2017 IEEE International Conference on Software Maintenance and Evolution

Supervised vs Unsupervised Models: A Holistic Look at Effort-Aware Just-in-Time Defect Prediction

Qiao Huang*, Xin Xia†✓, and David Lo‡
*College of Computer Science and Technology, Zhejiang University, China
†Department of Computer Science, University of British Columbia, Canada
‡School of Information Systems, Singapore Management University, Singapore
tkdsheep@zju.edu.cn, xxia02@cs.ubc.ca, davidlo@smu.edu.sg

***Abstract*—Effort-aware just-in-time (JIT) defect prediction aims at finding more defective software changes with limited code inspection cost. Traditionally, supervised models have been used; however, they require sufficient labelled training data, which is difficult to obtain, especially for new projects. Recently, Yang et al. proposed an unsupervised model (LT) and applied it to projects with rich historical bug data. Interestingly, they reported that, under the same inspection cost (i.e., 20 percent of the total lines of code modified by all changes), it could find more defective changes than a state-of-the-art supervised model (i.e., EALR). This is surprising as supervised models that benefit from historical data are expected to perform better than unsupervised ones. Their finding suggests that previous studies on defect prediction had made a simple problem too complex.**

Considering the potential high impact of Yang et al.'s work, in this paper, we perform a replication study and present the following new findings: (1) Under the same inspection budget, LT requires developers to inspect a large number of changes necessitating many more context switches. (2) Although LT finds more defective changes, many highly ranked changes are false alarms. These initial false alarms may negatively impact practitioners' patience and confidence. (3) LT does not outperform EALR when the harmonic mean of *Recall* and *Precision* (i.e., *F1-score*) is considered.

Aside from highlighting the above findings, we propose a simple but improved supervised model called CBS. When compared with EALR, CBS detects about 15% more defective changes and also significantly improves *Precision* and *F1-score*. When compared with LT, CBS achieves similar results in terms of *Recall*, but it significantly reduces context switches and false alarms before first success. Finally, we also discuss the implications of our findings for practitioners and researchers.

***Index Terms*—Change Classification, Cost Effectiveness, Evaluation, Bias**

I. Introduction

Defect prediction techniques aim to help developers prioritize testing and debugging effort by recommending likely defective code. Most defect prediction studies propose prediction models built on various types of features (e.g., process or code features), and predict defects at coarse granularity level, such as file, package, or module [1]–[6]. Mockus and Weiss [7] are the first to propose a prediction model which focuses on identifying defect-prone software changes instead of files or packages. Such prediction is also referred as *just-in-time (JIT) defect prediction* by Kamei et al. [8]. JIT defect prediction is more practical since (1) it leads to smaller amount of code to be reviewed[1], and (2) developers can review and test these risky changes while they are still fresh in their minds (i.e., at commit time).

Different changes would require different amount of effort to inspect, and intuitively, a change that modifies (i.e., adds or deletes) a larger number of lines of code (LOC) requires a developer to spend more effort to inspect it. Based on this intuition, *effort-aware JIT defect prediction* [8] takes into account the inspection cost of a change (measured by the number of modified LOC); a prediction model in this setting focuses on optimizing the number of defects that can be found given a fixed inspection budget (e.g., inspecting 20% LOC modified by all changes). Kamei et al. proposed a state-of-the-art supervised model called EALR which leveraged linear regression to help developers review changes more effectively given a fixed inspection budget [8]. They reported that the EALR model could identify 35% of all defective changes, when 20% LOC modified by all changes are inspected.

One disadvantage for supervised defect prediction models is that they require a large amount of labelled instances for training [9]. Unfortunately, it is difficult to get sufficient training data for a new project. To address this limitation, Yang et al. [10] proposed an unsupervised model for effort-aware JIT defect prediction, which simply sort the changes by one metric. Their idea is inspired by Koru et al.'s finding that smaller modules are proportionally more defect-prone and should be inspected first [11]. For example, consider a metric LT (i.e., lines of code in a file before a change); Yang et al. hypothesizes that changes with lower LT are in *smaller modules* and should be inspected earlier. By performing empirical study on the dataset published by Kamei et al. [8], they found that unsupervised model with the metric LT outperforms the state-of-the-art supervised model (i.e., the EALR model) in terms of *Recall*. Here *Recall* means the proportion of inspected defective changes among all defective changes.[2]

There are many advantages of the unsupervised model:

- It is straightforward to understand and much easier to implement.

✓Corresponding author.

[1]The amount of inspected code in an individual change is much less than the code in a file, package, or module.

[2]Some previous studies [12]–[14] also denoted this evaluation measure as *cost-effectiveness*.

- It does not require any labelled training data, or any machine learning techniques. Thus, it can be easily applied in a new project and runs much faster.
- Under the same inspection cost (i.e., 20% LOC), it can find more defects.

These advantages suggest that previous studies on defect prediction had made a simple problem too complex. This is a surprising finding, since intuitively, as a supervised model extracts prior knowledge (e.g., defect distribution, defective patterns) from historical changes, it is expected to perform better than a model which has no prior knowledge.

Considering the potential high impact of Yang et al.'s work, in this paper, we perform a replication study[3]. In particular, we would like to investigate why the unsupervised model achieves a high *Recall*. Additionally, to have a holistic view, we consider a number of additional metrics beyond *Recall* and use them as yardsticks to compare supervised and unsupervised models considered by Yang et al. Last but not least, we would like to boost the performance of a supervised model by leveraging the intuition underlying Yang et al.'s work.

Our study focuses on answering the following research questions:

RQ1: Why do Yang et al.'s unsupervised model (LT) perform better than Kamei et al.'s supervised model (EALR) in terms of *Recall*?

We explore the distribution of change size (i.e., LOC modified by the change) and find it highly skewed for every project. Most changes are small while a few are very large. Considering the same inspection cost (i.e., 20% LOC), on average, LT requires developers to inspect more than twice as many as the number of changes inspected when using EALR. Thus, it is of no surprise that Yang et al.'s unsupervised model finds more defects. However, it is not reasonable to expect developers to inspect too many changes due to the additional effort required for frequent context switches [15]. Additionally, inspecting many changes can map to a high number of false alarms which in turn may lead to developer fatigue and tool abandonment – c.f. [16], [17]

RQ2: How do the supervised and unsupervised models compare when different evaluation measures are considered?

We argue that, *Recall* cannot provide enough information to help practitioners fully evaluate a prediction model. Thus, we use 4 additional evaluation measures, namely *Precision*, *F1-score*, *PCI@20%* (i.e., Proportion of Changes Inspected when 20% LOC modified by all changes are inspected), and *IFA* (i.e., number of Initial False Alarms encountered before we find the first defect). We use *Recall*, *Precision* and *F1-score* because they are widely used in prior software engineering studies [13], [18]–[21]. We propose *PCI@20%* to measure the additional effort needed due to context switches between changes, since context switching has been shown harmful to developer productivity [15]. We propose *IFA* because previous studies [16], [17] have shown that developers are not willing to use a prediction model if the first few recommendations are all false alarms.

By replicating Yang et al.'s experiment with the same dataset but more evaluation measures, we find that LT does not outperform EALR considering these additional evaluation measures. In some projects, EALR even significantly outperforms LT considering some of these new evaluation measures.

RQ3: Could the supervised model be enhanced leveraging intuition of Yang et al.'s unsupervised model?

We propose a simple but improved supervised model called CBS. It first builds a logistic classifier to identify defective changes. Then it sorts the identified defective changes in ascending order by their inspection cost.

When compared with EALR, CBS significantly improves *Recall* for each project, and it also significantly improve both *Precision* and *F1-score* for 5 out of the 6 projects. When compared with Yang et al.'s unsupervised model (LT), it achieves similar results in terms of *Recall* in 4 out of 6 projects and significantly improves *Recall* in the other 2 projects. It also significantly reduces the amount of initial false alarms and the amount of changes required to inspect for each project.

The main contributions of our paper are as follows:

1) We perform an in-depth analysis of the experiment results in Yang et al.'s work, and analyze the reason why the unsupervised model outperforms supervised models in effort-aware JIT defect prediction.
2) We perform a holistic evaluation of supervised versus unsupervised models with two new considerations: context switches and developer fatigue due to initial false alarms. We present new findings and highlight limitations of unsupervised models that were not revealed by prior studies.
3) We propose a simple but improved supervised model called CBS. While CBS performs as well as Yang et al.'s unsupervised model (LT) in terms of *Recall*, it significantly outperforms LT in terms of the other evaluation measures.

Paper Organization. The remainder of the paper is organized as follows. We introduce the background and related work on JIT defect prediction in Section II. We describe the technical details of the supervised and unsupervised models proposed by previous studies (i.e., EALR and LT) in Section III. We introduce our improved supervised model in Section IV. We introduce the evaluation measures in Section V. We present our experimental setup and results in Section VI and VII, respectively. We discuss the implications for practitioners and researchers in Section VIII. We examine the threats to validity in Section IX. We conclude the paper and mention future work in Section X.

II. Background and Related Work

In this section, we introduce the background and related work of just-in-time (JIT) defect prediction and effort-aware JIT defect prediction.

[3]The replication package of our study can be downloaded at https://zenodo.org/record/836352#.WX2MlYiGOUk

A. Just-in-Time Defect Prediction

Traditional defect prediction models focus on identifying defect-prone classes, files or modules. Such granularity could be too coarse to be applied in practice. For example, a prediction model is more likely to recommend large files to developers for inspection, since defect proneness increases as file size increases [22]. However, it is difficult for developers to recall the logic and technical details hidden in the code when given a large file with thousands of lines of code. Also, it is difficult to decide which developer should be assigned to inspect the code since a large file often have multiple authors [23].

To address the above limitations, Mockus and Weiss are the first to study change-level defect prediction [7]. They proposed a supervised model to predict defects in a large-scale telecommunication system at the initial maintenance request (IMR) level, which consists of multiple changes. Their model used the properties of a change itself, such as lines of code added and deleted, diffusion of the change, measures of developer experience, etc. Kim et al. extracted text feature from various sources (i.e., change log, source code, and file names), and combined them with features extracted from change metadata and complexity metrics to build the prediction model for classifying clean or buggy changes [23]. Yin et al. empirically studied the incorrect bug-fixes from 4 large operating systems [24]. They found that at least 14.8% to 24.4% of fixes for post-release bugs are incorrect and affect end users. They also found that concurrency bugs are the most difficult to fix, and developers and reviewers of incorrect fixes usually do not have enough knowledge about the involved code. Shihab et al. performed an industrial study on the risk of software changes [25]. They found that the number of lines of code added by the change, the bugginess of the files being changed, the number of bug reports linked to a change and the developer experience are the best indicators of change risk. Kamei et al. [8] referred to the change-level defect prediction as *Just-in-Time Defect Prediction*, and they performed a large-scale empirical study on six open source projects and five commercial projects.

B. Effort-Aware JIT Defect Prediction

Previous studies [26]–[28] have pointed out that defect prediction model should also be *effort-aware*. For traditional defect prediction at file level, the number of lines of code in a file is used as a measure of the effort required to inspect the file [28], [29]. Kamei et al. [8] also evaluated the performance of defect prediction model at change level when considering inspection cost. They used the size of a change (i.e., total number of LOC added and deleted by the change) to measure the inspection cost of a change.

Considering tight development and release, and limited human resources, previous studies on *effort-aware* defect prediction focused on finding more defects with limited code inspection cost. Besides, previous studies [22], [30], [31] have shown that about 80% of the defects are contained in about 20% of the files. Motivated by these work, Kamei et al. assumed that the available resources only account for 20% of the effort it would take to inspect all changes, and they proposed a supervised model called EALR [8]. Instead of predicting defect-proneness, EALR would predict the defect-density for each change. Then it ranks changes in descending order by the predicted defect-density, and the top changes are inspected one by one until the accumulated inspection cost reaches the threshold of 20%. They used *Recall* (i.e., the proportion of inspected defective changes among all the defective changes) to evaluate the performance of the prediction under effort-aware setting.

TABLE I
SUMMARY OF CHANGE METRICS.

Metric	Description
NS	Number of subsystems touched by the current change
ND	Number of directories touched by the current change
NF	Number of files touched by the current change
Entropy	Distribution across the touched files
LA	Lines of code added by the current change
LD	Lines of code deleted by the current change
LT	Lines of code in a file before the current change
FIX	Whether or not the current change is a defect fix
NDEV	Number of developers that changed the files
AGE	Average time interval between the last and current change
NUC	Number of unique last changes to the files
EXP	Developers experience (number of files modified)
REXP	Developers experience in recent years
SEXP	Developer experience on a subsystem

More recently, Yang et al. [10] proposed an unsupervised model for effort-aware JIT defect prediction. Their model is based on the assumption that changes in smaller files should be inspected first, which is inspired by Koru et al.'s finding that smaller modules are proportionally more defect-prone and should be inspected first [11]. They reported that using the same data provided by Kamei et al. [8], their unsupervised model could achieve higher *Recall* when compared with supervised model. Following Yang et al.'s work, Yan et al. [32] applied the unsupervised model to effort-aware file-level defect prediction, and they found that the conclusion of Yang et al. does not hold under within-project setting for file-level defect prediction. Different from Yan et al.'s work, we focus on investigating why Yang et al.'s unsupervised model achieves high recall in effort-aware JIT defect prediction.

III. Effort-Aware JIT Defect Prediction Models

In this section, we introduce the technical details of the supervised model proposed by Kamei et al. [8], and unsupervised model proposed by Yang et al. [10] for effort-aware JIT defect prediction.

A. Supervised Model by Kamei et al. (EALR)

Kamei et al. considered 14 metrics derived from the source control repository data of a project to represent a change. Table I presents the name and description of each metric. These metrics can be grouped into five dimensions: diffusion (NS, ND, NF and Entropy), size (LA, LD and LT), purpose

(FIX), history (NDEV, AGE and NUC) and experience (EXP, REXP and SEXP).

The metrics in diffusion dimension characterize the distribution of a change. Previous studies showed that a highly distributed change is more likely to be defective [4], [7], [33], [34]. The metrics in size dimension characterize the size of a change, and a larger change is more likely to be defective since more code has to be changed or implemented [35], [36]. The purpose dimension only consists of FIX, and it is believed that a defect-fixing change is more likely to introduce a new defect [37]–[39]. The metrics in history dimension can tell us how developers interacted with different files in the past. As stated by Yang et al. [10], a change is more likely to be defective if the touched files have been modified by more developers [40], by more recent changes [37], or by more unique last changes [4], [33]. The experience dimension measures a developer's experience based on the number of changes made by the developer in the past. In general, a change made by a more experienced developer is less likely to introduce defects [7].

Based on these 14 metrics, Kamei et al. [8] built a logistic classifier learned from a training dataset to predict the risk score (i.e. defect-proneness) of new changes in the testing dataset. However, the score does not consider the inspection cost of each change, and the performance would be bad under the effort-aware setting [8]. To solve this problem, they proposed an effort-aware linear regression (EALR) model, which tries to learn the relationships between the various characteristic metrics of a change c (i.e., change metrics shown in Table I) and its *defect-density* $D(c)$ from the training dataset. The defect-density $D(c)$ is defined as follow:

$$D(c) = \frac{Y(c)}{\mathit{Effort}(c)} \tag{1}$$

Here $Y(c)$ is 1 if the change c is defective and 0 otherwise, and $\mathit{Effort}(c)$ is the amount of effort required to inspect the change.

Then the EALR model would predict the value of $D(c')$ for a new change c' in the testing dataset, and sort these changes in descending order by their risk scores. Note that they only use 12 metrics (excluding LA and LD) as independent variables to build the EALR model, since lines of code added/deleted (i.e., LA and LD) together make up the effort value in the dependent variable of EALR model [8].

In practice, it is difficult for a linear regression model to accurately predict the value of $D(c)$, which would negatively impact the performance of prediction. Kamei et al. [8] reported that the EALR model could detect 35% of all defective changes when developers inspect 20% of LOC modified by all changes.

B. Unsupervised Model by Yang et al. (LT)

More recently, Yang et al. [10] leveraged the same metrics in Kamei et al.'s work [8] to build an unsupervised model. The unsupervised model only uses one metric M among all the available metrics and sort the changes in descending order according to the reciprocal of M. More formally, given a change c and the metric value $M(c)$, the model would predict a risk score $R(c) = \frac{1}{M(c)}$. Changes will be sorted in descending order according to the predicted risk score. To follow Kamei et al.'s work [8], the unsupervised model also excluded LA and LD from the candidate metrics. Among all the other 12 candidate metrics, the unsupervised model with LT metric achieves the best performance in most cases, and it also significantly outperforms the EALR model in terms of *Recall*. Sorting based on LT follows Koru et al.'s finding, which reveals that smaller modules are proportionally more defect-prone and should be inspected first [11]. Thus, we also choose LT as the underlying metric for unsupervised model in our experiment.

IV. CBS: An Improved Supervised Model

In this section, we propose a simple but improved supervised model called CBS. We first introduce the motivation of CBS, then we present its technical details with a pseudocode.

The major problem of EALR is that the relationship between the change metrics and defect density (see Equation 1) may not be linear. Thus, it is difficult to accurately predict a specific value of defect-density using a linear model. However, as shown in Kamei et al.'s work [8], it is relatively easy to build a classifier to predict whether a change is defective or not. They reported that the classifier can find about 70% of all defective changes. To leverage the advantages of such a classifier, while benefiting from Koru et al.'s findings, we propose CBS (i.e., Classify-Before-Sorting). CBS assumes that among changes that are classified to be potentially buggy, small ones should be inspected first, since they give the best bang for the buck.

Algorithm 1 Pseudocode for Classifier Building

```
1: BuildClassifier($\mathit{TrainSet}$, $\mathit{Metrics}$)
2: **Input:**
3: $\mathit{TrainSet}$: Training set of changes
4: $\mathit{Metrics}$: Metrics (see Table I) of changes in TrainSet
5: **Output:**
6: $\mathit{Classifier}$: The classifier built on the training dataset
7: **Method:**
8: Re-sample $\mathit{TrainSet}$ to balance the number of defective
   and non-defective changes;
9: Remove ND, REXP, LA and LD from $\mathit{Metrics}$;
10: Apply standard logarithmic transformation to each metric
    in $\mathit{Metrics}$ except for FIX;
11: Build a classifier $\mathit{Logistic}$ by using logistic regression
    applied on $\mathit{TrainSet}$ and $\mathit{Metrics}$;
12: **return** $\mathit{Logistic}$;
```

Algorithm 1 presents the pseudo-code to build a classifier as proposed by Kamei et al. [8]. We first follow Kamei et al. to re-sample training data to deal with data imbalance (i.e., they randomly removed instances of the majority class until the training data is balanced) (Line 8). Then we remove several metrics (Line 9). Following Kamei et al., we remove the metrics ND and REXP, since they found that NF and ND, and REXP and EXP are highly correlated. Usage of highly correlated features may decrease classifier accuracy. We also remove the metrics LA and LD since they will be used for sorting. After that, we follow Kamei et al. to perform standard log transformation to several metrics (Line 10). Finally, we build a classifier by using logistic regression (Line 11).

Algorithm 2 Pseudocode for CBS

```
1: CBS($Logistic$, $TestSet$)
2: Input:
3: $Logistic$: The classifier built on training dataset
4: $TestSet$: Testing set of changes
5: Output:
6: $RankedList$: Ranked list of changes for inspection
7: Method:
8: $Defective, NonDefective = \varnothing$;
9: for all change $c \in TestSet$ do
10:    Use $Logistic$ to predict the label $l$ of change $c$;
11:    if $l$ is potentially defective then
12:       Add $c$ into $Defective$;
13:    else
14:       Add $c$ into $NonDefective$;
15:    end if
16: end for
17: Let $RankedList$ = Changes in $Defective$ sorted in ascending order of size (i.e., LA+LD);
18: return $RankedList$;
```

Algorithm 2 presents the pseudo-code of CBS. Using the classifier built on training dataset, we first identify potentially defective and non-defective changes in testing dataset (Lines 8-16). A change would be classified as potentially defective if its predicted score is larger than 0.5; otherwise it will be classified as potentially non-defective. Then we sort the predicted potentially defective changes in ascending order of their size (i.e., LA+LD) (Line 17). This ranked list of potentially defective changes is then returned (Line 18).

V. Evaluation Measures Considered

In this section, we introduce the following 5 evaluation measures used in our paper to evaluate the performance of both supervised and unsupervised models. Suppose we have a dataset with *M* changes and *N* defects. After inspecting 20% LOC, we inspected *m* changes and found *n* defects. Besides, when we find the first defective change, we have inspected k changes. Then the 5 evaluation measures are defined and computed as follows:

Recall: Proportion of inspected defective changes among all the actual defective changes. This is the evaluation measure used by many previous studies [8], [10], [41]–[43]. They focused on achieving high *Recall* so that more defective changes could be detected. *Recall* is computed as: n/N.

Precision: Proportion of inspected defective changes among all the inspected changes. A low *Precision* indicates that developers would encounter more false alarms, which may have negative impact on developers' confidence on the prediction model. *Precision* is computed as: n/m.

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*). In many cases, high *Recall* indicates the sacrifice of *Precision*, and vice versa [44]. Therefore, to fairly evaluate the prediction model, *F1-score* is also widely used in prior software engineering studies [13], [18]–[21]. Note that if all the inspected changes are not defective, then both *Precision* and *Recall* would be 0, and *F1-score* would be *NaN* (i.e., not a number) since it divides zero. In this case, we set *F1-score* to be 0 since the prediction model achieves the worst performance. *F1-score* is computed as: $\frac{2*Precision*Recall}{Precision+Recall}$.

PCI@20%: Proportion of Changes Inspected when 20% LOC modified by all changes are inspected. A high *PCI@k%* indicates that, under the same number of LOC to inspect, developers need to inspect more changes. Note that the definition of inspection cost in prior papers [8], [10] only considers the size of a change, and some problems may arise when a prediction model requires developers to inspect a large number of changes. Suppose Alpha team needs to review 10K changes where each change modifies only 1 LOC, and Delta team needs to review only 1 change while it modifies 10K LOC. The number of LOC that needs to be inspected by the two teams are the same (i.e., 10K LOC in total). However, developers in Alpha team would frequently switch between different changes and this may increase the actual time and effort spent. For example, Meyer et al. [15] conducted a survey with 379 professional software developers and they found that developers perceive their days as productive when they complete many or big tasks without significant interruptions or context switches. Also, a large number of changes may cover many different localities (e.g., hundreds of files and modules), thus requiring more coordination and communication between developers with different expertise. The additional effort required due to context switches and additional communication overhead among developers should not be ignored. To the best of our knowledge, this is the first paper that takes these factors into consideration to evaluate effort-aware JIT defect prediction models. *PCI@20%* is computed as: m/M.

IFA: Number of Initial False Alarms encountered before we find the first defect. Inspired by previous studies on fault localization [16], [17], [45], we assume that if the top-k changes recommended by the model are all *false alarms*, developers would be frustrated and are not likely to continue inspecting the other changes. For example, Parnin and Orso [16] investigated how developers use and benefit from automated debugging tools through a set of human studies. They found that developers would stop inspecting suspicious statements, and turn back to traditional debugging, if they couldn't get promising results within the first few statements they inspect. *IFA* is computed as: k.

VI. Experiment Setup

In this section, we first describe the statistics of our dataset. Then we introduce the experiment setting. Finally, we present the motivation of our research questions.

TABLE II
STATISTICS OF THE STUDIED PROJECTS.

Project	Period	Language	# of Changes	% of Defects	Mean LOC per change	# of changes per day	# of modified files per change
Bugzilla	08/1998-12/2006	Perl	4,620	36%	37.5	1.5	2.3
Columba	11/2002-07/2006	Java	4,455	31%	149.4	3.3	6.2
Eclipse JDT	05/2001-12/2007	Java	35,386	14%	71.4	14.7	4.3
Eclipse Platform	20/2001-12/2007	Java	64,250	14%	72.2	26.7	4.3
Mozilla	01/2000-12/2006	C++	98,275	5%	106.5	38.9	5.3
PostgreSQL	07/1996-05/2010	Ruby	20,431	25%	101.3	4.0	4.5

A. Data Statistics

Table II summarizes the statistics of the studied projects. This dataset is published by Kamei et al. [8], and also used in Yang et al.'s work [10]. We can see that the changes of each project are gathered in a long period of time, written in different programming languages. The number of changes in each project ranges between 4,455 and 98,275. A change is labeled as defective if it induces one or more defect. For each project, only a small percentage of all changes are defective (about 5% to 36%).

B. Experiment Setting

To evaluate the prediction model, for each project, we follow Yang et al. [10] to use the *time-wise-cross-validation* defined in their work. Specifically, we first sort all changes in chronological order according to the commit date. Then we gather the changes submitted in the same month into one group. Suppose we have N groups of changes in a project, we use changes in group i and group i+1 ($1 \leq i \leq N-5$) as training data to build the supervised model. Then we use changes in group i+4 and group i+5 as testing data to evaluate both supervised and unsupervised model. As stated by Yang et al. [10], they chose the period of two consecutive months because the release cycle of most projects is typically 6 to 8 weeks. Besides, using two consecutive months guarantees each training set will have enough instances for building supervised models, and also allows us to have enough runs for each project. Note that we use *time-wise-cross-validation* instead of 10-fold cross-validation, since 10-fold cross-validation cannot guarantee the changes for testing are always created later than changes for training. In real application, we cannot use data in the future to build the supervised model and predict the data in the past.

Finally, since there are multiple runs for each project, we apply the Wilcoxon signed-rank test [46] with Bonferroni correction [47] at 95% significance level on two competing models. We consider that one model performs significantly better than the other model at the confidence level of 95% if the corresponding Wilcoxon signed-rank test result (i.e., p-value) is less than 0.05. We also use the Cliff's delta (δ) [48] to quantify the amount of difference between two approaches. The amount of difference is considered negligable ($\mid \delta \mid < 0.147$), small ($0.147 \leq \mid \delta \mid < 0.33$), moderate ($0.33 \leq \mid \delta \mid < 0.474$), or large ($\mid \delta \mid \geq 0.474$).

C. Research Questions

We investigate the following three research questions:

TABLE III
THE MEDIAN NUMBER OF CHANGES TO BE INSPECTED GIVEN THE SAME LOC BUDGET (20%) WHEN USING THE SUPERVISED MODEL (EALR) AND UNSUPERVISED MODEL (LT)

Model	BUG	COL	JDT	PLA	MOZ	POS	**AVG**
EALR	24	66	290	411	460	89	223
LT	36	125	568	963	1245	157	516

RQ1: Why do Yang et al.'s unsupervised model (LT) perform better than Kamei et al.'s supervised model (EALR) in terms of *Recall*?

In intuition, supervised models extract prior knowledge from historical changes, and intuitively are likely to perform better than unsupervised models which have no prior knowledge. Thus, we are interested to explore the reason why the unsupervised model in Yang et al.'s work [10] could outperform supervised models in terms of *Recall*.

RQ2: How do the supervised and unsupervised models compare when different evaluation measures are considered?

Yang et al. [10] used *Recall* to evaluate a prediction model when using 20% effort. However, *Recall* does not consider the number of false alarms and context switches. False alarms may negatively impact developers' patience and confidence, while context switches may reduce developers' productivity. Thus, we argue that more evaluation measures should be used to assess defect prediction models. To gain more insights, in addition to *Recall*, we use another 4 evaluation measures, namely *Precision*, *F1-score*, *PCI@20%* and *IFA*, which have been introduced in Section V. Using these additional evaluation measures, we can compare the supervised and unsupervised models more comprehensively.

RQ3: Could the supervised model be enhanced leveraging intuition of Yang et al.'s unsupervised model?

Based on the intuition that defective changes with smaller sizes should be inspected first, we propose a simple but improved supervised model called CBS. We first compare it with EALR to investigate whether it achieves better performance in terms of different evaluation measures. Then we compare it with Yang et al.'s unsupervised model.

VII. EXPERIMENT RESULTS

A. RQ1: Why do Yang et al.'s unsupervised model (LT) perform better than Kamei et al.'s supervised model (EALR) in terms of Recall?

To answer this RQ, we investigate two specific sub-questions. The first sub-question explores the distribution of

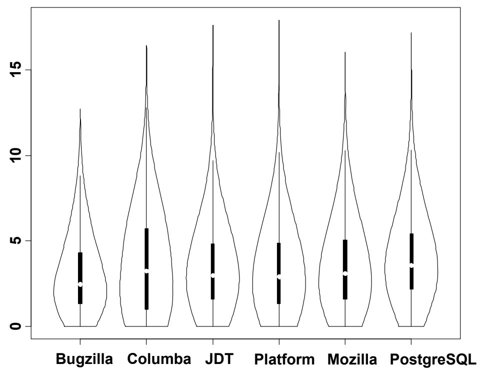

Fig. 1. Distribution of change size in each project.

change size in each project. Our preliminary experiment found that many changes in the dataset only modify a few or even zero lines of code (e.g., a change that only renames a file). On the other hand, some other changes modify a large number of LOC. For example, in the Columba project, the largest change modifies about 87K LOC, which accounts for about 13% of the total LOC (i.e., 665K) in the whole project. Since the unsupervised model prefers small changes (i.e., changes with small LT are not likely to modify too many LOC), it is possible that a large number of small changes would be inspected while the requirement of low inspection cost is still satisfied. The second sub-question investigates the number of changes required to inspect when using the supervised model and unsupervised model. We would like to see whether the unsupervised model requires developers to inspect more changes.

Question 1: What is the distribution of change size in each project?

To gain an overview of the distribution of change size (i.e., LOC added and deleted) in each project, we use violin plot [49] to visualize it. Since absolute values of some changes' sizes are quite huge (e.g., more than 100K LOC are modified), we applied a standard log transformation (base 2) to the size of each change before visualization.

Figure 1 presents the visualization results. The results show that, for each project, the majority of changes only modify a small number of LOC. Specifically, the sizes of most changes are less than 1000 (i.e., 2^{10}) LOC. On the other hand, a small number of changes modify a huge number of LOC (e.g., 2^{15} = 32K LOC), and they account for the majority of LOC modified in total. Thus, it is clear that the distribution of change size in each project is highly skewed.

Question 2: Given the same LOC budget, how many changes do the unsupervised and supervised model require developers to inspect?

Based on the observations above, we would like to validate whether Yang et al.'s unsupervised model (LT) requires developers to inspect more changes than Kamei et al.'s supervised model (EALR) when 20% LOC are inspected. Table III shows the median number of changes inspected when using EALR and LT. On average across the six projects, EALR requires developers to inspect 223 changes, while LT requires developers to inspect 516 changes. Since LT requires developers to inspect more than twice as many changes as those required by using EALR, it is of no surprise that LT can find more defective changes. On the other hand, LT does not proportionally increase the number of inspected defective changes. According to the results presented in Yang et al.'s work, on average, LT can find 43% of all defective changes, while EALR can find 31% of all defective changes. Finally, Yang et al. did not discuss the negative impact of the large number of false alarms and context switches during the manual inspection process. We further discuss this concern in RQ2.

The distribution of change size in every project is highly skewed. LT leverages this property to achieve higher Recall by requiring developers to inspect more than twice as many changes as those required by using EALR. However, problems may arise due to context switches and false alarms.

B. RQ2: How do the supervised and unsupervised models compare when different evaluation measures are considered?

Table IV presents the median results of Kamei et al.'s supervised model (EALR) and Yang et al.'s unsupervised model (LT) in terms of all evaluation measures (i.e., *Precision*, *Recall*, *F1-score*, *PCI@20%* and *IFA*) when inspecting 20% LOC. We choose to present median results following Yang et al. To follow Yang et al.'s work [10], we regard one prediction model in a project as a winner in terms of a certain evaluation measure, if it significantly outperforms the other model with a *moderate* or *large* improvement in terms of the Cliff's delta. The evaluation result of a winner is also marked by "$\surd$". The row "AVG" reports the average median over the six projects. The row "W/T/L" reports the number of projects for which the corresponding prediction model obtains a better, equal, and worse performance than the other model.

The results show that LT wins in terms of *Recall* in 5 out of the 6 projects, which is consistent with the results presented in Yang et al.'s work. On the other hand, EALR wins in terms of *Precision* in 5 out of the 6 projects. When considering *Recall* and *Precision* together (i.e., *F1-score*), the differences between the results of LT and EALR in every project are small or even negligible. As for *PCI@20%* and *IFA*, EALR significantly outperforms LT in at least 5 out of the 6 projects, which suggests that when using EALR, developers could be able to focus on a smaller number of changes and succeed in finding the first defective change earlier. We also notice that the *IFA* of unsupervised model is quite large. On average across the six projects, the top-70 changes recommended by unsupervised model are all false alarms. According to the survey [17] about practitioners' expectations on automated fault localization, it is not acceptable to use the automatic tool if the first **10** suspicious program elements are all false alarms. Thus, we doubt whether the unsupervised model is practical in use. As a comparison, the *IFA* of supervised model is much more reasonable. For every project, supervised model

TABLE IV
MEDIAN RESULTS OF EALR AND LT IN TERMS OF DIFFERENT EVALUATION MEASURES WHEN INSPECTING 20% LOC

Project	Recall			Precision			F1-score			PCI@20%			IFA		
	EALR	LT	$\|\delta\|$	EALR	LT	$\|\delta\|$	EALR	LT	$\|\delta\|$	EALR	LT	$\|\delta\|$	EALR	LT	$\|\delta\|$
BUG	0.299	0.449✓	0.41 (M)	0.364	0.333	-	0.325	0.378	-	0.312✓	0.516	0.52 (L)	3.5	5	-
COL	0.400	0.440	-	0.250✓	0.190	0.35 (M)	0.299	0.265	-	0.440✓	0.677	0.88 (L)	2✓	24	0.90 (L)
JDT	0.347	0.452✓	0.56 (L)	0.155✓	0.112	0.43 (M)	0.210	0.181	-	0.345✓	0.611	0.89 (L)	5✓	49	0.84 (L)
PLA	0.290	0.432✓	0.53 (L)	0.157✓	0.110	0.42 (M)	0.198	0.178	-	0.295✓	0.590	0.81 (L)	1✓	144	0.98 (L)
MOZ	0.190	0.363✓	0.77 (L)	0.045✓	0.035	0.33 (M)	0.072	0.062	-	0.232✓	0.554	0.93 (L)	8✓	185	0.82 (L)
POS	0.331	0.432✓	0.41 (M)	0.235✓	0.176	0.33 (M)	0.255	0.246	-	0.373✓	0.647	0.75 (L)	5✓	13	0.45 (M)
AVG	0.310	0.428	-	0.201	0.159	-	0.227	0.218	-	0.333	0.599	-	4.1	70	-
W/T/L	0/1/5	5/1/0	-	5/1/0	0/1/5	-	0/6/0	0/6/0	-	6/0/0	0/0/6	-	5/1/0	0/1/5	-

TABLE V
MEDIAN RESULTS OF EALR AND LT IN TERMS OF DIFFERENT EVALUATION MEASURES WHEN INSPECTING 5% LOC

Project	Recall			Precision			F1-score			PCI@20%			IFA		
	EALR	LT	$\|\delta\|$	EALR	LT	$\|\delta\|$	EALR	LT	$\|\delta\|$	EALR	LT	$\|\delta\|$	EALR	LT	$\|\delta\|$
BUG	0.095	0.171✓	0.39 (M)	0.333	0.330	-	0.144	0.215	-	0.106✓	0.219	0.46 (M)	3✓	5	0.34 (M)
COL	0.200	0.195	-	0.293✓	0.156	0.53 (L)	0.232	0.171	-	0.203✓	0.398	0.70 (L)	2✓	23	0.98 (L)
JDT	0.146	0.175	-	0.160✓	0.097	0.56 (L)	0.144	0.123	-	0.139✓	0.260	0.85 (L)	5✓	49	0.89 (L)
PLA	0.102	0.150	-	0.240✓	0.071	0.79 (L)	0.134✓	0.089	0.36 (M)	0.063✓	0.288	0.79 (L)	1✓	129	0.94 (L)
MOZ	0.053	0.115✓	0.48 (L)	0.069✓	0.021	0.57 (L)	0.057	0.036	-	0.042✓	0.265	0.85 (L)	8✓	172	0.92 (L)
POS	0.127	0.165	-	0.227✓	0.158	0.46 (M)	0.160	0.154	-	0.147✓	0.293	0.69 (L)	5✓	13	0.47 (M)
AVG	0.121	0.162	-	0.220	0.139	-	0.145	0.131	-	0.117	0.287	-	4	65.2	-
W/T/L	0/4/2	2/4/0	-	5/1/0	0/1/5	-	1/5/0	0/5/1	-	6/0/0	0/0/6	-	6/0/0	0/0/6	-

can find the first defective change when inspecting the top-10 suspicious changes.

Note that Yang et al. [10] and Kamei et al. [8] selected 20% LOC as the cut-off value. However, selecting another cut-off value might lead to different results. Besides, in our dataset, inspecting 20% LOC still requires developers to inspect a large amount of LOC. For example, in Mozilla project, 20% LOC corresponds to a total number of about 80K LOC. Thus, we investigate the performance of EALR and LT when inspecting less LOC (i.e., 5% LOC). Table V presents the comparison results when inspecting 5% LOC. The results are similar to those in Table IV. One difference is that, when inspecting 5% LOC, LT only wins in terms of *Recall* in 2 out of the 6 projects, and EALR additionally wins in terms of *F1-score* in 1 out of the 6 projects.

Yang et al.'s unsupervised model (LT) sacrifices Precision to achieve higher Recall. When considering Precision and Recall together (i.e., F1-score), unsupervised model no longer outperforms supervised model. Also, LT performs poorly in terms of IFA, which may negatively impact developers' patience and confidence.

C. RQ3: Could the supervised model be enhanced leveraging intuition of Yang et al.'s unsupervised model?

Table VI compares the evaluation results of CBS with EALR. On average across the six projects, CBS could find about 46% of all defective changes when inspecting 20% LOC, which significantly outperforms EALR in terms of *Recall* with an average improvement of 47%. CBS also significantly outperforms EALR in terms of *Precision* and *F1-score* in at least 5 out of the 6 projects. The *PCI@20%* of CBS is close to that of EALR for each project, which suggests that CBS does not require inspecting additional changes. However, CBS does not achieve the expected *IFA* (i.e., no more than 10) in 3 out of the 6 projects.

Table VII compares the evaluation results of CBS with Yang et al.'s unsupervised model (i.e., LT). CBS and LT achieve similar results in terms of *Recall* in 4 out the 6 projects, while CBS significantly improves *Recall* in the other 2 projects. When considering *Precision*, *F1-score* and *PCI@20%*, CBS significantly outperforms LT in at least 5 out of the 6 projects. This suggests that CBS significantly reduces the amount of false alarms and the amount of changes required to inspect for each project.

CBS can find more defective changes than EALR, and also significantly improves Precision and F1-score. When compared with Yang et al.'s unsupervised model, CBS achieves similar results in terms of Recall, but it significantly reduces context switches and false alarms before first success.

VIII. DISCUSSION

A. Comparison with Fu and Menzies's Work

Most recently, Fu and Menzies [50] also revisited unsupervised model in effort-aware JIT defect prediction. They replicated Yang et al.'s work [10] and pointed out that supervised model performs better in terms of *Precision* and *F1-score*. They also proposed a supervised model called *OneWay*, which leverages the training data to automatically choose the best metric for the unsupervised model in Yang et al.'s work.

Compared with their work, we present more findings considering additional perspectives, as shown below:

1) We investigate why the unsupervised model performs better in terms of *Recall*. We point out that the distribution of change size in every project is highly skewed. The unsupervised model leverages this property to achieve higher *Recall* by requiring developers to inspect more than twice as many changes as those required by using EALR.
2) We propose 2 additional evaluation measures (i.e., *PCI@20%* and *IFA*), which considers the negative impact of frequent context switches between different changes, and developer fatigue leading to likelihood of tool aban-

TABLE VI
COMPARISON OF THE PERFORMANCE ACHIEVED BY EALR AND CBS WHEN INSPECTING 20% LOC

Project	Recall			Precision			F1-score			PCI@20%			IFA		
	EALR	CBS	$\|\delta\|$	EALR	CBS	$\|\delta\|$	EALR	CBS	$\|\delta\|$	EALR	CBS	$\|\delta\|$	EALR	CBS	$\|\delta\|$
BUG	0.299	0.438✓	0.43 (M)	0.364	0.473✓	0.39 (M)	0.325	0.442✓	0.48 (L)	0.312	0.368	-	3.5	4	-
COL	0.400	0.464✓	0.35 (M)	0.250	0.352✓	0.34 (M)	0.299	0.390✓	0.44 (M)	0.440	0.364	-	2✓	9	0.56 (L)
JDT	0.347	0.453✓	0.49 (L)	0.155	0.216✓	0.49 (L)	0.210	0.297✓	0.60 (L)	0.345	0.302	-	5✓	14	0.45 (M)
PLA	0.290	0.515✓	0.79 (L)	0.157	0.207	-	0.198	0.304✓	0.53 (L)	0.295	0.369	-	1✓	22	0.83 (L)
MOZ	0.190	0.435✓	0.94 (L)	0.045	0.098✓	0.66 (L)	0.072	0.156✓	0.77 (L)	0.232	0.252	-	8✓	34	0.45 (M)
POS	0.331	0.444✓	0.51 (L)	0.235	0.376✓	0.54 (L)	0.255	0.387✓	0.72 (L)	0.373	0.321	-	5	7	-
AVG	0.310	0.458	-	0.201	0.287	-	0.227	0.329	-	0.333	0.329	-	4.1	15	-
W/T/L	0/0/6	6/0/0	-	0/1/5	5/1/0	-	0/0/6	6/0/0	-	0/6/0	0/6/0	-	4/2/0	0/2/4	-

TABLE VII
COMPARISON OF THE PERFORMANCE ACHIEVED BY LT AND CBS WHEN INSPECTING 20% LOC

Project	Recall			Precision			F1-score			PCI@20%			IFA		
	LT	CBS	$\|\delta\|$	LT	CBS	$\|\delta\|$	LT	CBS	$\|\delta\|$	LT	CBS	$\|\delta\|$	LT	CBS	$\|\delta\|$
BUG	0.449	0.438	-	0.333	0.473✓	0.43 (M)	0.378	0.442	-	0.516	0.368✓	0.40 (M)	5	4	-
COL	0.440	0.464	-	0.190	0.352✓	0.61 (L)	0.265	0.390✓	0.48 (L)	0.677	0.364✓	0.88 (L)	24	9✓	0.70 (L)
JDT	0.452	0.453	-	0.112	0.216✓	0.75 (L)	0.181	0.297✓	0.70 (L)	0.611	0.302✓	1.00 (L)	49	14✓	0.74 (L)
PLA	0.432	0.515✓	0.50 (L)	0.110	0.207✓	0.67 (L)	0.178	0.304✓	0.66 (L)	0.590	0.369✓	0.87 (L)	144	22✓	0.92 (L)
MOZ	0.363	0.435✓	0.44 (M)	0.035	0.098✓	0.84 (L)	0.062	0.156✓	0.84 (L)	0.554	0.252✓	0.99 (L)	185	34✓	0.82 (L)
POS	0.432	0.444	-	0.176	0.376✓	0.78 (L)	0.246	0.387✓	0.64 (L)	0.647	0.321✓	0.91 (L)	13	7	-
AVG	0.428	0.458	-	0.159	0.287	-	0.218	0.329	-	0.599	0.329	-	70	15	-
W/T/L	0/4/2	2/4/0	-	0/0/6	6/0/0	-	0/1/5	5/1/0	-	0/0/6	6/0/0	-	0/2/4	4/2/0	-

TABLE VIII
COMPARISON OF THE PERFORMANCE ACHIEVED BY CBS AND ONEWAY WHEN INSPECTING 20% LOC

Project	Recall			Precision			F1-score			PCI@20%			IFA		
	CBS	OW	$\|\delta\|$	CBS	OW	$\|\delta\|$	CBS	OW	$\|\delta\|$	CBS	OW	$\|\delta\|$	CBS	OW	$\|\delta\|$
BUG	0.438	0.362	-	0.473✓	0.394	0.34 (M)	0.442	0.369	-	0.368	0.396	-	4	6	-
COL	0.464	0.561	-	0.352✓	0.227	0.53 (L)	0.390✓	0.315	0.37 (M)	0.364✓	0.649	0.73 (L)	9✓	31	0.68 (L)
JDT	0.453	0.422	-	0.216✓	0.117	0.71 (L)	0.297✓	0.183	0.69 (L)	0.302✓	0.553	0.70 (L)	14✓	52	0.67 (L)
PLA	0.515✓	0.407	0.59 (L)	0.207✓	0.110	0.65 (L)	0.304✓	0.167	0.67 (L)	0.369✓	0.537	0.66 (L)	22✓	238	0.80 (L)
MOZ	0.435✓	0.327	0.60 (L)	0.098✓	0.041	0.72 (L)	0.156✓	0.074	0.74 (L)	0.252✓	0.449	0.58 (L)	34✓	157	0.55 (L)
POS	0.444	0.451	-	0.376✓	0.224	0.69 (L)	0.387✓	0.294	0.57 (L)	0.321✓	0.555	0.69 (L)	7✓	18	0.49 (L)
AVG	0.458	0.422	-	0.287	0.186	-	0.329	0.234	-	0.329	0.523	-	15	83.7	-
W/T/L	2/4/0	0/2/4	-	6/0/0	0/0/6	-	5/1/0	0/1/5	-	5/1/0	0/1/5	-	5/1/0	0/1/5	-

donment due to occurrences of many false alarms before success (i.e., a buggy change is identified).

We also compare our approach CBS with OneWay. Table VIII presents the comparison results of CBS and OneWay (denoted as "OW" in the table) when inspecting 20% LOC. The results show that, CBS wins in terms of *Recall* in 2 out of the 6 projects and ties with OneWay in the other 4 projects, which suggests that OneWay cannot find more defects than CBS. CBS significantly outperforms OneWay in terms of *Precision*, *F1-score* and *IFA* in at least 5 out of the 6 projects, which suggests that CBS can significantly reduce the number of false alarms before first success. Finally, compared with CBS, OneWay would require developers to inspect about 20% more changes on average across the six projects, which translates a large number of additional context switches.

B. Implications

1) Implications For Practitioners: Our experiment results have shown that, in most cases, unsupervised model performs better in terms of *Recall*, while supervised model performs better in terms of *Precision*. Although we can use *F1-score* to balance between *Precision* and *Recall*, the importance of *Precision* and *Recall* are not always the same in different projects.

For example, if the recommended changes are separately assigned to a large group of developers, the number of false alarms encountered by each developer would be significantly reduced. Thus, *Recall* is likely to be more important than *Precision* in this scenario, since a prediction model with high *Recall* can detect more defective changes. On the other hand, if the recommended changes are assigned to a few developers only, the negative impact of false alarms on developers' patience and confidence should be carefully considered. In this scenario, the importance of *Precision* should be weighted more than *Recall*.

In summary, we suggest developers use different measures to evaluate a prediction model more comprehensively, and choose the most appropriate model according to the requirement, schedule and resources in their own project.

2) Implications For Researchers: Both the studies by Yang et al. [10] and Kamei et al. [8] assumed that the inspection cost of a change is linearly associated with the change's size (i.e., the number of modified LOC). However, we have found some changes in the dataset which modified thousands of LOC. The actual effort required to inspect such a large change may not be linearly correlated with change size. For example, some changes only add a common comment (e.g., copyright) to a large number of files, and the amount of time and effort to inspect such changes is likely to be low. Thus, we argue that more factors (e.g., change type) should be considered to decide the inspection cost of a change. We recommend future research to do an empirical study on which additional factors influence the amount of time and effort needed to inspect a change, and how to determine the weights of different factors.

We also encourage future research on effort-aware JIT defect prediction to consider context switch cost and initial false alarms in evaluating the proposed solutions.

IX. Threats To Validity

A. Internal Validity

The internal validity relates to errors in our code when replicating the supervised and unsupervised model, which are both published by their authors using R language. Although our code is written in Java, we have carefully read the published source code and strictly follow the implementation. Since we use the same experiment setting as Yang et al.'s work, we compare our experiment results with theirs. For supervised model, our results are slightly different from those in [10]. Specifically, for each project, the differences between Yang et al.'s results and ours in terms of *Recall* are no more than 0.02. We argue that small difference is acceptable since supervised model requires data preprocessing and introduces random numbers. For unsupervised model, we reproduced the same experiment results since it is straightforward to implement. Thus, we believe there is little threat to internal validity.

B. External Validity

The external validity relates to the quality and generalizability of our dataset. We use six open source projects, which belong to different application domains, vary in size, cover a long period of time and are written in different programming languages. In total, we have analyzed 227,417 changes. However, there are still many other projects in other domains using other programming languages, which are not considered in our study. Besides, all the six projects in our study are developed by open source communities, it is still unclear whether our conclusion is generalizable to commercial projects. In the future, we plan to reduce this threat further by analyzing even more changes from additional software projects.

C. Construct Validity

The construct validity relates to the suitability of our evaluation measures. In addition to *Recall*, we use 4 evaluation measures, namely *Precision*, *F1-score*, *PCI@20%* and *IFA*. We use *Precision* because *Recall* and *Precision* are usually paired. We use *F1-score* because it balances the tradeoff between *Precision* and *Recall*. Also, *F1-score* is widely used in prior software engineering studies [13], [18]–[21]. We use *PCI@20%* because we find that the distribution of change size in every project is highly skewed, and we argue that inspecting too many changes would introduce additional effort cost. We use *IFA* because previous studies have shown that developers are not willing to use the prediction model if its *IFA* is quite large. Since we have carefully discussed the motivation of using these additional evaluation measures and cited previous studies to support our assumptions, we believe this construct validity should be acceptable.

Another threat to construct validity relates to the underlying metric we choose for Yang et al.'s unsupervised model. We choose the metric LT since it achieves the best average *Recall* in Yang et al.'s paper. However, another metric AGE also achieves similar *Recall*. We re-run our experiment with AGE-based unsupervised model and find that our conclusion remains the same, and thus not interesting to report.

Finally, as recent studies [51], [52] pointed out, the experimental design may also impact the conclusion of our paper. First, our results rely on time-wise cross validation scenario and we only choose two consecutive months as the time slots in order to follow Yang et al.'s work [10]. However, different validation techniques (e.g., choosing other time slots) may produce different performance estimates [53]. Second, we only use logistic regression as the underlying classification technique for supervised models in order to follow Kamei et al.'s work [8]. However, recent studies pointed out that defect prediction models with different classification techniques or different parameter settings may yield different results [54], [55]. Last but not the least, we do not consider possible mislabelling in our dataset – Tantithamthavorn et al. [56] pointed out that noise may creep into defect datasets and impact the recall achieved by defect prediction models. To reduce these threats to experimental design, as future work, we plan to conduct more experiments using additional datasets, different validation techniques and different classification techniques.

X. Conclusion and Future Work

In this paper, we revisit Yang et al.'s recent study on supervised versus unsupervised models in effort-aware JIT defect prediction. We first highlight that it is of no surprise that Yang et al.'s unsupervised model (LT) can find more defects, since it requires developers to inspect more than twice as many changes as those required by using Kamei et al.'s supervised model (EALR). We point out that inspecting too many changes would introduce additional effort due to frequent context switches. Then we use 4 additional evaluation measures to gain more insights of a prediction model. We find that LT sacrifices *Precision* to achieve higher *Recall*, and it no longer outperforms EALR when considering *Recall* and *Precision* together (i.e., *F1-score*). We also point out that, when using LT, developers may feel frustrated due to the large number of initial false alarms. Finally, we propose a simple but improved supervised model called CBS. When compared with Yang et al.'s unsupervised model, CBS achieves similar results in terms of *Recall*, but it performs significantly better in terms of *Precision* and *F1-score*. CBS also significantly reduces context switches and initial false alarms.

In the future, we plan to conduct a user study to investigate the actual effort required to inspect different types of changes. We are also interested to investigate the performance of supervised and unsupervised models in commercial projects.

Acknowledgment. We would like to thank Kamei et al. [8] and Yang et al. [10] for providing us the datasets and source code used in their study. This work was partially supported by NSFC Program (No. 61602403 and 61572426), and National Key Technology R&D Program of the Ministry of Science and Technology of China (No. 2015BAH17F01).

References

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

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

[3] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software engineering*, vol. 31, no. 10, pp. 897–910, 2005.

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

[5] P. L. Li, J. Herbsleb, M. Shaw, and B. Robinson, "Experiences and results from initiating field defect prediction and product test prioritization efforts at abb inc." in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 413–422.

[6] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 423–433, 1992.

[7] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

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

[9] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 91–100.

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

[11] G. Koru, H. Liu, D. Zhang, and K. El Emam, "Testing the theory of relative defect proneness for closed-source software," *Empirical Software Engineering*, vol. 15, no. 6, pp. 577–598, 2010.

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

[13] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 279–289.

[14] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 432–441.

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

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

[17] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 165–176.

[18] E. Arisholm, L. C. Briand, and M. Fuglerud, "Data mining techniques for building fault-proneness models in telecom java software," in *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*. IEEE, 2007, pp. 215–224.

[19] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the imprecision of cross-project defect prediction," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 61.

[20] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Studying re-opened bugs in open source software," *Empirical Software Engineering*, vol. 18, no. 5, pp. 1005–1042, 2013.

[21] H. Valdivia Garcia and E. Shihab, "Characterizing and predicting blocking bugs in open source projects," in *Proceedings of the 11th working conference on mining software repositories*. ACM, 2014, pp. 72–81.

[22] A. G. Koru, D. Zhang, K. El Emam, and H. Liu, "An investigation into the functional form of the size-defect relationship for software modules," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 293–304, 2009.

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

[24] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 26–36.

[25] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 62.

[26] T. Menzies and J. S. Di Stefano, "How good is your blind spot sampling policy," in *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*. IEEE, 2004, pp. 129–138.

[27] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE, 2010, pp. 107–116.

[28] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *Journal of Systems and Software*, vol. 83, no. 1, pp. 2–17, 2010.

[29] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010.

[30] M. Hamill and K. Goseva-Popstojanova, "Common trends in software fault and failure data," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 484–496, 2009.

[31] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4. ACM, 2004, pp. 86–96.

[32] M. Yan, Y. Fang, D. Lo, X. Xia, and X. Zhang, "File-level defect prediction: Unsupervised vs. supervised models," in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2017, p. to appear.

[33] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 31–41.

[34] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 452–461.

[35] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 181–190.

[36] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE, 2005, pp. 284–292.

[37] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on software engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[38] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1. IEEE, 2010, pp. 495–504.

[39] R. Purushothaman and D. E. Perry, "Toward understanding the rhetoric of small source code changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 511–526, 2005.

[40] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM, 2010, p. 18.

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

[42] X. Xia, D. Lo, X. Wang, and X. Yang, "Collective personalized change classification with multiobjective search," *IEEE Transactions on Reliability*, vol. 65, no. 4, pp. 1810–1829, 2016.

[43] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 17–26.

[44] J. Han, J. Pei, and M. Kamber, *Data mining: concepts and techniques*. Elsevier, 2011.

[45] X. Xia, L. Bao, D. Lo, and S. Li, "automated debugging considered harmful considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016, pp. 267–278.

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

[47] H. Abdi, "Bonferroni and šidák corrections for multiple comparisons," *Encyclopedia of measurement and statistics*, vol. 3, pp. 103–107, 2007.

[48] N. Cliff, *Ordinal methods for behavioral data analysis*. Lawrence Erlbaum Associates, 1996.

[49] J. L. Hintze and R. D. Nelson, "Violin plots: a box plot-density trace synergism," *The American Statistician*, vol. 52, no. 2, pp. 181–184, 1998.

[50] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," in *Proceedings of the 2017 25th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2017, p. to appear.

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

[52] T. Menzies and M. Shepperd, "Special issue on repeatable results in software engineering prediction," *Empirical Software Engineering*, vol. 17, no. 1-2, pp. 1–17, 2012.

[53] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, 2017.

[54] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 789–800.

[55] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 321–332.

[56] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, "The impact of mislabelling on the performance and interpretation of defect prediction models," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 812–823.