizing two objective functions (i.e., F1-scores and cost effectiveness). To further improve classification performance, we propose CPCC+ which utilizes the advantage of CPCC and PCC. We perform experiments on 5000 changes from six software projects: Eclipse JDT, Jackrabbit, the Linux kernel, Lucene, PostgreSQL, and Xorg. The experiment results show that on average CPCC+ can discover 122 more bugs than PCC+ (371 versus 249) per project if developers inspect the top 20% LOC that are predicted buggy. In addition, CPCC+ can achieve F1-scores of 0.60–0.75, which are statistically significantly higher than those of PCC+ on all of the six projects.

In the future, we plan to evaluate CPCC+ and CPCC with datasets from more software projects and develop a better technique which could improve the prediction performance further.

Acknowledgment

The authors would like to thank L. Tan for providing them the datasets used to evaluate PCC+.

REFERENCES

[1] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, Mar./Apr. 2008.

[2] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," *IEEE Trans. Softw. Eng.*, vol. 39, no. 4, pp. 552–569, Apr. 2013.

[3] J. T. Madhavan and E. J. Whitehead Jr., "Predicting buggy changes inside an integrated development environment," in *Proc. OOPSLA Workshop Eclipse Technol. Exchange*, 2007, pp. 36–40.

[4] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proc. IEEE 33rd Int. Conf. Softw. Eng.*, 2011, pp. 481–490.

[5] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Programmer-based fault prediction," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng.*, 2010, p. 19.

[6] M. Lumpe, R. Vasa, T. Menzies, R. Rush, and B. Turhan, "Learning better inspection optimization policies," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 22, no. 5, pp. 621–644, 2012.

[7] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Proc. IEEE/ACM 28th Int. Conf. Autom. Softw. Eng.*, 2013, pp. 279–289.

[8] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the imprecision of cross-project defect prediction," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, p. 61.

[9] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 432–441.

[10] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu, "Sample size vs. bias in defect prediction," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, 2013, pp. 147–157.

[11] E. Arisholm, L. C. Briand, and M. Fuglerud, "Data mining techniques for building fault-proneness models in telecom java software," in *Proc. 18th IEEE Int. Symp. Softw. Rel.*, 2007, pp. 215–224.

[12] E. Arisholm and L. C. Briand, "Predicting fault-prone components in a java legacy system," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng.*, 2006, pp. 8–17.

[13] 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.*, to be published.

[14] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proc. IEEE Int. Conf. Softw. Quality Rel. Security*, 2015, pp. 17–26.

[15] Y. Zhang, D. Lo, X. Xia, and J. Sun, "An empirical study of classifier combination for cross-project defect prediction," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, 2015, vol. 2, pp. 264–269.

[16] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 382–391.

[17] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Multi-objective cross-project defect prediction," in *Proc. IEEE 6th Int. Conf. Softw. Testing Verification Validation*, 2013, pp. 252–261.

[18] J. Eyolfson, L. Tan, and P. Lam, "Correlations between bugginess and time-based commit characteristics," *Empirical Softw. Eng.*, vol. 19, pp. 1009–1039, 2014.

[19] Y. Freund and L. Mason, "The alternating decision tree learning algorithm," in *Proc. 16th Int. Conf. Mach. Learning*, 1999, vol. 99, pp. 124–133.

[20] K. Deb *et al.*, *Multi-Objective Optimization Using Evolutionary Algorithms*, vol. 2012. Hoboken, NJ, USA: Wiley, 2001.

[21] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[22] S. Sivanandam and S. Deepa, *Introduction to Genetic Algorithms*. Berlin, Germany: Springer, 2007.

[23] D. E. Goldberg and J. H. Holland, "Genetic algorithms and machine learning," *Mach. Learning*, vol. 3, no. 2, pp. 95–99, 1988.

[24] K. Deb and R. B. Agrawal, "Simulated binary crossover for continuous search space," *Complex Syst.*, vol. 9. pp. 115–148, 1994.

[25] K. Deb and M. Goyal, "A combined genetic adaptive search (geneas) for engineering design," *Comput. Sci. Informat.*, vol. 26, pp. 30–45, 1996.

[26] Z. Li, H. Liao, and D. W. Coit, "A two-stage approach for multi-objective decision making with applications to system reliability optimization," *Rel. Eng. Syst. Safety*, vol. 94, no. 10, pp. 1585–1592, 2009.

[27] Z. Li, M. Mobin, and T. Keyser, "Multi-objective and multi-stage reliability growth planning in early product-development stage," *IEEE Trans. Rel.*, vol. 65, no. 2, pp. 769–781, Jun. 2015.

[28] M. Tavana, Z. Li, M. Mobin, M. Komaki, and E. Teymourian, "Multi-objective control chart design optimization using NSGA-III and MOPSO enhanced with DEA and TOPSIS," *Expert Syst. Appl.*, vol. 50, pp. 17–39, 2016.

[29] X. Xia, Y. Feng, D. Lo, Z. Chen, and X. Wang, "Towards more accurate multi-label software behavior learning," in *Proc. Softw. Evolution Week-IEEE Conf. Softw. Maintenance, Reeng. Reverse Eng.*, 2014, pp. 134–143.

[30] X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," in *Proc. 10th Int. Working Conf. Mining Softw. Repositories*, 2013, pp. 287–296.

[31] N. Bettenburg, M. Nagappan, and A. E. Hassan, "Think locally, act globally: Improving defect and effort prediction models," in *Proc. 9th IEEE Working Conf. Mining Softw. Repositories*, 2012, pp. 60–69.

[32] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2008, pp. 346–355.

[33] Y. Liu, T. M. Khoshgoftaar, and N. Seliya, "Evolutionary optimization of software quality modeling with multiple repositories," *IEEE Trans. Softw. Eng.*, vol. 36, no. 6, pp. 852–864, Nov./Dec. 2010.

[34] Q. Zhang and H. Li, "MOEA/D: A multiobjective evolutionary algorithm based on decomposition," *IEEE Trans. Evol. Comput.*, vol. 11, no. 6, pp. 712–731, Dec. 2007.

[35] 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.

[36] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*. San Mateo, CA, USA: Morgan Kaufmann, 2006.

[37] Y. Tian, J. Lawall, and D. Lo, "Identifying Linux bug fixing patches," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 386–396.

[38] 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. Found. Softw. Eng.*, 2011, pp. 15–25.

[39] N. Cliff, *Ordinal Methods for Behavioral Data Analysis*. Abingdon, U.K.: Psychology Press, 2014.

[40] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics*, vol. 1, no. 6, pp. 80–83, 1945.

[41] H. Abdi, "Bonferroni and Šidák corrections for multiple comparisons," in *Encyclopedia of Measurement and Statistics*, N. J. Salkind, Ed., 2007. [Online]. Available: http://www.utdallas.edu/˜herve/abdi-bonferroni2007-pretty.pdf

[42] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in *Proc. 10th IEEE Working Conf. Mining Softw. Repositories*, 2013, pp. 409–418.

[43] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proc. 7th Joint Meet. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 91–100.

[44] X. Xia, D. Lo, E. Shihab, X. Wang, and B. Zhou, "Automatic, high accuracy prediction of reopened bugs," *Autom. Softw. Eng.*, vol. 22, no. 1, pp. 75–109, 2014.

[45] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *Proc. 19th ACM SIGSOFT Symp./13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 365–375.

[46] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead Jr., "Automatic identification of bug-introducing changes," in *Proc. 21st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2006, pp. 81–90.

[47] C. Williams and J. Spacco, "SZZ revisited: Verifying when changes induce fixes," in *Proc. Workshop Defects Large Softw. Syst.*, 2008, pp. 32–36.

[48] T. F. Bissyandé, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Reveillere, "Empirical evaluation of bug linking," in *Proc. IEEE 17th Eur. Conf. Softw. Maintenance Reeng.*, 2013, pp. 89–98.

[49] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Multi-layered approach for recovering links between bug reports and fixes," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, p. 63.

[50] 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.

[51] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 531–540.

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

[53] 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.

[54] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proc. 7th IEEE Working Conf. Mining Softw. Repositories*, 2010, pp. 31–41.

[55] A. Panichella, R. Oliveto, and A. De Lucia, "Cross-project defect prediction models: L'Union fait la force," in *Proc. IEEE Conf. Softw. Evolution Week Softw. Maintenance, Reeng. Reverse Eng.*, 2014, pp. 164–173.

[56] C. Seiffert, T. M. Khoshgoftaar, and J. Van Hulse, "Improving software-quality predictions with data sampling and boosting," *IEEE Trans. Syst., Man Cybern. A, Syst. Humans*, vol. 39, no. 6, pp. 1283–1294, Nov. 2009.

[57] A. Tosun, B. Turhan, and A. Bener, "Ensemble of software defect predictors: A case study," in *Proc. 2nd ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2008, pp. 318–320.

[58] B. Turhan, A. T. Mısırlı, and A. Bener, "Empirical evaluation of the effects of mixed project data on learning defect predictors," *Inf. Softw. Technol.*, vol. 55, no. 6, pp. 1101–1118, 2013.

[59] M. Harman and B. F. Jones, "Search-based software engineering," *Inf. Softw. Technol.*, vol. 43, no. 14, pp. 833–839, 2001.

[60] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surveys*, vol. 45, no. 1, p. 11, 2012.

[61] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 522–531.

[62] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan./Feb. 2012.

[63] P. M. S. Bueno, M. Jino, and W. E. Wong, "Diversity oriented test data generation using metaheuristic search techniques," *Inf. Sci.*, vol. 259, pp. 490–509, 2014.

[64] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: a rigorous approach to clone evaluation," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, 2013, pp. 455–465.

[65] S. Lohar, S. Amornborvornwong, A. Zisman, and J. Cleland-Huang, "Improving trace accuracy through data-driven configuration and composition of tracing features," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, 2013, pp. 378–388.