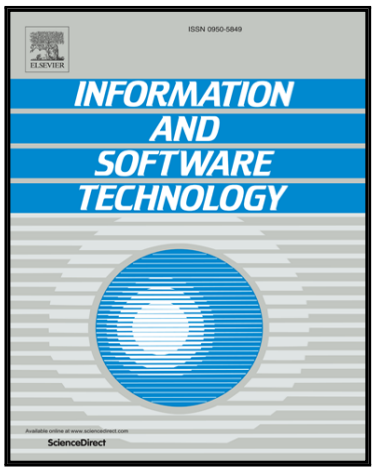



# MULTI: Multi-Objective Effort-Aware Just-in-Time Software Defect Prediction

Xiang Chen, Yingquan Zhao, Qiuping Wang, Zhidan Yuan

PII: S0950-5849(17)30462-7
DOI: 10.1016/j.infsof.2017.08.004
Reference: INFSOF 5862

To appear in: *Information and Software Technology*

Received date: 19 April 2017
Revised date: 22 July 2017
Accepted date: 10 August 2017

Please cite this article as: Xiang Chen, Yingquan Zhao, Qiuping Wang, Zhidan Yuan, MULTI: Multi-Objective Effort-Aware Just-in-Time Software Defect Prediction, *Information and Software Technology* (2017), doi: 10.1016/j.infsof.2017.08.004

# MULTI: Multi-Objective Effort-Aware Just-in-Time Software Defect Prediction

Xiang Chen[a,b,c,*], Yingquan Zhao[a], Qiuping Wang[a], Zhidan Yuan[a]

[a] *School of Computer Science and Technology, Nantong University, Nantong, China*
[b] *Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin, China*
[c] *State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China*

**Abstract**

*Context:* Just-in-time software defect prediction (JIT-SDP) aims to conduct defect prediction on code changes, which have finer granularity. A recent study by Yang et al. has shown that there exist some unsupervised methods, which are comparative to supervised methods in effort-aware JIT-SDP.

*Objective:* However, we still believe that supervised methods should have better prediction performance since they effectively utilize the gathered defect prediction datasets. Therefore we want to design a new supervised method for JIT-SDP with better performance.

*Method:* In this article, we propose a multi-objective optimization based supervised method MULTI to build JIT-SDP models. In particular, we formalize JIT-SDP as a multi-objective optimization problem. One objective is designed to maximize the number of identified buggy changes and another object is designed to minimize the efforts in software quality assurance activities. There exists an obvious conflict between these two objectives. **MULTI uses logistic regression to build the models and uses NSGA-II to generate a set of non-dominated solutions**, which each solution denotes the coefficient vector for the logistic regression.

*Results:* We design and conduct a large-scale empirical studies to compare

*Corresponding author
*Email addresses:* xchencs@ntu.edu.cn (Xiang Chen), enockchao@163.com (Yingquan Zhao), 724350252@qq.com (Qiuping Wang), 3022868066@qq.com (Zhidan Yuan)

MULTI with 43 state-of-the-art supervised and unsupervised methods under the three commonly used performance evaluation scenarios: cross-validation, cross-project-validation, and timewise-cross-validation. Based on six open-source projects with 227417 changes in total, our experimental results show that MULTI can perform significantly better than all of the state-of-the-art methods when considering $ACC$ and $P_{OPT}$ performance metrics.

*Conclusion:* By using multi-objective optimization, MULTI can perform significantly better than the state-of-the-art supervised and unsupervised methods in the three performance evaluation scenarios. The results confirm that supervised methods are still promising in effort-aware JIT-SDP.

*Keywords:* Just-in-time Defect Prediction, Multi-objective Optimization, Empirical Studies, Search based Software Engineering

## 1. Introduction

Software defect prediction (SDP) [1, 2, 3] is an active research topic in current software engineering research domain. In particular, after mining software repositories (such as version control system, bug tracking system, or developer emails), we can extract numerous program modules. Then we can use metrics to measure these modules and label these modules based on the analysis of bug reports and change logs. These metrics [4] are mainly designed based on the analysis of code complexity or code development process. Finally we can construct SDP models based on these gathered datasets and use the constructed models to predict potential defective modules. By identifying potentially defective modules in advance, software quality assurance (SQA) team can allocate more SQA efforts (i.e., time for designing more test cases or conducting more rigorous code inspection) on these identified modules.

According to Kamei et al. [5], previous research on SDP exists the following disadvantages: (1) Predicting at the coarser granularity (such as package or file) of the program module needs more SQA efforts. (2) Once a module is predicted as defect-proness, it is difficult to find appropriate developers to localize and fix

these defects in this module. Especially for open source software, a module may be developed by multiple developers from all over the world. (3) Developers ideally want the prediction performed as early as possible so that they can still be familiar with their developed codes. To alleviate these disadvantages, researchers aimed to conduct SDP on code changes (i.e., commits to a version control system) and classified the code changes as buggy or clean. Kamei et al. [5] called this process as just-in-time software defect prediction (JIT-SDP).

Previous studies have proposed different supervised methods and unsupervised methods for JIT-SDP [6, 5, 7, 8, 9, 10]. Especially in a recent study, Yang et al. [8] found that some simple unsupervised methods can perform better than previous proposed supervised methods in most cases. This conclusion contradicts our common sense (i.e., supervised methods can exploit more knowledge from the gathered datasets and then have better performance when compared with unsupervised methods). **To investigate whether there is a way to improve the performance of supervised methods**, we firstly apply multi-objective optimization algorithm (MOA) to JIT-SDP. This attempt is motivated by the idea of search based software engineering (SBSE) [11]. Harman [11] once suggested that SBSE (in particular MOA) can be potentially used to build SDP models. In practice, he suggested that these objectives can be set as prediction performance, construction cost, privacy or comprehensibility of the model.

For this problem, we mainly consider two optimization objectives. In particular, the first objective is designed from the view of benefit, it aims to maximize the number of identified buggy changes. The second objective is designed from the view of cost, it aims to minimize the efforts of SQA. It is not hard to find that there is an obvious conflict between these two objectives during the model construction phase. Generally speaking, if we want the trained model to identify more buggy changes in the training set, it will need more SQA efforts. On the contrary, if we want the trained model to reduce SQA efforts, it will miss more buggy changes. Based on the above analysis, we propose our MULTI method based on NSGA-II [12], which is a state-of-the-art multi-objective optimization algorithm. In our method, we use logistic regression to build the model. By

using NSGA-II, we can generate a set of non-dominated solutions. A solution is composed of coefficient vector and each coefficient vector can be used to construct a specific JIT-SDP model by utilizing logistic regression. After generating multiple models simultaneously, we can select appropriate models based on a specific preference (i.e., identifying more buggy changes or using less SQA efforts).

In our empirical studies, we consider six open-source projects as our experimental subjects, which cover a wide range of application domains. These projects have 227417 code changes in total. We consider two effort-aware performance metrics (i.e., $ACC$ and $P_{OPT}$) and evaluate our method in three different model performance evaluation scenarios, such as cross-validation, cross-project-validation, and timewise-cross-validation. To verify the effectiveness of our proposed method, we consider 43 state-of-the-art baseline methods. In particular, 31 methods are supervised methods and the remaining 12 methods are unsupervised methods. Final results show that using $ACC$ performance metric or $P_{OPT}$ performance metric, our method can perform significantly better than all the baseline methods in all the three scenarios.

The main contributions of this article can be summarized as follows:

- To the best of our knowledge, we firstly apply multi-objective optimization to JIT-SDP and propose MULTI method. This method aims to maximize the number of identified buggy changes while minimizing the efforts of software quality assurance.
- We designed and performed large-scale empirical studies to investigate the performance of MULTI in three different model evaluation scenarios and compare our method with 43 state-of-the-art baseline methods. The conclusions confirm that the supervised methods are still promising in effort-ware JIT-SDP.

The rest of this article is organized as follows. Section 2 introduces the background and related work for JIT-SDP. Section 3 describes our proposed method

MULTI in detail. Section 4 reports our empirical setup, including experimental subjects, performance metrics, model performance evaluation scenarios, baseline methods and experimental design. Section 5 analyzes our empirical results and threats to validity. Section 6 concludes this article and points out some potential future work.

## 2. Background and Related Work

### *2.1. Background*

Previous research on software defect prediction may not be practical for large-scale software since module granularity is often set as file or package. Some researchers aim to conduct SDP on code changes, which have smaller granularity. During the process of software development and maintenance, developers may submit code changes for a variety of reasons, such as fixing defects, extending functionality, refactoring codes, or improving system performance. In this problem, they want to classify these changes into two categories: buggy changes and clean changes. Here a buggy change means that this change will introduce one or more defects, while a clean change means that this change will not introduce any defects.

The process of JIT-SDP can be summarized as follows: (1) It first extracts code changes from software repositories (such as version control system). (2) It measures these code changes based on code complexity, programmer experience, development process or text mining techniques [5, 7]. (3) It uses SZZ algorithm [13] to identify defect-fixing changes and then classifies the extracted code changes to buggy change and clean change. (3) Based on the gathered training dataset, it uses a specific modeling method (such as logistic regression) to construct JIT-SDP model. (4) For a new code change, it measures this change and uses a trained model to predict whether this change is clean or buggy.

### *2.2. Related Work*

Mockus and Weiss [6] firstly applied JIT-SDP to a large switching system software. They considered metrics based on the analysis of the diffusion and size

of a change, the type of change, and the developers' experience with the system. Kamei et al. [5] considered similar metrics for the code changes and conducted large-scale empirical studies on six open-source and five commercial projects. They found their models can achieve 68% accuracy and 64% recall on average. Moreover, they also proposed a novel method EALR, which considers efforts to test or inspect code changes. Later they [14] investigated JIT-SDP in cross-project scenarios and provided suggestions to improve the performance in this scenario. Yang et al. [9] proposed Deeper method, which applies deep learning to JIT-SDP. In particular, they constructed a set of new metrics from original metrics by leveraging deep belief network. Then they used logistic regression to build the model based on these new metrics. Later Yang et al also proposed a two-layer ensemble learning method TLEL which leverages decision tree and ensemble learning to improve the performance of JIT-SDP. Recently Yang et al. [8] considered simple unsupervised methods for JIT-SDP and found that these unsupervised methods perform better than previous supervised methods (including EALR) in most cases under cross-validation, timewise-cross-validation, and cross-project-validation scenarios. Fu and Menzies [15] revisited Yang et al.'s empirical results [8] and found that not all unsupervised methods have better performance than supervised methods. Therefore they proposed OneWay method, which can automatically select the potential best method.

Kim et al. [7] constructed JIT-SDP models from another point of view. They extracted features (i.e., metrics) from change log messages, source code and file names by using text mining methods. In their empirical studies on 12 open-source projects, they found their model can achieve 78% accuracy and 60% recall on average. Since using text mining methods will extract a large number of features, Shivaji et al. [16] investigated multiple feature selection methods to overcome the curse of dimensionality. Final results showed that these methods can select less than 10% of the original features and the performance can be significantly improved.

A recent study by Yang et al. [8] shows that some unsupervised methods can achieve better performance than supervised method. However, their finding

contradicts our common sense and we still believe that supervised methods should have better performance. In this article, we firstly apply multi-objective optimization to JIT-SDP and then propose a supervised method MULTI. To investigate the effectiveness of our proposed method, we conduct large-scale empirical studies on 6 open-source projects and compare MULTI with 43 state-of-the-art supervised and unsupervised methods. The final results show that supervised methods are still promising in effort-aware JIT-SDP. Software defect prediction is a relatively new application area of multi-objective optimization. Canfora et al. [17] once applied multi-objective optimization for cross-project defect prediction and proposed MODEP method. While in this article, we mainly applied multi-objective optimization for JIT-SDP, therefore the solved problem, selected experimental subjects, used performance metrics, and the model evaluation process are all different from their work [17].

## 3. Our Proposed Method MULTI

Our research is motivated by the idea of search based software engineering (SBSE) [18]. SBSE concept was first proposed by Mark Harman. It has become an active research topic in recent software engineering research. SBSE has been applied to many problems throughout the software life cycle (i.e., from requirement analysis, software design, software testing, to software maintenance). The approach is promising because it can provide automated or semi-automated solutions in complex problems with large-scale search spaces, which have multiple competing or even conflicting objectives. For SDP, Harman [11] also firstly suggested that SBSE (in particular MOA) can be potentially used to build SDP models. The optimization objectives can be designed based on the analysis for predictive quality, cost, or privacy.

In this article, we use logistic regression, which is widely used in previous SDP research [19, 20, 21, 22], to build JIT-SDP model. The coefficient vector $w =< w_0, \cdots, w_n >$ of the model can be denoted as a solution for JIT-SDP. Supposing there is $n$ metrics to measure the code change, we use $m_i$ to denote

the $i$-th code change and use $v_{i,j}$ to denote its value on $j$-th metric. For this code change, we can use function $y()$, which is the modeling formula used in logistic regression, to estimate the defect-proneness probability of this change and this function can be set as below:

$$y(m_i, w) = \frac{1}{1 + e^{-(w_0 + w_1 v_{i,1} + \cdots + w_n v_{i,n})}} \tag{1}$$

In our article, we treat JIT-SDP as a binary classification problem. However the range of the output value of function $y()$ is [0,1]. Therefore if the value of $y()$ is greater than 0.5, we classify the code change as buggy, otherwise we classify the code change as clean. The new function $Y()$ can be set as below:

$$Y(m_i, w) = \begin{cases} 1 & if\ y(m_i, w) > 0.5 \\ 0 & if\ y(m_i, w) \leq 0.5 \end{cases} \tag{2}$$

For effort-aware JIT-SDP, we mainly consider two optimization objectives based on cost-benefit analysis when constructing models. The first objective is designed from the view of benefit. Based on a set of changes $M$ and a solution $w$, it can be computed by the following formula:

$$benefit(w) = \sum_{m_i \in M} Y(m_i, w) \times buggy(m_i) \tag{3}$$

Here $buggy(m_i)$ denotes whether code change $m_i$ really has defects. If this code change has defects, its value is 1. Otherwise its value is 0.

The second objective is designed from the view of cost. Based on a set of changes $M$, it can be computed by the following formula:

$$cost(w) = \sum_{m_i \in M} Y(m_i, w) \times SQA(m_i) \tag{4}$$

Here we use function $SQA()$ to measure the SQA efforts on the change $m_i$. During the software development and maintenance phase, SQA efforts denotes the time allocated for designing more test cases or conducting more rigorous code inspection on potential faulty changes. In this article, this function is designed

to return the total number of LOC (lines of code) modified by a change as the SQA efforts which is suggested by Kamei et al. [5]. They assume that a change, which adds or deletes more lines, will require more efforts to test or inspect than the change, which adds or deletes fewer lines.

We use a synthesized example to illustrate the computing process of these two objectives. Supposing a set of changes $M$, their actual class (i.e., $buggy()$), their predicted class based on a specific JIT-SDP model (i.e., $Y()$), and their efforts (i.e., $SQA()$) are shown in Table 1, the benefit objective of this model can be computed by $(0 \times 1 + 1 \times 0 + \cdots + 1 \times 1)$ and the cost objective of this model can be computed by $(1 \times 30 + 0 \times 20 + \cdots + 1 \times 15)$.

Table 1: A Synthesized Example to Illustrate the Computing Process of Two Objectives

| Changes | Actual Class | Predicted Class | SQA Efforts |
|---|---|---|---|
| $c_1$ | 0 | 1 | 30 |
| $c_2$ | 1 | 0 | 20 |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $c_n$ | 1 | 1 | 15 |

Based on the above analysis, we can formalize JIT-SDP as a bi-objective optimization problem and then propose a novel method MULTI based on MOA. To facilitate the subsequent description of our proposed method, we first give some definitions concerning about MOA.

**Definition 1 (Pareto Dominance).** *Supposing $w_i$ and $w_j$ are two feasible solutions to construct JIT-SDP models, we call $w_i$ is Pareto dominance on $w_j$, if and only if: $benefit(w_i) > benefit(w_j)$ and $cost(w_i) \leq cost(w_j)$ or $benefit(w_i) \geq benefit(w_j)$ and $cost(w_i) < cost(w_j)$*

**Definition 2 (Pareto Optimal Solution).** *A feasible solution $w$ is a Pareto optimal solution, if and only if there is no other feasible solution $w^*$ which is Pareto dominance on $w$.*

**Definition 3 (Pareto Optimal Set).** *This set is composed by all the Pareto optimal solutions.*

**Definition 4 (Pareto Front).** *The surface composed by the vectors corresponding to all the Pareto optimal solutions is called Pareto front.*

We use an example to interpret these definitions. Supposing MULTI method generate 7 solutions given a training set and these solutions are shown in Fig. 1. In this Figure, $x$-axis denotes the cost value of the solution and $y$-axis denotes the benefit value of the solution. Here solution B is Pareto dominance on solution E, since $benefit(B) > benefit(E)$ and $cost(B) \leq cost(E)$. Solutions A, B, C and D are pareto optimal solutions since there is no other solution which is Pareto dominance on them. Therefore the surface A-B-C-D constitutes Pareto front in these solutions.

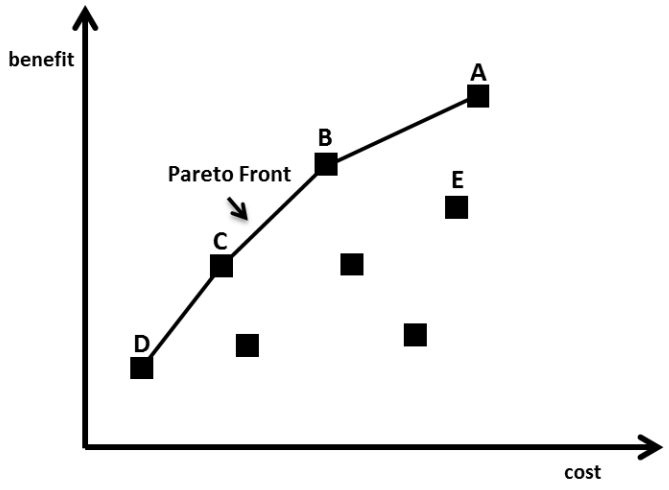



Figure 1: Interpretation for Definitions in MOA

Researchers have proposed many different types of MOAs. These algorithms use evolutionary algorithms to construct the Pareto optimal set. Our proposed MULTI is mainly designed based on NSGA-II[12], which is a classical MOA.

Before introducing our method MUTI in detail, we first show the coding

schema of the chromosome. For JIT-SDP problem, a chromosome can be coded as the coefficient vector $w$ and we can use $benefit()$ and $cost()$ to compute the benefit and cost value of the corresponding value on the given dataset $M$. Our method first initializes population. The population has $N$ chromosomes and each chromosome is randomly generated (i.e., the value of each element in $w$ is assigned a random real value). Then it uses classical evolutionary operators to generate new chromosomes, which maybe have better cost or better benefit value. For example, crossover operator will randomly choose two chromosomes according to crossover probability, perform crossover operation, and generate two new chromosomes. Mutation operator will randomly choose a chromosome according to mutation probability, perform mutation operation, and generate a new chromosome. Later it performs selection operation to select high-quality chromosomes into the new population using the fast non-dominated sorting algorithm and the concept of crowding distance [12]. After a sufficient population evolution, it will satisfy the termination criterion and converge to stable solutions. Finally it returns all the Pareto optimal solutions in the current population.

It is worth noting that all the Pareto optimal solutions are generated based on the training set. Then each solution (i.e., coefficient vector) can be used to construct a JIT-SDP model respectively based on logistic regression. Finally we apply these models to perform defect prediction on the testing set.

## 4. Experimental Setup

To verify the effectiveness of our proposed method MULTI, we design the following three research questions.

**RQ1:** How about the performance of our proposed method MULTI when compared to baseline methods in cross-validation scenario?

**RQ2:** How about the performance of our proposed method MULTI when compared to baseline methods in cross-project-validation scenario?

**RQ3:** How about the performance of our proposed method MULTI when

compared to baseline methods in timewise-cross-validation scenario?

Here we consider two performance metrics $ACC$ and $P_{OPT}$, which are based on SQA efforts, to evaluate different methods. More details of these two metrics can be found in Section 4.2. Based on these three RQs, we want to compare our proposed method MULTI with classical baseline methods (including supervised and unsupervised methods) in three different model performance validation scenarios, which is consistent with Yang et al. [8].

### *4.1. Experimental Subjects*

In our empirical studies, we consider 6 open-source projects, which were first used by Kamei et al. [5]. They considered 14 metrics based on analysis the characteristics on code change. The summarization (i.e., dimension, name, and description) of these metrics is shown in Table 2. These metrics have been grouped into 5 dimensions. In particular, the diffusion dimension characterizes the distribution of a change. The assumption is that a highly distributed change is more likely to be a buggy change [6, 23]. The size dimension characterizes the size of a change. The assumption is that a complex change is expected to be a buggy change with high probability [24, 25]. The assumption of purpose dimension is that a defect-fixing change is more likely to introduce new defects [26]. This history dimension assumes that a defect is more likely introduced by a change if the touched files have been modified by more developers [27]. The experience dimension assumes that experienced developers are less likely to introduce defects when modifying codes [6].

These six open-source projects are large-scale, well-known, and long-lived. They also cover a wide range of domains. For example, bugzilla is a web-based bug tracking system, Eclipse JDT is a Java development tool, Mozilla is a widely used web browser, PostgreSQL is an object-relational database system. All the code changes are extracted from the CVS repositories of the projects and labeled by analyzing bug reports and change logs. Table 3 summarizes the characteristics of datasets used in our empirical studies, including project name (abbreviation), period, the number of code changes and the percent of buggy

Table 2: Summarization of Metrics used for JIT-SDP [**5**]

| Dimension | Name | Description |
|---|---|---|
| Diffusion | NS | Number of modified subsystems |
| | ND | Number of modified directories |
| | NF | Number of modified files |
| | Entropy | Distribution of modified code across each file |
| Size | LA | Lines of code added |
| | LD | Lines of code deleted |
| | LT | Lines of code in a file before the change |
| Purpose | FIX | Whether or not the change is a defect fix |
| History | NDEV | The number of deveopers that changed the modified files |
| | AGE | The average time interval between the last and current change |
| | NUC | The number of unique changes to the modified files |
| Experience | EXP | Developer experience |
| | REXP | Recent deveoper experience |
| | SEXP | Developer experience on a sub-system |

changes.

Table 3: Summarization of Datasets

| Project(Abbreviation) | Period | # Changes | % Buggy |
|---|---|---|---|
| Bugzilla (BUG) | 08/1998-12/2006 | 4620 | 37% |
| Columba (COL) | 11/2002-07/2006 | 4455 | 31% |
| Eclipse JDT (JDT) | 05/2001-12/2007 | 35386 | 14% |
| Eclipse Platform (PLA) | 05/2001-12/2007 | 64250 | 15% |
| Mozilla (MOZ) | 01/2000-12/2006 | 98275 | 5% |
| PostgreSQ (POS) | 07/1996-05/2010 | 20431 | 25% |

### *4.2. Performance Metrics*

To evaluate the performance of the JIT-SDP model, we consider the SQA efforts. Similar to Kamei et al. [5], we use the code churn (i.e., the total number of lines added and deleted) to measure the SQA efforts and consider two effort-aware performance metrics [28, 29]. In particular, *ACC* denotes the recall of buggy changes when using 20% of the entire efforts. Previous work [30] shows that defect distribution in software project satisfies 20-80 principle, that is the majority of defects (about 80%) are contained in a small number of program modules (about 20% measure in LOC). This principle is partially confirmed in our used projects. For example, the LOC of buggy changes is 15.12% of the LOC of all the code changes for JDT project. $P_{opt}$ is the normalized version of the effort-aware performance indicator originally proposed by Mende and Koschke [29]. More details of these two performance metrics can be found in [5, 8].

### *4.3. Model Performance Evaluation Scenarios*

In our empirical studies, we mainly consider three model performance evaluation scenarios. These scenarios are cross-validation, cross-project-validation, and timewise-cross-validation. In particular:

- For cross-validation scenario, we mainly consider 10×10-fold cross validation within the same project.
- For cross-project-validation, we perform JIT-SDP across different projects. We construct models based on one project (i.e., source project) and use these models to predict the changes on another project (i.e, target project).
- For timewise-cross-validation, JIT-SDP is performed within the same project, in which the chronological order of changes based on the commit date is considered [13]. Assuming the changes are divided into $n$ parts, we first construct the models based on the changes from part $i$ and $i+1$. Then we use the constructed models to predict the changes from part $i$+4 and $i$+5.

More details of these three scenarios can be found in [8].

### *4.4. Baseline Methods*

To evaluate the effectiveness of our proposed method, we consider 43 state-of-the-art baseline methods used by yang et al. [8], including 31 supervised methods and 12 unsupervised methods.

All the supervised methods and their abbreviations are summarized in Table 4. Here method EALR [5] uses $buggy(m_i)/LOC(m_i)$ as the dependent variable and uses linear regression to construct the models. The remaining supervised methods are used by a recent paper [19] to revisit their impact on the performance of SDP. These supervised methods can be classified into 6 families. In particular, function family includes 4 methods: linear regression (EALR), simple logistic (SL), radial basis functions network (RBFNet), sequential minimal optimization (SMO). Lazy family includes only 1 method: K-nearest neighbour (Ibk), Rule family includes 2 methods: propositional rule (Jrip) and ripple down rules (Ridor). Bayes family includes only 1 method: Naive Bayes (NB). Tree family includes 3 methods: J48, logistic model tree (LMT), and random forest (RF). Ensemble family considers bagging (BG), Adaboost (AB), rotation forest (RF), and random subspace (RS) ensemble methods. In the abbreviations

of the Ensemble family, BG+LMT means that this method uses LMT as the base learner and uses Bagging as the ensemble method. It is not hard to find that these methods can cover different types of supervised methods in machine learning. In our empirical studies, we use the same parameters setting used by Yang et al. [8] to build these supervised methods.

Unsupervised methods has attracted more interest in current SDP research. These methods do not need training set, are very simple, and have a low model construction cost. We consider 12 methods proposed by Yang et al. [8]. These methods are based on NS, ND, NF, Entropy, LT, FIX, NDEV, AGE, NUC, EXP, REXP, SEXP metrics respectively. For $j$-th metric, it compute the defect-proneness probability for $i$-th change $m_i$ as $1/v_{i,j}$. This means that the smaller the metric value, the higher the defect-proneness probability. Then all the changes will be ranked in descendant order according to the computed probability. These methods are motivated by ManualUp model [31]. In some empirical studies, these unsupervised methods surprisedly perform well both in traditional SDP and JIT-SDP when considering SQA efforts [31, 32, 8]. Except for these unsupervised methods proposed by Yang et al. [8], there also exist other unsupervised methods [33, 34] for SDP. However these methods are not suitable as our baseline methods. The reasons can be summarized as follows: (1) The granularity of program modules is set as file or package in these two studies. For example, datasets NetGene and ReLink used by Nam and Kim [33] set the granularity as file. While in JIT-SDP, the granularity is set as code change, which has coarser granularity. (2) Different granularity results in different metrics. For example, NetGene dataset used network and change genealogy metrics, ReLink used code complexity metrics extracted by the Understand tool [33]. These metrics are different from metrics used by JIT-SDP. (3) The design of these methods [33, 34] does not consider SQA efforts on program modules, therefore these methods can not be applicable to effort-aware JIT-SDP.

Table 4: Overview of the supervised methods

| Family | Method | Abbreviation |
|---|---|---|
| Function | Linear regression | EALR |
| | Simple logistic | SL |
| | Radial basis functions network | RBFNet |
| | Sequential minimal optimization | SMO |
| Lazy | K-nearest neighbour | Ibk |
| Rule | Propositional rule | Jrip |
| | Ripple down rules | Ridor |
| Bayes | Naive Bayes | NB |
| Tree | J48 | J48 |
| | Logistic model tree | LMT |
| | Random Forest | RF |
| Ensemble | Bagging | BG+LMT, BG+NB, BG+SL, BG+SMO, and BG+J48 |
| | Adaboost | AB+LMT, AB+NB, AB+SL, AB+SMO, and AB+J48 |
| | Rotation Forest | RF+LMT, RF+NB, RF+SL, RF+SMO, and RF+J48 |
| | Random Subspace | RS+LMT, RS+NB, RS+SL, RS+SMO, and RS+J48 |

*4.5. Experimental Design*

For MULTI and 31 supervised baseline methods, we conduct a series of data preprocessing on the training set, which is suggested by Kamei et al. [5]. (1) To remove highly correlated metrics, we exclude ND and REXP metrics since NF and ND, REXP and EXP are highly correlated. We normalize LA and LD by dividing by LT, since LA and LD are highly correlated. We also normalize LT and NUC by dividing NF since these metrics have high correlation with NF. (2) Since most metrics are highly skewed, we perform logarithmic transformation to each metric (except for FIX). (3) To deal with the problem of class imbalance in datasets. we apply random under-sampling to the training set. In particular, we will delete clean changes randomly until the number of clean changes keeps same as the number of buggy changes. Note that we do not apply under-sampling to the testing set.

For 12 unsupervised baseline methods, we only use changes in the testing set to construct the model. Thus can make a fair comparison among all the methods.

For our proposed method MULTI, we make the following setting for parameters, which is typically used by MOAs for numerical problems [35]: (1) Population size is set as 200. (2) The range of the element in the coefficient vector is within the interval [-10000, 10000]. While in population initialization, the range is within the interval [-10, 10]. (3) The maximum number of generation is set as 400. (4) Crossover probability is set as 0.5. (5) Mutation probability is set as 1/11, here 11 is the number of metrics used in our JIT-SDP research.

To examine whether there is a significant difference in the prediction performance between our proposed method and other baseline methods. We firstly use the Benjamini-Hochberg (BH) corrected $p$-value [36] to examine whether a difference is statistically significant at the significance level of 0.05. If the statistical test shows a significant difference, we then use the Cliff's $\delta$ to measure the magnitude of the difference. The meaning of different Cliffs $\delta$ values and their corresponding interpretation is shown in Table 5. In summary, our proposed method performs significantly better/worse than a baseline method, if BH cor-

rected $p$-value is less than 0.05 and the effectiveness level is not negligible based on Cliff's $\delta$. The difference between our method and a baseline method is not significant if $p$-value is not less than 0.05 or the effectiveness level is negligible.

Table 5: Cliff's $\delta$ and the Effectiveness Level [36]

| Cliff's $\delta$ | Effectiveness Level |
|---|---|
| $\|\delta\| < 0.147$ | Negligible |
| $0.147 \leq \|\delta\| < 0.33$ | Small |
| $0.33 \leq \|\delta\| < 0.474$ | Medium |
| $0.474 \leq \|\delta\|$ | Large |

As there is randomness variation inherent in our method, we perform 10 independent runs to get a high statistical confidence. For each run, we can get a Pareto front based on the training set. Therefore after 10 independent runs, we can get 10 Pareto fronts. Supposing the $i$-th Pareto front has $l_i$ solutions, then the 10 Pareto fronts will have $\sum_{i=1}^{10} l_i$ solutions in total. We firstly use **MULTI-B** to gather the best result of the solutions in these 10 Pareto fronts in the given testing set. Then we use **MULTI-M** to gather the median result of the solutions in these 10 Pareto fronts in the given testing set. It is not hard to find that MULTI-B and MULTI-M can denote the optimal performance and the average performance of MULTI method respectively. The running process can be found in Fig. 2.

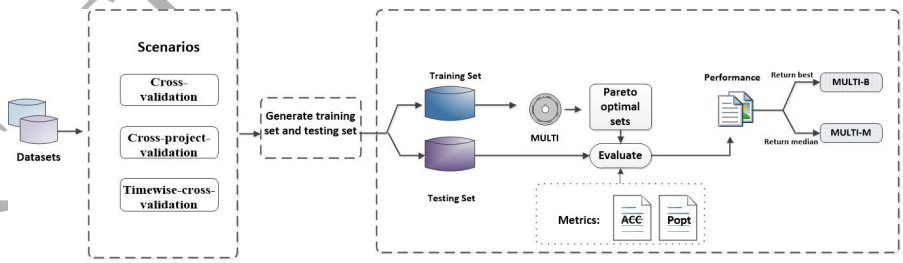


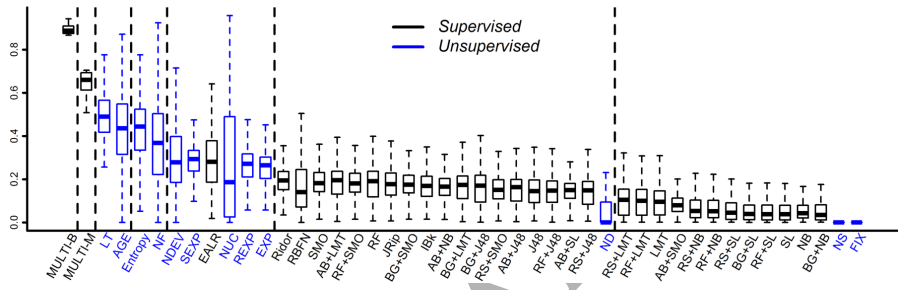

Figure 2: The Running Process of Our Proposed Method

## 5. Result Analysis

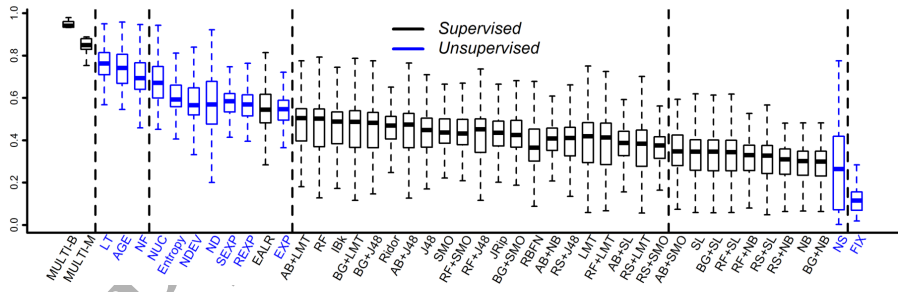
### *5.1. Analysis for RQ1*

To make an analysis for RQ1, we use the Scott-Knott test [37] to group all the methods into statistically distinct ranks ($\alpha$=0.05). In particular, it uses hierarchical cluster analysis to partition the methods into ranks. It starts by dividing the methods into two ranks on the basis of mean performance metric values ($ACC$ or $P_{OPT}$). If the divided ranks are statistically significantly different, then it recursively executes again within each rank to further divide the ranks. It terminates when ranks can no longer be divided into statistically distinct ranks. The result is shown in Fig. 3. The dotted lines represent groups divided by using the Scott-Knott test. All methods are ordered based on their mean ranks. The distribution of $P_{OPT}$ and $ACC$ at cross-validation scenario over all the six projects is shown using boxplot. The blue label denotes supervised methods and the black label denotes unsupervised methods.

From Fig. 3, we can find that MULTI performs significantly better than all the baseline supervised and unsupervised methods when using $ACC$ performance metric or $P_{OPT}$ performance metric. Then we further compare MULTI to baseline methods for each project. To make a fair comparison, here we firstly choose the best two supervised baseline methods and the best two unsupervised baseline methods respectively. Then we only use MULTI-M to make a comparison with these four baseline methods. The median results of these chosen methods are shown in Table 6 and Table 7. The row "Average" show the average value over all the six projects. The row "W/D/L" summarizes the number of projects for which MULTI-M obtains a better, equal, and worse performance than a specific baseline method (i.e., Win/Draw/Loss analysis). If we use $ACC$ metric, we can find our MULTI-M can identify 63.8% buggy changes on average when using only 20% efforts, which can improve 119%, 231%, 30%, and 45% when compared to EALR, Ridor, LT, and AGE. If we use $P_{OPT}$ metric, we can find our MULTI-M can improve 50%, 73%, 11%, and 14% on average when compared to EALR, AB+LMT, LT, and AGE. Based on "W/D/L" analysis, our




(a) ACC




(b) $P_{OPT}$

Figure 3: The Result of Scott-Knott Test in Cross-validation Scenario

MULTI-M can perform significantly better than four chosen baseline methods on all the projects both for $ACC$ and $P_{OPT}$ metrics.

Table 6: MULTI-M vs Top Two Supervised and Unsupervised Methods in Cross-Validation Scenario using *ACC*

| Project | Supervised | | | Unsupervised | |
|---|---|---|---|---|---|
| | MULTI-M | EALR | Ridor | LT | AGE |
| BUG | 0.696 | 0.407 | 0.226 | 0.476 | 0.441 |
| COL | 0.696 | 0.414 | 0.162 | 0.590 | 0.659 |
| JDT | 0.626 | 0.215 | 0.196 | 0.531 | 0.459 |
| PLA | 0.684 | 0.289 | 0.217 | 0.466 | 0.409 |
| MOZ | 0.511 | 0.148 | 0.198 | 0.367 | 0.241 |
| POS | 0.613 | 0.271 | 0.157 | 0.504 | 0.427 |
| Average | 0.638 | 0.291 | 0.193 | 0.489 | 0.439 |
| W/D/L | — | 6/0/0 | 6/0/0 | 6/0/0 | 6/0/0 |

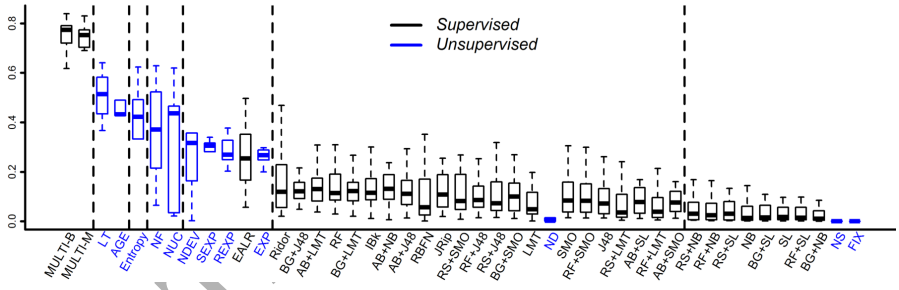
### *5.2. Analysis for RQ2*

The result of Scott-Knott test in cross-project scenario is shown in Fig. 4. The result is also promising. Our proposed method MULTI can perform significantly better than all the baseline methods when using $ACC$ performance metric or $P_{OPT}$ performance metric.

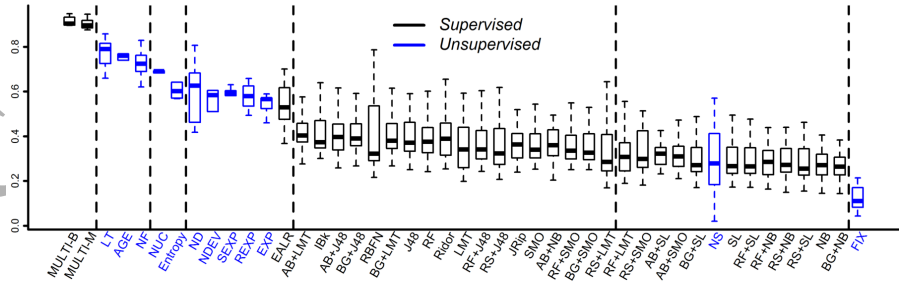
Then we further compare MULTI to baseline methods for each cross-project prediction. Here we still choose the best two supervised baseline methods and the best two unsupervised baseline methods respectively. Final results are shown in Table 8 and Table 9. The "Average" is summarized in the last row. In these two tables, the first column presents specific cases of cross-project prediction. For example, in Table 8, the case "BUG $\Rightarrow$ COL" means that the project BUG is used as the source project to construct the model, then this model is used to predict the changes in the target project COL. For $ACC$ metric, based on

Table 7: MULTI-M vs Top Two Supervised and Unsupervised Methods in Cross-Validation Scenario using $P_{OPT}$

| Project | Supervised | | | Unsupervised | |
|---|---|---|---|---|---|
| | MULTI-M | EALR | AB+LMT | LT | AGE |
| BUG | 0.883 | 0.722 | 0.625 | 0.756 | 0.757 |
| COL | 0.880 | 0.605 | 0.371 | 0.828 | 0.848 |
| JDT | 0.829 | 0.514 | 0.515 | 0.777 | 0.744 |
| PLA | 0.853 | 0.549 | 0.511 | 0.748 | 0.715 |
| MOZ | 0.757 | 0.453 | 0.527 | 0.655 | 0.620 |
| POS | 0.843 | 0.517 | 0.361 | 0.796 | 0.751 |
| Average | 0.841 | 0.560 | 0.485 | 0.760 | 0.739 |
| W/D/L | — | 6/0/0 | 6/0/0 | 6/0/0 | 6/0/0 |




(a) ACC



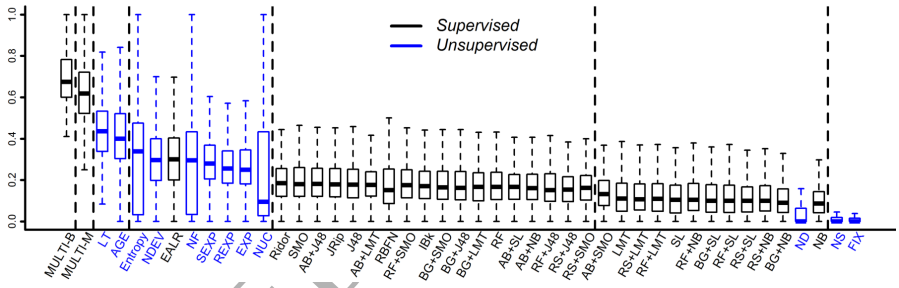

(b) $P_{OPT}$

Figure 4: The Result of Scott-Knott Test in Cross-project Scenario

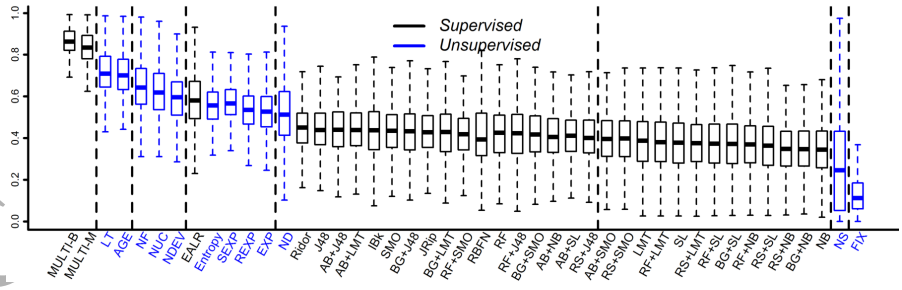
average value, we can find that our MULTI-M can identify 73% buggy changes on average when using only 20% efforts, which can improve 178%, 390%, 43%, and 61% respectively when compared to EALR, Ridor, LT, and AGE. For $P_{OPT}$ metric, based on average value, we can find that our MULTI-M can improve 66%, 104%, 15%, and 18% respectively on average when compared to EALR, AB+LMT, LT, and AGE.

### *5.3. Analysis for RQ3*

The result of Scott-Knott test in timewise-cross-project scenario over the six projects can be found in Fig. 5. The result is consistent with previous two scenarios. we can find that MULTI performs significantly better than all the baseline methods when using *ACC* metric or $P_{OPT}$ metric.




(a) ACC




(b) $P_{OPT}$

Figure 5: The Result of Scott-Knott Test in Timewise-cross-validation Scenario

Then we further compare MULTI to baseline methods for each each project.

Table 8: MULTI-M vs Top Two Supervised and Unsupervised Methods in Cross-Project Scenario using *ACC*

| Source ⇒ | Supervised | | | Unsupervised | |
|---|---|---|---|---|---|
| Target | MULTI-M | EALR | Ridor | LT | AGE |
| BUG ⇒ COL | 0.716 | 0.497 | 0.468 | 0.641 | 0.702 |
| BUG⇒ JDT | 0.773 | 0.281 | 0.308 | 0.582 | 0.490 |
| BUG⇒ PLA | 0.772 | 0.429 | 0.275 | 0.494 | 0.427 |
| BUG⇒ MOZ | 0.763 | 0.219 | 0.265 | 0.367 | 0.240 |
| BUG⇒ POS | 0.765 | 0.349 | 0.267 | 0.533 | 0.432 |
| COL⇒ BUG | 0.821 | 0.336 | 0.056 | 0.435 | 0.432 |
| COL ⇒ JDT | 0.828 | 0.113 | 0.051 | 0.582 | 0.490 |
| COL ⇒ PLA | 0.824 | 0.239 | 0.189 | 0.494 | 0.427 |
| COL ⇒ MOZ | 0.830 | 0.148 | 0.168 | 0.367 | 0.240 |
| COL ⇒ POS | 0.825 | 0.167 | 0.023 | 0.533 | 0.432 |
| JDT ⇒ BUG | 0.757 | 0.376 | 0.052 | 0.435 | 0.432 |
| JDT ⇒ COL | 0.721 | 0.378 | 0.077 | 0.641 | 0.702 |
| JDT ⇒ PLA | 0.733 | 0.277 | 0.184 | 0.494 | 0.427 |
| JDT ⇒ MOZ | 0.740 | 0.168 | 0.173 | 0.367 | 0.240 |
| JDT ⇒ POS | 0.749 | 0.299 | 0.020 | 0.533 | 0.432 |
| PLA ⇒ BUG | 0.768 | 0.367 | 0.084 | 0.435 | 0.432 |
| PLA ⇒ COL | 0.763 | 0.363 | 0.130 | 0.641 | 0.702 |
| PLA ⇒ JDT | 0.772 | 0.152 | 0.229 | 0.582 | 0.490 |
| PLA ⇒ MOZ | 0.780 | 0.168 | 0.207 | 0.367 | 0.240 |
| PLA ⇒ POS | 0.780 | 0.230 | 0.053 | 0.533 | 0.432 |
| MOZ ⇒ BUG | 0.592 | 0.350 | 0.075 | 0.435 | 0.432 |
| MOZ ⇒ COL | 0.572 | 0.247 | 0.104 | 0.641 | 0.702 |
| MOZ ⇒ JDT | 0.587 | 0.057 | 0.066 | 0.582 | 0.490 |
| MOZ ⇒ PLA | 0.580 | 0.173 | 0.265 | 0.494 | 0.427 |
| MOZ ⇒ POS | 0.584 | 0.135 | 0.030 | 0.533 | 0.432 |
| POS ⇒ BUG | 0.711 | 0.351 | 0.052 | 0.435 | 0.432 |
| POS ⇒ COL | 0.690 | 0.439 | 0.107 | 0.641 | 0.702 |
| POS ⇒ JDT | 0.703 | 0.159 | 0.063 | 0.582 | 0.490 |
| POS ⇒ PLA | 0.700 | 0.261 | 0.255 | 0.494 | 0.427 |
| POS ⇒ MOZ | 0.706 | 0.159 | 0.185 | 0.367 | 0.240 |
| Average | 0.730 | 0.263 | 0.149 | 0.509 | 0.454 |

Table 9: MULTI-M vs Top Two Supervised and Unsupervised Methods in Cross-Project Scenario using $P_{OPT}$

| Source ⇒ | Supervised | | | Unsupervised | |
|---|---|---|---|---|---|
| Target | MULTI-M | EALR | AB+LMT | LT | AGE |
| BUG ⇒ COL | 0.894 | 0.701 | 0.569 | 0.858 | 0.868 |
| BUG ⇒ JDT | 0.923 | 0.592 | 0.599 | 0.815 | 0.769 |
| BUG ⇒ PLA | 0.922 | 0.695 | 0.613 | 0.770 | 0.740 |
| BUG ⇒ MOZ | 0.918 | 0.563 | 0.576 | 0.660 | 0.622 |
| BUG ⇒ POS | 0.918 | 0.603 | 0.646 | 0.810 | 0.762 |
| COL ⇒ BUG | 0.945 | 0.633 | 0.354 | 0.726 | 0.758 |
| COL ⇒ JDT | 0.945 | 0.418 | 0.276 | 0.815 | 0.769 |
| COL ⇒ PLA | 0.944 | 0.506 | 0.313 | 0.770 | 0.740 |
| COL ⇒ MOZ | 0.946 | 0.476 | 0.448 | 0.660 | 0.622 |
| COL ⇒ POS | 0.943 | 0.426 | 0.337 | 0.810 | 0.762 |
| JDT ⇒ BUG | 0.895 | 0.674 | 0.437 | 0.726 | 0.758 |
| JDT ⇒ COL | 0.876 | 0.564 | 0.437 | 0.858 | 0.868 |
| JDT ⇒ PLA | 0.883 | 0.555 | 0.402 | 0.770 | 0.740 |
| JDT ⇒ MOZ | 0.885 | 0.502 | 0.486 | 0.660 | 0.622 |
| JDT ⇒ POS | 0.891 | 0.514 | 0.324 | 0.810 | 0.762 |
| PLA ⇒ BUG | 0.895 | 0.685 | 0.409 | 0.726 | 0.758 |
| PLA ⇒ COL | 0.889 | 0.564 | 0.356 | 0.858 | 0.868 |
| PLA ⇒ JDT | 0.897 | 0.458 | 0.414 | 0.815 | 0.769 |
| PLA ⇒ MOZ | 0.899 | 0.498 | 0.511 | 0.660 | 0.622 |
| PLA ⇒ POS | 0.899 | 0.478 | 0.451 | 0.810 | 0.762 |
| MOZ ⇒ BUG | 0.806 | 0.619 | 0.438 | 0.726 | 0.758 |
| MOZ ⇒ COL | 0.793 | 0.482 | 0.457 | 0.858 | 0.868 |
| MOZ ⇒ JDT | 0.803 | 0.367 | 0.400 | 0.815 | 0.769 |
| MOZ ⇒ PLA | 0.799 | 0.414 | 0.416 | 0.770 | 0.740 |
| MOZ ⇒ POS | 0.800 | 0.382 | 0.443 | 0.810 | 0.762 |
| POS ⇒ BUG | 0.902 | 0.629 | 0.315 | 0.726 | 0.758 |
| POS ⇒ COL | 0.884 | 0.617 | 0.431 | 0.858 | 0.868 |
| POS ⇒ JDT | 0.897 | 0.459 | 0.400 | 0.815 | 0.769 |
| POS ⇒ PLA | 0.894 | 0.544 | 0.381 | 0.770 | 0.740 |
| POS ⇒ MOZ | 0.898 | 0.497 | 0.432 | 0.660 | 0.622 |
| Average | 0.889 | 0.537 | 0.436 | 0.773 | 0.753 |

Here we still choose the best two supervised baseline methods and the best two unsupervised baseline methods respectively. Final results are shown in Table 10 and Table 11. The "Average" and "W/D/L" are summarized in the last two rows. For *ACC* metric, based on average value, we can find that our MULTI-M can identify 62.3% buggy changes on average when using only 20% efforts, which can improve 102%, 219%, 46%, and 50% respectively when compared to EALR, Ridor, LT, and AGE. For $P_{OPT}$ metric, based on average value, we can find that our MULTI-M can improve 44%, 87%, 18%, and 20% on average when compared to EALR, Ridor, LT, and AGE. Based on "W/D/L" analysis, our MULTI-M can perform significantly better than four chosen baseline methods on all projects both for *ACC* and $P_{OPT}$ metrics.

Table 10: MULTI-M vs Top Two Supervised and Unsupervised Methods in Timewise-Cross-Validation Scenario using *ACC*

| Project | Supervised | | | Unsupervised | |
|---|---|---|---|---|---|
| | MULTI-M | EALR | Ridor | LT | AGE |
| BUG | 0.598 | 0.286 | 0.218 | 0.449 | 0.375 |
| COL | 0.686 | 0.400 | 0.200 | 0.440 | 0.568 |
| JDT | 0.625 | 0.323 | 0.197 | 0.452 | 0.408 |
| PLA | 0.688 | 0.305 | 0.204 | 0.432 | 0.429 |
| MOZ | 0.549 | 0.180 | 0.187 | 0.363 | 0.280 |
| POS | 0.592 | 0.356 | 0.165 | 0.432 | 0.426 |
| Average | 0.623 | 0.308 | 0.195 | 0.428 | 0.414 |
| W/D/L | — | 6/0/0 | 6/0/0 | 6/0/0 | 6/0/0 |

### *5.4. Discussions*

In this subsection, we firstly use a specific case to illustrate why our proposed method MULTI is promising. In this case, we use the dataset from Bugzilla as the training set, and use the dataset from Columba as the testing set. Then

Table 11: MULTI-M vs Top Two Supervised and Unsupervised Methods in Timewise-Cross-Validation Scenario using $P_{OPT}$

| Project | Supervised | | | Unsupervised | |
|---|---|---|---|---|---|
| | MULTI-M | EALR | Ridor | LT | AGE |
| BUG | 0.838 | 0.596 | 0.480 | 0.721 | 0.661 |
| COL | 0.885 | 0.619 | 0.429 | 0.732 | 0.786 |
| JDT | 0.828 | 0.590 | 0.443 | 0.709 | 0.685 |
| PLA | 0.859 | 0.583 | 0.476 | 0.717 | 0.709 |
| MOZ | 0.786 | 0.498 | 0.466 | 0.651 | 0.638 |
| POS | 0.837 | 0.600 | 0.398 | 0.742 | 0.731 |
| Average | 0.839 | 0.581 | 0.449 | 0.712 | 0.702 |
| W/D/L | — | 6/0/0 | 6/0/0 | 6/0/0 | 6/0/0 |

we draw effort-based cumulative lift charts of different methods in Fig. 6. Here $x$-axis shows the percentage of used SQA efforts and $y$-axis shows the percentage of identified buggy changes. Each line plots the cumulative lift chart for changes order by decreasing defect-proneness probability given a corresponding method. These methods are MULTI (denoted by MUTI-B and MULTI-M), the best supervised method EALR and the best unsupervised method LT. In this figure we can find that our method can assign buggy changes with less SQA efforts higher defect-proneness probability by using multi-objective optimization. Therefore our method MULTI can get higher value of $ACC$ and $P_{OPT}$ than state-of-the-art baseline methods.

**In the rest of this subsection, we first compare MULTI with a simple genetic algorithm and a random search algorithm to show the challenge of JIT-SDP problem. Then we compare MULTI with OneWay method, which is recently proposed by Fu and Menzies [15]. Later we analyze the competitiveness of MULTI based on $F1$ metric and perform model construction time cost analysis on MULTI.**

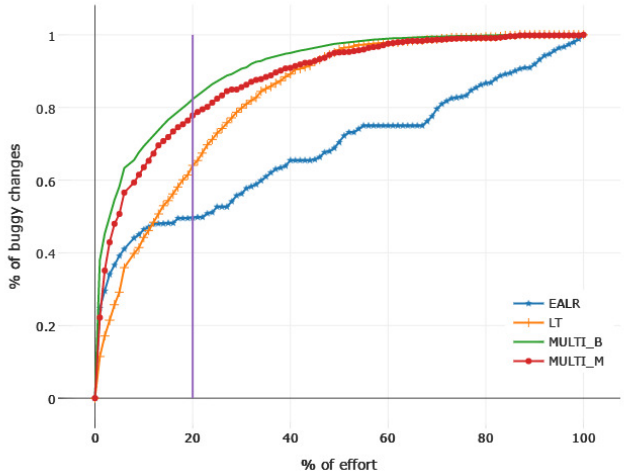



Figure 6: Effort-based Cumulative Lift Chart of Different Methods for BUG ⇒ COL

*5.4.1. Comparison with A Simple Genetic Algorithm and A Random Search Algorithm*

**To show that JIT-SDP is a complex problem with a large-scale search space, we use a simple genetic algorithm based on single objective and a random search algorithm based on multiple objectives to compare with MULTI.**

**We use SIMPLE-GA to denote the simple genetic algorithm. Given a set of changes $M$ and a solution $w$, the objective of this solution can be computed by the following formula:**

$$0.5 \times \frac{benefit(w)}{|M|} + 0.5 \times \frac{cost(w)}{\sum_{m_i \in M} SQA(m_i)} \quad (5)$$

**Here we combine two different objectives into a single objective by taking an average (i.e., the weight of each objective is set as 0.5) and these objectives are scaled into the same range (i.e., [0,1]).**

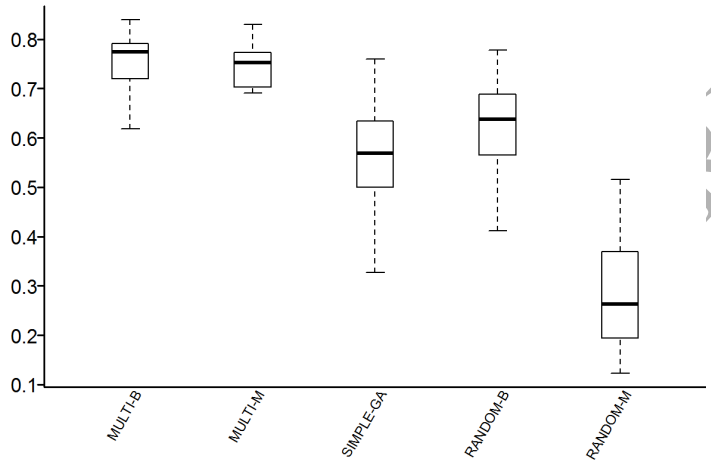
**We use RANDOM to denote the random search algorithm. For each iteration, it will generate new chromosomes (i.e., solutions) ran-**

domly and then select high-quality chromosomes from old chromosomes and new chromosomes into the new population. When reaching the maximum number of generations, it returns all the Pareto optimal solutions in the current population.

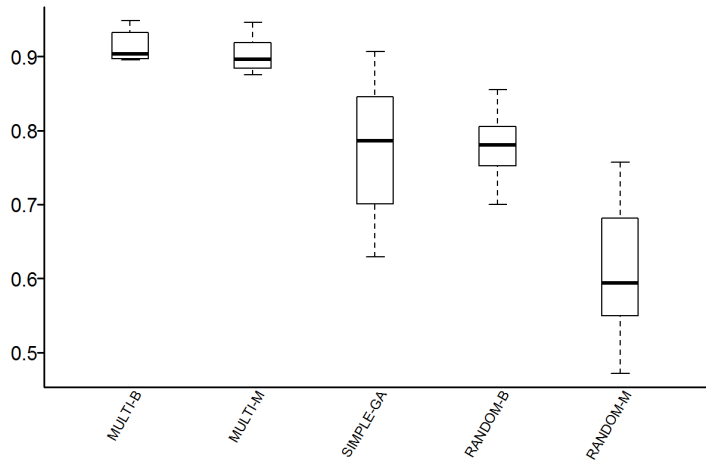
As there is randomness inherent in SIMPLE-GA and RANDOM, we also perform 10 independent runs. For SIMPLE-GA, we gather the result of the solution with the highest fitness value in these 10 runs. For RANDOM, we gather the result, which keeps in line with MULTI. In particular, we firstly use RANDOM-B to gather the best result of the solutions in these 10 Pareto fronts in the given testing set. Then we use RANDOM-M to gather the median result of the solutions in these 10 Pareto fronts in the given testing set. The setting for parameters (such as the population size, the maximum number of generation, the range of the element in the coefficient vector) of SIMPLE-GA and RANDOM is the same as MULTI.

Final result in cross-project scenario is shown in Fig. 7. The result shows that MULTI can perform significantly better than SIMPLE-GA and RANDOM when using $ACC$ performance metric or $P_{OPT}$ performance metric. From the result, we can find that for JIT-SDP, using simple genetic algorithms or random search algorithms can not obtain good results and therefore we should consider more complicated search algorithms, such as MOAs used in this article.

Finally, we use quality indicators to evaluate the quality of Pareto fronts generated by MOAs. These quality indicators can be classified into four categories: convergence, diversity, combination of convergence and diversity, and coverage [38]. In this article, we select Hypervolume (HV) [39] because of its popularity. HV can measure the volume in the objective space that is covered by a Pareto front, therefore it can measure both the convergence and the diversity of a Pareto front. A higher value of HV means a better quality of the Pareto front. In Fig. 8, we Comparing HV of Pareto fronts gener-




(a) ACC




(b) $P_{OPT}$

Figure 7: Comparison with SIMPLE-GA and RANDOM in Cross-project Scenario

**ated by MULTI and RANDOM in Cross-validation Scenario for each dataset. Here BUG-M means the Pareto fronts generated by MULTI based on BUG dataset and BUG-R means the Pareto fronts generated by RANDOM based on BUG dataset. For each dataset, the value of HV is scaled into the range [0,1] by dividing the maximum value of HV of 20 Pareto fronts (i.e., 10 Pareto fronts generated by MULIT and 10 Pareto fronts generated by RANDOM). Final results show MULTI can generate high quality Pareto fronts than RANDOM for each dataset.**

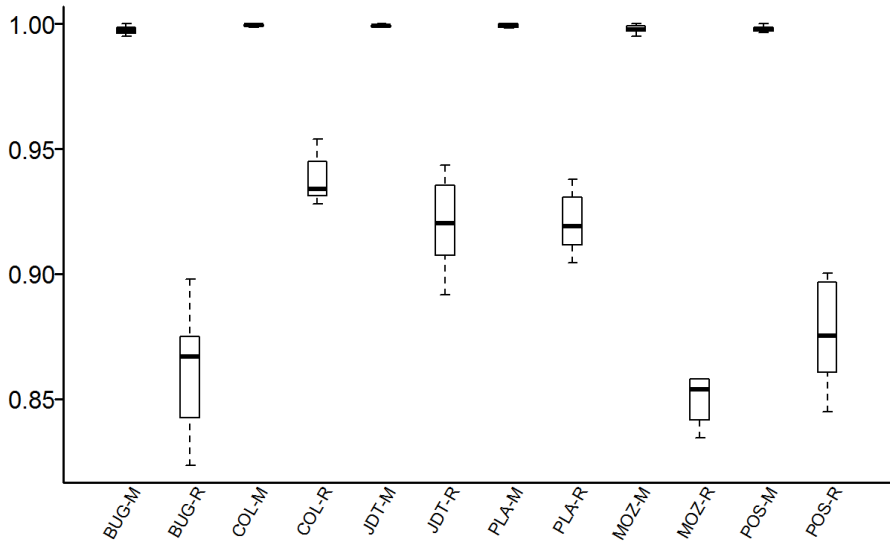
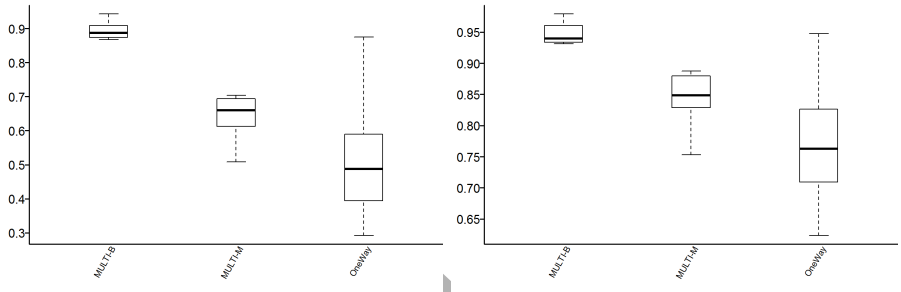



Figure 8: MULTI vs RANDOM based on HV in Cross-validation Scenario

*5.4.2. Comparison with OneWay method*

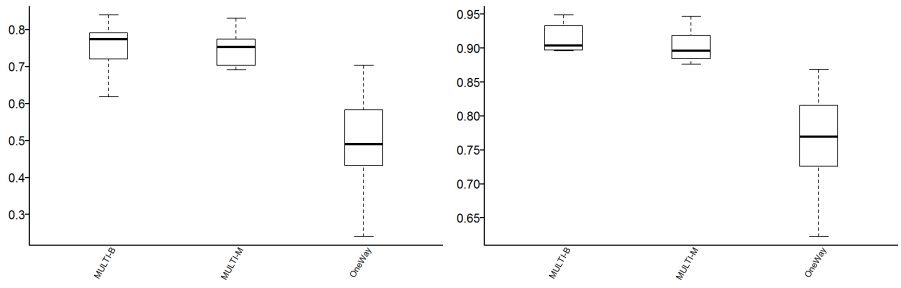
**OneWay [15] is a newest supervised method, which is based on the simple unsupervised methods proposed by Yang et al. [8]. It removes all but one of Yang et al. methods based on the analysis on the training set, and then it applies the chosen method to the testing set. In empirical studies, Fu and Menzies found that OneWay has**

**competitive performance and it performs better than most unsupervised methods [8] for effort-aware JIT-SDP. Therefore it is necessary to compare MULTI with OneWay. Final result in cross-validation scenario is shown in Fig. 9. The distribution of $P_{OPT}$ and $ACC$ over all the six projects is shown using boxplot. In addition, results for cross-project scenario and for timewise-cross-validation scenario are also shown in Fig. 10 and Fig. 11 respectively. From these figures, we can find that MULTI can perform significantly better than OneWay.**



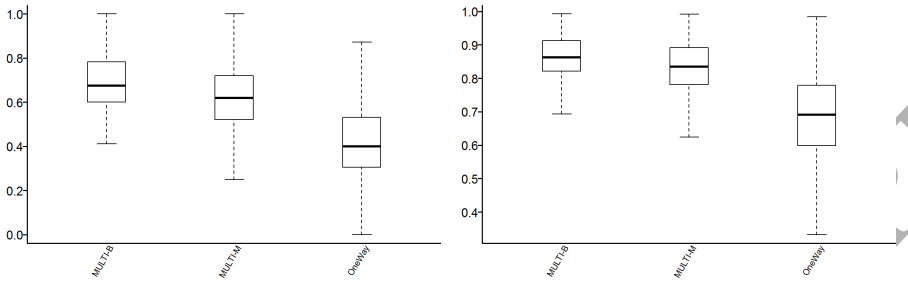

(a) ACC (b) $P_{OPT}$

Figure 9: MULTI vs OneWay in Cross-validation Scenario




(a) ACC (b) $P_{OPT}$

Figure 10: MULTI vs OneWay in Cross-project Scenario




(a) ACC (b) $P_{OPT}$

Figure 11: MULTI vs OneWay in Timewise-cross-validation Scenario

*5.4.3. Performance Comparison on F1 metric*

Except for *ACC* and $P_{OPT}$ performance metrics which are widely used in effort-aware JIT-SDP, we also compare MULTI using $F1$ performance metric which is used in previous SDP [1, 7, 10, 9, 16, 33] research. To be consistent with previous studies, we restrict our efforts to 20% of the entire efforts and predict these changes as buggy changes. We use *precision* to denote the percentage of actual buggy changes to all the predicted buggy changes, and use *recall* (i.e., *ACC* metric in this article) to denote the percentage of predicted buggy changes to all the actual buggy changes. $F1$ is the harmonic mean of *precison* and *recall*. Due to the limitation of this article, we show the result of Scott-Knott test in cross-project scenario in Fig. 12. From this Figure, we can find that MULTI can still perform significantly better than all the baseline methods when using $F1$ metric. For $F1$ metric, based on average value, we can find that our MULTI-M can improve 24%, 45%, 18%, and 9% on average when compared to the best two supervised methods (i.e., EALR and BG+J48) and the best two unsupervised methods (i.e., AGE and LT).

*5.4.4. Model Construction Time Cost Analysis*

Here we analyze the model construction time cost of MULTI. Since unsupervised methods do not need training set and are very simple (i.e., compute the defect-proness probability only by using the specific metric value), their running

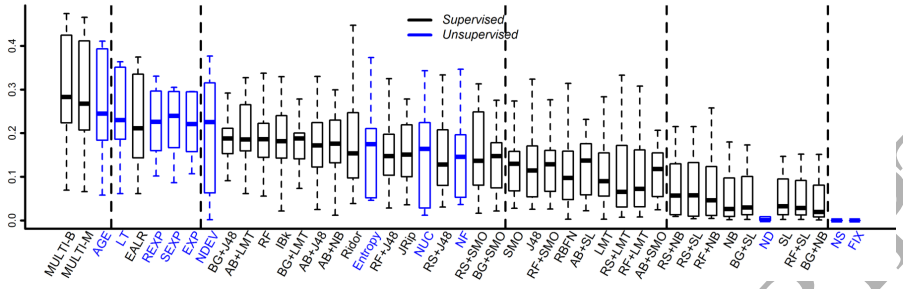



Figure 12: The Result of Scott-Knott Test in Cross-project Scenario using $F1$ metric

time is very fast. Therefore we only compare MULTI with 31 supervised methods. All the methods are run on Win 10 operation system (Intel i5-4210U CPU with 8 GB of memory). The model construction time in cross-project scenario is shown in Table 12.

From Table 12, we can find that the model construction time of MULTI method is acceptable (i.e., 20.91 seconds $\sim$ 98.20 seconds). Even in some cases, MULTI method has lower model construction cost than some supervised methods. For example, on BUG dataset, the model construction time of MULTI is 24.10 seconds, however for some ensemble methods BG+LMT, AB+LMT, RF+LMT, the construction time is 29.10, 31.09, and 32.96 respectively. Moreover based on the results of empirical studies, our proposed MULTI can achieve better performance than all the baseline methods. Therefore we think MULTI is applicable in practice.

### *5.5. Threats to Validity*

In this subsection, we mainly discuss the potential threats to validity of our empirical studies.

Threats to external validity are about whether the observed experimental results can be generalized to other subjects. To guarantee the representative of our empirical subjects, we chose datasets which have been widely used in previous JIT-SDP research works [5, 8, 9, 10]. In addition, to guarantee the representative of the model construction algorithm, we choose logistic regression

Table 12: Model Construction Time of Different Supervised Methods (Unit: Seconds)

| Method | BUG | COL | JDT | PLA | MOZ | POS |
|---|---|---|---|---|---|---|
| MULTI | 24.10 | 20.91 | 55.82 | 98.20 | 57.30 | 57.42 |
| EALR | 0.40 | 0.22 | 0.55 | 0.90 | 0.93 | 0.83 |
| SL | 0.55 | 0.55 | 1.73 | 2.84 | 1.93 | 1.43 |
| RBFNet | 0.11 | 0.09 | 0.19 | 0.28 | 0.16 | 0.17 |
| SMO | 0.37 | 0.27 | 4.44 | 10.51 | 2.74 | 6.43 |
| Ibk | 1.49 | 1.01 | 12.62 | 35.22 | 10.37 | 9.68 |
| Jrip | 0.39 | 0.21 | 0.69 | 1.71 | 2.17 | 1.30 |
| Ridor | 0.29 | 0.24 | 1.58 | 3.39 | 1.02 | 1.28 |
| NB | 0.06 | 0.05 | 0.12 | 0.17 | 0.09 | 0.10 |
| J48 | 0.10 | 0.10 | 0.35 | 0.67 | 0.31 | 0.26 |
| LMT | 1.18 | 1.76 | 13.49 | 25.63 | 12.47 | 6.72 |
| RF | 1.83 | 1.42 | 6.42 | 10.65 | 5.03 | 5.26 |
| BG+LMT | 29.10 | 32.73 | 2.30 | 5.72 | 1.86 | 1.54 |
| BG+NB | 0.26 | 0.24 | 0.72 | 1.14 | 0.62 | 0.63 |
| BG+SL | 6.11 | 4.64 | 19.71 | 37.31 | 17.25 | 14.90 |
| BG+SMO | 3.19 | 2.44 | 41.41 | 1.72 | 27.39 | 57.47 |
| BG+J48 | 0.78 | 0.77 | 3.51 | 8.38 | 3.11 | 3.12 |
| AB+LMT | 31.09 | 26.66 | 3.50 | 5.55 | 1.86 | 2.46 |
| AB+NB | 0.61 | 0.39 | 1.82 | 2.29 | 1.52 | 1.48 |
| AB+SL | 3.00 | 1.78 | 15.14 | 16.80 | 9.23 | 7.95 |
| AB+SMO | 3.36 | 1.47 | 18.68 | 1.78 | 25.81 | 18.89 |
| AB+J48 | 0.58 | 0.47 | 2.05 | 4.71 | 2.10 | 2.17 |
| RF+LMT | 32.96 | 16.22 | 1.72 | 4.76 | 1.31 | 1.31 |
| RF+NB | 0.75 | 0.60 | 1.99 | 2.98 | 1.57 | 1.58 |
| RF+SL | 7.80 | 5.13 | 19.32 | 48.59 | 18.01 | 16.83 |
| RF+SMO | 2.28 | 1.63 | 25.80 | 1.02 | 16.48 | 39.37 |
| RF+J48 | 1.27 | 0.89 | 4.90 | 8.83 | 3.91 | 3.41 |
| RS+LMT | 11.41 | 9.15 | 50.38 | 53.30 | 38.07 | 36.80 |
| RS+NB | 0.16 | 0.15 | 0.39 | 0.65 | 0.37 | 0.36 |
| RS+SL | 4.50 | 3.77 | 36.40 | 20.39 | 12.50 | 11.73 |
| RS+SMO | 2.46 | 2.53 | 20.63 | 4.22 | 53.64 | 1.64 |
| RS+J48 | 0.25 | 0.21 | 0.84 | 1.78 | 0.81 | 0.71 |

which is also widely used in previous SDP research [19, 20, 21, 22].

Threats to internal validity are mainly concerned with the uncontrolled internal factors that might have influence on the experimental results. The main internal threat is the potential faults during our method implementation. To reduce this threat, we use implementation of baseline methods provided by Yang et al. [8]. For our proposed method MULTI, we use test cases to verify the correctness of our implementation and we use mature third-party libraries, such as packages from Matlab and R.

**Threats to construct validity** are about whether the performance metrics used in the empirical studies reflect the real-world situation. We used $P_{OPT}$ and $ACC$ to evaluate the performance of JIT-SDP models. These effort-aware metrics have been used in previous JIT-SDP research [5, 8, 9, 10]. In addition, we also use $F1$ [1, 7, 10, 9, 16, 33] to show the competitiveness of our proposed method.

## 6. Conclusion and Future Work

In this article, we firstly formalize JIT-SDP as a multi-objective optimization problem and then propose a novel method MULTI. Large-scale empirical studies show the competitiveness of our proposed method in cross-validation, cross- project-validation, and timewise-cross-validation scenarios. These results confirm that supervised methods are still promising in effort-aware JIT-SDP.

In the future, we plan to extend our research in several ways. Firstly we want to consider more commercial projects to verify whether our conclusion can be generalized. Secondly the quality of JIT-SDP datasets needs to be improved. For example, researchers often used SZZ algorithm [13] to identify buggy changes. However a recent study [40] shows that current SZZ implementations still lack mechanisms to accurately identify buggy changes. Finally we want to optimize the performance of our method by considering other MOAs (such as SPEA [39]) to implement MULTI method and select appropriate quality indicators to assess the quality of Pareto fronts generated by different MOAs

[38].

For other researchers to follow our work, we provide a package [1] to repeat our empirical studies. This package includes: (1) the datasets of the training set and the testing set. (2) the scripts for running MULTI. (3) The detailed results for our empirical studies.

**Acknowledgment**

The authors would like to thank the editors and the anonymous reviewers for their insightful comments and suggestions, which can substantially improve the quality of this work. This work is supported in part by National Natural Science Foundation of China (Grant Nos. 61202006, 61602267), Guangxi Key Laboratory of Trusted Software (Grant No. kx201610), and Open Project of State Key Laboratory for Novel Software Technology at Nanjing University (Grant No. KFKT2016B18).

**References**

[1] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Transactions on Software Engineering 38 (6) (2012) 1276–1304.

[2] Y. Kamei, E. Shihab, Defect prediction: Accomplishments and future challenges, in: Proceedings of International Conference on Software Analysis, Evolution, and Reengineering, 2016, pp. 33–45.

[3] M. D'Ambros, M. Lanza, R. Robbes, Evaluating defect prediction approaches: A benchmark and an extensive comparison, Empirical Software Engineering 17 (4-5) (2012) 531–577.

[1] https://github.com/Hecoz/MultiObjectResearch

[4] D. Radjenovic, M. Hericko, R. Torkar, A. Zivkovic, Software fault prediction metrics: A systematic literature review, Information and Software Technology 55 (8) (2013) 1397 – 1418.

[5] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, N. Ubayashi, A large-scale empirical study of just-in-time quality assurance, IEEE Transactions on Software Engineering 39 (6) (2013) 757–773.

[6] A. Mockus, D. M. Weiss, Predicting risk of software changes, Bell Labs Technical Journal 5 (2) (2000) 169–180.

[7] S. Kim, E. J. W. Jr., Y. Zhang, Classifying software changes: Clean or buggy?, IEEE Transactions on Software Engineering 34 (2) (2008) 181–196.

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

[9] X. Yang, D. Lo, X. Xia, Y. Zhang, J. Sun, Deep learning for just-in-time defect prediction, in: Proceedings of International Conference on Software Quality, Reliability and Security, 2015, pp. 17–26.

[10] X. Yang, D. Lo, X. Xia, J. Sun, Tlel: A two-layer ensemble learning approach for just-in-time defect prediction, Information and Software Technology 87 (2017) 206–220.

[11] M. Harman, The relationship between search based software engineering and predictive modeling, in: Proceedings of the International Conference on Predictive Models in Software Engineering, 2010, pp. 1:1–1:13.

[12] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: Nsga-ii, IEEE Transactions on Evolutionary Computation 6 (2) (2002) 182–197.

[13] J. Śliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes?, in: Proceedings of the International Workshop on Mining Software Repositories, 2005, pp. 1–5.

[14] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, A. E. Hassan, Studying just-in-time defect prediction using cross-project models, Empirical Software Engineering 21 (5) (2016) 2072–2106.

[15] W. Fu, T. Menzies, Revisiting unsupervised learning for defect prediction, in: Proceedings of The joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2017.

[16] S. Shivaji, E. J. Whitehead, R. Akella, S. Kim, Reducing features to improve code change-based bug prediction, IEEE Transactions on Software Engineering 39 (4) (2013) 552–569.

[17] G. Canfora, A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, S. Panichella, Defect prediction as a multi-objective optimization problem, Software Testing Verification and Reliability 25 (4) (2015) 426–459.

[18] M. Harman, S. A. Mansouri, Y. Zhang, Search-based software engineering: Trends, techniques and applications, ACM Computing Surveys 45 (1) (2012) 11:1–11:61.

[19] B. Ghotra, S. McIntosh, A. E. Hassan, Revisiting the impact of classification techniques on the performance of defect prediction models, in: Proceedings of the International Conference on Software Engineering, 2015, pp. 789–800.

[20] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: A proposed framework and novel findings, IEEE Transactions on Software Engineering 34 (4) (2008) 485–496.

[21] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, K. Matsumoto, Automated parameter optimization of classification techniques for defect prediction models, in: Proceedings of the International Conference on Software Engineering, 2016, pp. 321–332.

[22] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, K. Matsumoto, An empirical comparison of model validation techniques for defect prediction models, IEEE Transactions on Software Engineering 43 (1) (2017) 1–18.

[23] A. E. Hassan, Predicting faults using the complexity of code changes, in: Proceedings of the International Conference on Software Engineering, 2009, pp. 78–88.

[24] N. Nagappan, T. Ball, Use of relative code churn measures to predict system defect density, in: Proceedings of the International Conference on Software Engineering, 2005, pp. 284–292.

[25] R. Moser, W. Pedrycz, G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, in: Proceedings of the International Conference on Software Engineering, 2008, pp. 181–190.

[26] T. L. Graves, A. F. Karr, J. S. Marron, H. Siy, Predicting fault incidence using software change history, IEEE Transactions on Software Engineering 26 (7) (2000) 653–661.

[27] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, M. Nakamura, An analysis of developer metrics for fault prediction, in: Proceedings of the International Conference on Predictive Models in Software Engineering, 2010, pp. 18:1–18:9.

[28] E. Arisholm, L. C. Briand, E. B. Johannessen, A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, Journal of Systems and Software 83 (1) (2010) 2–17.

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

[30] T. J. Ostrand, E. J. Weyuker, R. M. Bell, Predicting the location and number of faults in large software systems, IEEE Transactions on Software Engineering 31 (4) (2005) 340–355.

[31] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, A. Bener, Defect prediction from static code features: Current results, limitations, new approaches, Automated Software Engineering 17 (4) (2010) 375–407.

[32] Y. Zhou, B. Xu, H. Leung, L. Chen, An in-depth study of the potentially confounding effect of class size in fault prediction, ACM Transactions on Software Engineering and Methodology 23 (1) (2014) 10:1–10:51.

[33] J. Nam, S. Kim, Clami: Defect prediction on unlabeled datasets, in: Proceedings of International Conference on Automated Software Engineering, 2015, pp. 452–463.

[34] F. Zhang, Q. Zheng, Y. Zou, A. E. Hassan, Cross-project defect prediction using a connectivity-based unsupervised classifier, in: Proceedings of the International Conference on Software Engineering, 2016, pp. 309–320.

[35] C. A. C. Coello, G. B. Lamont, D. A. V. Veldhuizen, Evolutionary Algorithms for Solving Multi-Objective Problems (Second Edition), Springer US, 2007.

[36] Y. Benjamini, Y. Hochberg, Controlling the false discovery rate: A practical and powerful approach to multiple testing, Journal of the Royal Statistical Society. Series B (Methodological) 57 (1) (1995) 289–300.

[37] E. Jelihovschi, J. Faria, I. Allaman, Scottknott: a package for performing the scott-knott clustering algorithm in r, TEMA (Sao Carlos) 15 (1) (2014) 3–17.

[38] S. Wang, S. Ali, T. Yue, Y. Li, M. Liaaen, A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering, in: Proceedings of the International Conference on Software Engineering, 2016, pp. 631–642.

[39] E. Zitzler, L. Thiele, Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach, IEEE Transactions on Evolutionary Computation 3 (4) (1999) 257–271.

[40] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, A. Hassan, A framework for evaluating the results of the szz approach for identifying bug-introducing changes, IEEE Transactions on Software Engineering 43 (7) (2017) 641 – 657.