Noname manuscript No.
(will be inserted by the editor)

Revisiting Supervised and Unsupervised Models for Effort-Aware Just-in-Time Defect Prediction

Qiao Huang · Xin Xia · David Lo

Received: date / Accepted: date

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 (i.e., 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 about 12% - 27% more defective changes than a state-of-the-art supervised model (i.e., EALR) when using different evaluation settings. 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.

Qiao Huang
College of Computer Science and Technology, Zhejiang University, China
E-mail: tkdsheep@zju.edu.cn

Xin Xia
Faculty of Information Technology, Monash University, Australia
E-mail: xin.xia@monash.edu

David Lo
School of Information Systems, Singapore Management University, Singapore
E-mail: davidlo@smu.edu.sg

Aside from highlighting the above findings, we propose a simple but improved supervised model called CBS+, which leverages the idea of both EALR and LT. We investigate the performance of CBS+ using three different evaluation settings, including time-wise cross-validation, 10-times 10-fold cross-validation and cross-project validation. When compared with EALR, CBS+ detects about 15% - 26% more defective changes, while keeping the number of context switches and initial false alarms close to those of EALR. When compared with LT, the number of defective changes detected by CBS+ is comparable to LT's result, while CBS+ significantly reduces context switches and initial false alarms before first success. Finally, we discuss how to balance the tradeoff between the number of inspected defects and context switches, and present the implications of our findings for practitioners and researchers.

1 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 (Turhan et al., 2009; Xia et al., 2016b; Gyimothy et al., 2005; Hassan, 2009; Li et al., 2006; Munson and Khoshgoftaar, 1992). Mockus and Weiss (Mockus and Weiss, 2000) 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. (2013). 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* (Kamei et al., 2013) 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. (2013) 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. 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 (Zimmermann et al., 2009). Unfortunately, it is difficult to get sufficient training data for a new

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

project. To address this limitation, Yang et al. (2016) 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 (Koru et al., 2010). For example, considering 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. (2013), 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.
- 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. 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 does 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 or even triple as many as the number of changes inspected when using EALR in different evaluation settings. 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 (Meyer et al., 2014). Additionally, inspecting many changes can map to a high number of false alarms which in turn may lead to developer

[2] Some previous studies (Hall et al., 2012; Jiang et al., 2013; Rahman and Devanbu, 2013) also denoted this evaluation measure as *cost-effectiveness*.

fatigue and tool abandonment – c.f. (Parnin and Orso, 2011; Kochhar et al., 2016)

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 (Arisholm et al., 2007; Rahman et al., 2012; Jiang et al., 2013; Shihab et al., 2013; Valdivia Garcia and Shihab, 2014; Huang et al., 2017a). 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 (Meyer et al., 2014). We propose *IFA* because previous studies (Parnin and Orso, 2011; Kochhar et al., 2016) 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 and the same evaluation settings (including time-wise cross-validation, 10-times 10-fold cross-validation and cross-project validation) 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 descending order by the ratio between a change's defect proneness and its size. We inspect these sorted changes one by one until we reach the limit of inspection cost. If there is still budget left, we apply the same operation to those identified clean changes.

We investigate the performance of CBS+ and compare it with EALR and LT. When compared with EALR, CBS+ significantly improves the average *Recall* by 47% - 108% for three different evaluation settings, while keeping the results of *PCI@20%* and *IFA* close to those of EALR. When compared with LT, CBS+ achieves comparable results in terms of *Recall*, but it significantly reduces context switches and false alarms before first success.

This paper extends our preliminary study which appears as a research paper of ICSME 2017 (Huang et al., 2017b). In particular, we extend our preliminary work in the following directions:

1. We propose CBS+, which is an extended version of the supervised model (i.e., CBS) proposed in our preliminary work (Huang et al., 2017b). There are two major differences between CBS+ and CBS: (1) In CBS, we directly remove the changes that are predicted as non-defective, while in CBS+, we

continue to inspect these changes if there is still budget left after inspecting all the predicted defective changes. (2) In CBS, we sort the list of changes in ascending order by their size, while in CBS+, we sort the list of changes in descending order by the ratio between each change's defect proneness and its size (i.e., LA+LD). The motivation of these modifications will be further discussed in Section 4. Our experiments show that CBS+ performs much better in terms of *IFA* and achieves similar results in terms of other evaluation measures when compared with CBS.
2. We strengthen the experimental part by adding two more evaluation settings, including 10-times 10-fold cross-validation and cross-project validation.
3. We investigated the performance of CBS+ with different underlying classifiers and we found that Random Forest is also a good choice.
4. We conducted a survey with professional developers to further investigate the impact of inspecting too many changes in practice.
5. We evaluated different prediction models with the evaluation measure P_{opt} that was used in previous studies (Yang et al., 2016; Fu and Menzies, 2017).
6. We further discuss how to balance the tradeoff between *Recall* and *PCI@20%* by tuning a threshold λ in CBS+. Specifically, given a change, if its defect proneness predicted by CBS+ is no less than the threshold λ, then it would be identified as defective, and non-defective otherwise. In the discussion, we also compared our approach with a more recent work by Zhou et al., in which they applied an unsupervised model called ManualUp for cross-project defect prediction.

Our contributions, which form a super-set of those in our preliminary study, 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 2. We describe the technical details of the supervised and unsupervised models proposed by previous studies (i.e., EALR and LT) in Section 3. We introduce our improved supervised model in Section 4. We introduce the evaluation measures in Section 5. We present our experimental setup and results in Section 6 and 7, respectively. We discuss how to balance the tradeoff

between the number of inspected defects and context switches, and present the implications of our findings for practitioners and researchers in Section 8. We examine the threats to validity in Section 9. We conclude the paper and mention future work in Section 10.

2 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.

2.1 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 (Koru et al., 2009). 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 (Kim et al., 2008).

To address the above limitations, Mockus and Weiss (2000) are the first to study change-level defect prediction. 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. (2008) 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. Yin et al. (2011) empirically studied the incorrect bug-fixes from 4 large operating systems. 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. (2012) performed an industrial study on the risk of software changes. 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. (2013) 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.

2.2 Effort-Aware JIT Defect Prediction

Previous studies (Menzies and Di Stefano, 2004; Mende and Koschke, 2010; Arisholm et al., 2010) 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 (Arisholm et al., 2010; Menzies et al., 2010). Kamei et al. (2013) 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 (Hamill and Goseva-Popstojanova, 2009; Koru et al., 2009; Ostrand et al., 2004) have shown that about 80% of the defects are contained in about 20% of the files. Motivated by these work, Kamei et al. (2013) 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. 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.

More recently, Yang et al. (2016) 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 (Koru et al., 2010). They reported that using the same data provided by Kamei et al. (2013), their unsupervised model could achieve higher *Recall* when compared with supervised model. Following Yang et al.'s work, Yan et al. (2017) 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.

3 Effort-Aware JIT Defect Prediction Models

In this section, we introduce the technical details of the supervised model proposed by Kamei et al. (2013), and unsupervised model proposed by Yang et al. (2016) for effort-aware JIT defect prediction.

Table 1 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
REXP	Recent developer experience
SEXP	Developer experience on a subsystem

3.1 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 1 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 (D'Ambros et al., 2010; Hassan, 2009; Mockus and Weiss, 2000; Nagappan et al., 2006b). 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 (Moser et al., 2008; Nagappan and Ball, 2005). The purpose dimension only consists of FIX, and it is believed that a defect-fixing change is more likely to introduce a new defect (Graves et al., 2000; Guo et al., 2010; Purushothaman and Perry, 2005). The metrics in history dimension can tell us how developers interacted with different files in the past. As stated by Yang et al. (Yang et al., 2016), a change is more likely to be defective if the touched files have been modified by more developers (Matsumoto et al., 2010), by more recent changes (Graves et al., 2000), or by more unique last changes (D'Ambros et al., 2010; Hassan, 2009). 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 (Mockus and Weiss, 2000).

Based on these 14 metrics, Kamei et al. (2013) built a linear regression model 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 (Kamei et al., 2013). 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 1) 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 (Kamei et al., 2013).

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. (2013) reported that the EALR model could detect 35% of all defective changes when developers inspect 20% of LOC modified by all changes.

3.2 Unsupervised Model by Yang et al. (LT)

More recently, Yang et al. (2016) leveraged the same metrics in Kamei et al.'s work (2013) 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, 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 (Koru et al., 2010). Thus, we also choose LT as the underlying metric for unsupervised model in our experiment.

4 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 (2013), 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. On the other hand, for those changes that are classified to be potentially clean, CBS+ just puts them aside unless there is still budget left after inspecting all the predicted defective changes.

Algorithm 1 Pseudocode for Classifier Building

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

Algorithm 1 presents the pseudo-code to build a classifier as proposed by Kamei et al. (2013). We first follow Kamei et al. to use under-sampling strategy to deal with data imbalance (i.e., they randomly removed instances of the majority class until the training data is balanced) (Line 8). In practice, we can also use other sampling strategies, such as over-sampling or a combination of over-sampling and under-sampling. However, our preliminary experiment results show that under-sampling always achieves the better performance in predicting defective changes when compared with other sampling strategies. Besides, using under-sampling, the amount of training data is also substantially reduced, thus reducing time and memory cost during training. 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 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 9-17). In practice, given a new change, the logistic classifier would output a value between 0 and 1 to represent the defect

Algorithm 2 Pseudocode for CBS+

```
1: CBS+(Logistic, TestSet, λ)
2: Input:
3: Logistic: The classifier built on training dataset
4: TestSet: Testing set of changes
5: λ: The threshold of defect proneness to decide whether a change is defective or not
6: Output:
7: RankedList: Ranked list of changes for inspection
8: Method:
9: RankedList, Defective, NonDefective = ∅;
10: for all change c ∈ TestSet do
11:     Use Logistic to predict the defect proneness p of change c;
12:     if p ≥ λ then
13:         Add c into Defective;
14:     else
15:         Add c into NonDefective;
16:     end if
17: end for
18: Sort changes in Defective in descending order by the ratio between each change's defect
    proneness p and its size (i.e., LA+LD).
19: Sort changes in NonDefective in descending order by the ratio between each change's
    defect proneness p and its size (i.e., LA+LD).
20: Append Defective to RankedList and then append NonDefective to RankedList.
21: return RankedList;
```

proneness of this change. We use the input parameter λ as a threshold of defect proneness to decide whether a change is defective or not. By default, we set λ as 0.5, which means that a change would be classified as potentially defective if its predicted defect proneness is no less than 0.5; otherwise it will be classified as potentially non-defective. Then we separately sort the list of predicted defective changes and non-defective changes by the ratio between each change's defect proneness p and its size (i.e., LA+LD) (Lines 18-19). Finally, we append the sorted non-defective list to the end of the defective list, and return the defective list for inspection (Lines 20-21).

Note that CBS+ is an extended version of the supervised model (i.e., CBS) proposed in our preliminary work (Huang et al., 2017b). The major differences between CBS+ and CBS are as follows:

1. In CBS, we directly remove the changes that are predicted as non-defective. However, in some cases (e.g., when a higher value is set for the threshold λ), only a few changes would be predicted as defective and we would have a lot of budget left after inspecting these changes. Since our classifier is imperfect, for those changes that are predicted as non-defective, some of them may be false negatives (i.e., they are actually defective changes). Thus, in CBS+, we continue to inspect these changes if there is still budget left after inspecting all the predicted defective changes. Note that when evaluation measures are calculated (see Section 5), the ground truth labels of these changes are used to identify true/false positives and negatives.
2. In CBS, we sort the list of changes in ascending order by their size (i.e., LA+LD) so that smaller changes would be inspected first. Since smaller

changes are less likely to be defective (Moser et al., 2008; Nagappan and Ball, 2005), it would be highly possible that many highly ranked changes are false alarms. These initial false alarms may negatively impact practitioners' patience and confidence. To solve this problem, in CBS+, we sort the list of changes in descending order by the ratio between each change's defect proneness p and its size (i.e., LA+LD). In this way, most small changes are still highly ranked, while among these small changes, the change that has high defect proneness would be inspected first.

5 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 (Kamei et al., 2013; Yang et al., 2016, 2017; Xia et al., 2016c; Yang et al., 2015). They focused on achieving high *Recall* so that more defective changes could be detected. *Recall* is computed as:

$$Recall = n/N \tag{2}$$

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:

$$Precision = n/m \tag{3}$$

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 (Han et al., 2011). Therefore, to fairly evaluate the prediction model, *F1-score* is also widely used in prior software engineering studies (Arisholm et al., 2007; Rahman et al., 2012; Jiang et al., 2013; Shihab et al., 2013; Valdivia Garcia and Shihab, 2014; Huang et al., 2017a). 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:

$$F1score = \frac{2 * Precision * Recall}{Precision + Recall} \tag{4}$$

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 (Kamei et al., 2013; Yang et al., 2016) 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. (2014) 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:

$$PCI@20\% = m/M \tag{5}$$

IFA: Number of Initial False Alarms encountered before we find the first defect. Inspired by previous studies on fault localization (Parnin and Orso, 2011; Kochhar et al., 2016; Xia et al., 2016a), 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 (2011) 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:

$$IFA = k \tag{6}$$

6 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 2 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	4,4,55	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

6.1 Data Statistics

Table 2 summarizes the statistics of the studied projects. This dataset is published by Kamei et al. (2013), and also used in Yang et al.'s work (2016). 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%).

6.2 Experiment Setting

In this paper, we follow Yang et al. (2016) to evaluate each prediction model using the following three evaluation settings:

Time-wise cross-validation. In this setting, a prediction model is evaluated within the same project while the chronological order of changes is considered additionally. For each project, 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. Finally, we would get $N-5$ prediction effectiveness results for each prediction model and we also record the median result. As stated by Yang et al. (2016), 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.

10 times 10-fold cross-validation. In this setting, a prediction model is also evaluated within the same project. For each project, we first randomly shuffle the dataset and divide it into ten folds of approximately equal size. Then, each fold is used as a testing dataset to evaluate the prediction model build on the other nine folds (i.e., training dataset). The entire process is repeated ten times to alleviate possible sampling bias during random shuffle and splitting. Thus, we would get $10 \times 10 = 100$ prediction effectiveness results for each prediction model and we follow Yang et al. (2016) to record the median result. Note that 10-fold cross-validation cannot guarantee the changes

used 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. However, 10-fold cross-validation is a standard evaluation setting when there is only one independent dataset, and it was used by previous studies for defect prediction. For example, Moser et al. (2008) used this setting to evaluate the efficiency of change metrics and static code attributes for defect prediction. Considering that Yang et al. (2016) also used 10-fold cross-validation, we strictly follow them to keep this setting in our work for a more comprehensive comparison. Overall, the main goal of this paper is to replicate Yang et al.'s work and ensure that our new finding is not due to a different validation setting.

Cross-project validation. In this setting, a prediction model is evaluated across different projects. Given n projects, each time we choose 1 project as a target project for testing, and the other $n-1$ projects are used as source projects to train the prediction model. Since it is difficult to decide which source project is the best choice for model training in real application, we simply merge the changes in all of the $n-1$ source projects to train the prediction model.

Finally, since there are multiple runs for each prediction model in every evaluation setting, we apply the Wilcoxon signed-rank test (Wilcoxon, 1945) with Bonferroni correction (Abdi, 2007) 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) (Cliff, 1996) to quantify the amount of difference between two approaches. The amount of difference is considered negligible ($\mid \delta \mid < 0.147$), small ($0.147 \leq \mid \delta \mid < 0.33$), moderate ($0.33 \leq \mid \delta \mid < 0.474$), or large ($\mid \delta \mid \geq 0.474$). Note that in time-wise cross-validation and 10 times 10-fold cross-validation, we calculate p-value and Cliff's delta for each project, since there are multiple runs for each project in these two settings. In cross-project validation, we group the results of all projects to calculate p-value and Cliff's delta, since the prediction model only runs once for each project in this setting.

6.3 Research Questions

We investigate the following three research questions:

RQ1: Why does 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 (2016) could outperform supervised models in terms of *Recall*.

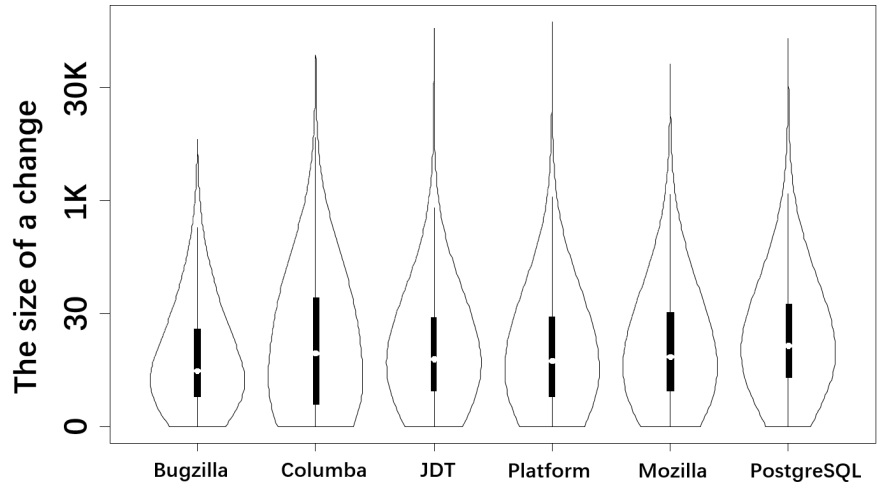

Fig. 1 Distribution of change size in each project.

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

Yang et al. (2016) 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 5. 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.

7 Experiment Results

7.1 RQ1: Why does 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 change size in each project. Our prelimi-

Table 3 The median number of changes to be inspected given the same LOC budget (20%) when using the supervised model (EALR) and unsupervised model (LT)

Project	Time-wise cross-validation		10-times 10-fold cross-validation		Cross-project validation	
	EALR	LT	EALR	LT	EALR	LT
BUG	24	36	195	255	1,712	2,359
COL	66	125	178	343	1,687	3,605
JDT	290	568	708	2,434	2,370	26,062
PLA	411	963	1,318	3,906	17,562	40,748
MOZ	460	1,245	1,563	5,672	19,584	56,578
POS	89	157	603	1,438	3,147	15,105
Average	223	516	761	2,341	7,677	24,076
Comparison		2.3x		3.1x		3.1x

nary 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 (Hintze and Nelson, 1998) to visualize it. Figure 1 presents the visualization results. Since the 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 figure's y-axis, which represents the size of a change. 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 one thousand LOC. On the other hand, a small number of changes modify a huge number of LOC (e.g., 30K LOC in the figure), 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 3 shows the number of changes inspected when using EALR and LT for three different evaluation settings. Note that in *time-wise cross-validation* and *10-times 10-fold cross-validation*, since there are multiple runs for each

Table 4 Comparison results of EALR and LT for time-wise cross-validation

Time-wise cross-validation										
Project	Recall		Precision		F1-score		PCI@20%		IFA	
	EALR	LT	EALR	LT	EALR	LT	EALR	LT	EALR	LT
BUG	0.299	**0.449(M)**	0.364	0.333	0.325	0.378	**0.312(L)**	0.516	1	2
COL	0.400	0.440	**0.250(M)**	0.190	0.299	0.265	**0.440(L)**	0.677	**0(L)**	16
JDT	0.347	**0.452(L)**	**0.155(M)**	0.112	0.210	0.181	**0.345(L)**	0.611	**1(L)**	32
PLA	0.290	**0.432(L)**	**0.157(M)**	0.110	0.198	0.178	**0.295(L)**	0.590	**0(L)**	123
MOZ	0.190	**0.363(L)**	**0.045(M)**	0.035	0.072	0.062	**0.232(L)**	0.554	**4(L)**	137
POS	0.331	**0.432(L)**	**0.235(M)**	0.176	0.255	0.246	**0.373(L)**	0.647	**1(L)**	6
AVG	0.310	0.428	0.201	0.159	0.227	0.218	0.333	0.599	1	53
Wins	0	5	5	0	0	0	6	0	5	0

Table 5 Comparison results of EALR and LT for 10 times 10-fold cross-validation

10-times 10-fold cross-validation										
Project	Recall		Precision		F1-score		PCI@20%		IFA	
	EALR	LT	EALR	LT	EALR	LT	EALR	LT	EALR	LT
BUG	0.441	0.476	**0.386(L)**	0.314	0.405	0.384	**0.422(L)**	0.552	**0(L)**	8
COL	0.404	**0.591(L)**	**0.317(L)**	0.236	0.357	0.336	**0.398(L)**	0.771	**0(L)**	34
JDT	0.208	**0.530(L)**	**0.158(L)**	0.115	0.177	**0.189(M)**	**0.200(L)**	0.688	**0(L)**	92
PLA	0.280	**0.467(L)**	**0.193(L)**	0.113	**0.225(L)**	0.182	**0.205(L)**	0.608	**0(L)**	524
MOZ	0.148	**0.370(L)**	**0.050(L)**	0.034	**0.074(L)**	0.062	**0.159(L)**	0.577	**1(L)**	526
POS	0.284	**0.502(L)**	**0.248(L)**	0.180	0.262	0.266	**0.295(L)**	0.704	**1(L)**	21
AVG	0.294	0.489	0.225	0.165	0.250	0.237	0.280	0.650	0	201
Wins	0	5	6	0	2	1	6	0	6	0

project, we report the median number of inspected changes. The row *Average* represents the average number of inspected changes across the six projects. The row *Comparison* represents how many times the average result of LT is larger than EALR. The results show that, LT requires developers to inspect more than twice or even triple as many changes as those required by using EALR. Thus, 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. For example, according to the evaluation results under *time-wise cross-validation* in Yang et al.'s work, on average, EALR can find 31% of all defective changes, while LT can find 43% of all defective changes, which only improves the *Recall* by 39%. 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 or even triple as many changes as those required by using EALR. However, problems may arise due to frequent context switches and large number of false alarms.

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

Tables 4 - 6 present the evaluation results of Kamei et al.'s supervised model (EALR) and Yang et al.'s unsupervised model (LT) under the three differ-

Table 6 Comparison results of EALR and LT for cross-project validation

Cross-project validation										
Project	Recall		Precision		F1-score		PCI@20%		IFA	
	EALR	LT	EALR	LT	EALR	LT	EALR	LT	EALR	LT
BUG	0.380	0.435	0.377	0.312	0.379	0.364	0.371	0.511	0	67
COL	0.371	0.641	0.299	0.242	0.331	0.351	0.379	0.809	0	220
JDT	0.111	0.582	0.238	0.114	0.151	0.190	0.067	0.737	0	695
PLA	0.242	0.494	0.130	0.115	0.169	0.186	0.273	0.634	0	5004
MOZ	0.158	0.367	0.041	0.033	0.066	0.061	0.199	0.576	2	4911
POS	0.196	0.533	0.319	0.180	0.243	0.270	0.154	0.739	3	167
AVG	0.243	0.509	0.234	0.166	0.223	0.237	0.241	0.668	1	1844
Winner	**LT (Large)**		**EALR (Medium)**		-		**EALR (Large)**		**EALR (Large)**	

ent evaluation settings, respectively. For *10 times 10-fold cross-validation* and *time-wise cross-validation*, given a target project and evaluation measure, we regard a prediction model to win against a competing model, if the earlier significantly outperforms the latter model with moderate or large improvement in terms of Cliff's delta. In the tables, we highlight these winning cases in bold, with (M) and (L) indicating that the improvement is moderate or large. The row *AVG* reports the average result over the six projects. The row *Wins* reports the number of projects for which the corresponding prediction model wins against the other model. For *cross-project validation*, since the p-value and Cliff's delta are calculated using the grouped results of all projects, we directly report the *Winner* in the last row, along with the magnitude of its improvement measured using Cliff's delta (i.e., moderate or large) over the other model. If there is no winner, we mark it by a hyphen.

For *time-wise cross-validation*, LT wins in terms of *Recall* in five out of the six 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 five out of the six 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 five out of the six 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-50 changes recommended by unsupervised model are all false alarms. According to a survey by Kochhar et al. (2016) that investigates practitioners' expectations on automated fault localization, most practitioners find it unacceptable if the first **10** suspicious program elements returned by a tool are all false alarms. Thus, this raises a question on the practicality of the unsupervised model. As a comparison, the *IFA* of EALR is much more reasonable. For every project, EALR can find the first defective change when inspecting the top-10 suspicious changes. Note that for some projects, the *IFA* results achieved by EALR can be zeros, which means that the first change recommended by EALR is truly defective and the number of initial false alarms encountered is zero.

Table 7 Comparison results of EALR and CBS+ for time-wise cross-validation

Time-wise cross-validation										
Project	Recall		Precision		F1-score		PCI@20%		IFA	
	EALR	CBS+	EALR	CBS+	EALR	CBS+	EALR	CBS+	EALR	CBS+
BUG	0.299	**0.452(M)**	0.364	**0.452(M)**	0.325	**0.448(L)**	0.312	0.375	1	1
COL	0.400	**0.488(M)**	0.250	**0.339(M)**	0.299	**0.390(M)**	0.440	0.366	0	0
JDT	0.347	**0.452(L)**	0.155	**0.214(L)**	0.210	**0.299(L)**	0.345	0.301	1	1
PLA	0.290	**0.524(L)**	0.157	0.207	0.198	**0.304(L)**	0.295	0.368	0	0
MOZ	0.190	**0.440(L)**	0.045	**0.101(L)**	0.072	**0.159(L)**	0.232	0.252	4	4
POS	0.331	**0.441(L)**	0.235	**0.372(L)**	0.255	**0.389(L)**	0.373	0.323	1	1
AVG	0.310	0.466	0.201	0.281	0.227	0.332	0.333	0.331	1	1
Wins	0	6	0	5	0	6	0	0	0	0

For *10-times 10-fold cross-validation* and *cross-project validation*, the results are similar to those for *time-wise cross-validation*. Generally, in these two settings, LT significantly outperforms EALR by a substantial margin in terms of *Recall*, while performing relatively bad in terms of *Precision*. LT performs much worse in terms of *PCI@20%* and *IFA*. On average, EALR requires developers to inspect less than 30% of all changes, while LT requires developers to inspect about 65% of all changes, which is more than twice as many changes as those required by EALR. When using LT in these two evaluation settings, developers would encounter hundreds or even thousands of false alarms before the first defective change. On the contrary, when using EALR, developers would only encounter no more than one initial false alarms in most cases.

Note that the *IFA* results achieved by LT and EALR under time-wise cross-validation are different from those reported in our previous work (Huang et al., 2017b). The reason is that the source code in our previous work mistakenly calculated *IFA* as the number of initial false alarms encountered before we find the *second* defective change. In this journal paper, we fixed this bug and updated the results of *IFA*. In general, the new *IFA* results of all models are lower than those reported in our ICSME paper. However, this error does not affect our conclusion, i.e., the *IFA* achieved by LT is much larger than EALR.

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

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

We first compare the evaluation results of CBS+ with EALR. Tables 7 - 9 present the comparison results of EALR and CBS+ for three different evaluation settings.

– For *time-wise cross-validation*, CBS+ can find about 15% more defective changes on average across the six projects, which significantly outperforms EALR in terms of *Recall* with an average improvement of 47%. The average

Table 8 Comparison results of EALR and CBS+ for 10-times 10-fold cross-validation

10-times 10-fold cross-validation										
Project	Recall		Precision		F1-score		PCI@20%		IFA	
	EALR	CBS+	EALR	CBS+	EALR	CBS+	EALR	CBS+	EALR	CBS+
BUG	0.441	**0.572(L)**	0.386	**0.503(L)**	0.405	**0.533(L)**	0.422	0.419	0	0
COL	0.404	**0.519(L)**	0.317	**0.461(L)**	0.357	**0.487(L)**	0.398	0.354	0	0
JDT	0.208	**0.530(L)**	0.158	**0.238(L)**	0.177	**0.329(L)**	**0.200(L)**	0.323	0	0
PLA	0.280	**0.609(L)**	0.193	**0.247(L)**	0.225	**0.351(L)**	**0.205(L)**	0.361	0	0
MOZ	0.148	**0.458(L)**	0.050	**0.112(L)**	0.074	**0.180(L)**	**0.159(L)**	0.212	1	2
POS	0.284	**0.497(L)**	0.248	**0.444(L)**	0.262	**0.470(L)**	0.295	0.278	1	0
AVG	0.294	0.531	0.225	0.334	0.250	0.392	0.280	0.325	0	0
Wins	0	6	0	6	0	6	3	0	0	0

Table 9 Comparison results of EALR and CBS+ for cross-project validation

Cross-project validation										
Project	Recall		Precision		F1-score		PCI@20%		IFA	
	EALR	CBS+	EALR	CBS+	EALR	CBS+	EALR	CBS+	EALR	CBS+
BUG	0.380	0.285	0.377	0.621	0.379	0.391	0.371	0.169	0	0
COL	0.371	0.566	0.299	0.402	0.331	0.470	0.379	0.431	0	0
JDT	0.111	0.549	0.238	0.214	0.151	0.308	0.067	0.369	0	0
PLA	0.242	0.551	0.130	0.238	0.169	0.333	0.273	0.340	0	0
MOZ	0.158	0.569	0.041	0.058	0.066	0.104	0.199	0.518	2	6
POS	0.196	0.508	0.319	0.364	0.243	0.424	0.154	0.349	3	2
AVG	0.243	0.505	0.234	0.316	0.223	0.338	0.241	0.363	1	1
Winner	**CBS+ (Large)**		-		**CBS+ (Large)**		-		-	

Table 10 Comparison results of LT and CBS+ for time-wise cross-validation

Time-wise cross-validation										
Project	Recall		Precision		F1-score		PCI@20%		IFA	
	LT	CBS+	LT	CBS+	LT	CBS+	LT	CBS+	LT	CBS+
BUG	0.449	0.452	0.333	**0.452(M)**	0.378	0.448	0.516	**0.375(M)**	2	1
COL	0.440	0.488	0.190	**0.339(L)**	0.265	**0.390(L)**	0.677	**0.366(L)**	16	**0(L)**
JDT	0.452	0.452	0.112	**0.214(L)**	0.181	**0.299(L)**	0.611	**0.301(L)**	32	**1(L)**
PLA	0.432	**0.524(L)**	0.110	**0.207(L)**	0.178	**0.304(L)**	0.590	**0.368(L)**	123	**0(L)**
MOZ	0.363	**0.440(M)**	0.035	**0.101(L)**	0.062	**0.159(L)**	0.554	**0.252(L)**	137	**4(L)**
POS	0.432	0.441	0.176	**0.372(L)**	0.246	**0.389(L)**	0.647	**0.323(L)**	6	**1(L)**
AVG	0.428	0.466	0.159	0.281	0.218	0.332	0.599	0.331	53	1
Wins	0	2	0	6	0	5	0	6	0	5

Table 11 Comparison results of LT and CBS+ for 10-times 10-fold cross-validation

10-times 10-fold cross-validation										
Project	Recall		Precision		F1-score		PCI@20%		IFA	
	LT	CBS+	LT	CBS+	LT	CBS+	LT	CBS+	LT	CBS+
BUG	0.476	**0.572(L)**	0.314	**0.503(L)**	0.384	**0.533(L)**	0.552	**0.419(L)**	8	**0(L)**
COL	**0.591(M)**	0.519	0.236	**0.461(L)**	0.336	**0.487(L)**	0.771	**0.354(L)**	34	**0(L)**
JDT	0.530	0.530	0.115	**0.238(L)**	0.189	**0.329(L)**	0.688	**0.323(L)**	92	**0(L)**
PLA	0.467	**0.609(L)**	0.113	**0.247(L)**	0.182	**0.351(L)**	0.608	**0.361(L)**	524	**0(L)**
MOZ	0.370	**0.458(L)**	0.034	**0.112(L)**	0.062	**0.180(L)**	0.577	**0.212(L)**	526	**2(L)**
POS	0.502	0.497	0.180	**0.444(L)**	0.266	**0.470(L)**	0.704	**0.278(L)**	21	**0(L)**
AVG	0.489	0.531	0.165	0.334	0.237	0.392	0.650	0.325	201	0
Wins	1	3	0	6	0	6	0	6	0	6

PCI@20% of CBS+ is quite close to that of EALR for each project, which suggests that CBS+ does not require inspection of additional changes. Also, CBS+ achieves the same results as EALR in terms of *IFA*, and the result of *IFA* for each project is less than 10.

– For *10-times 10-fold cross-validation*, CBS+ can find about 24% more defective changes and wins in terms of *Recall*, *Precision* and *F1-score* for every project. We also notice that EALR wins in terms of *PCI@20%* in

Table 12 Comparison results of LT and CBS+ for cross-project validation

Cross-project validation										
Project	Recall		Precision		F1-score		PCI@20%		IFA	
	LT	CBS+	LT	CBS+	LT	CBS+	LT	CBS+	LT	CBS+
BUG	0.435	0.285	0.312	0.621	0.364	0.391	0.511	0.169	67	0
COL	0.641	0.566	0.242	0.402	0.351	0.470	0.809	0.431	220	0
JDT	0.582	0.549	0.114	0.214	0.190	0.308	0.737	0.369	695	0
PLA	0.494	0.551	0.115	0.238	0.186	0.333	0.634	0.340	5004	0
MOZ	0.367	0.569	0.033	0.058	0.061	0.104	0.576	0.518	4911	6
POS	0.533	0.508	0.180	0.364	0.270	0.424	0.739	0.349	167	2
AVG	0.509	0.505	0.166	0.316	0.237	0.338	0.668	0.363	1844	1
Winner	-		-		-		**CBS+ (Large)**		**CBS+ (Large)**	

three out of the six projects, which indicates that CBS+ also sacrifices *PCI@20%* to achieve higher *Recall*. Considering that CBS+ improves *Recall* by 118% - 209% in these three projects, we believe the sacrifice on *PCI@20%* is acceptable since CBS+ only requires developers to inspect about 1.5 - 1.75 times as many changes as those required by EALR.

– For *cross-project validation*, CBS+ can find about 26% more defective changes and wins in terms of *Recall* and *F1-score*. For the other three measures, CBS+ achieves comparable results when compared with EALR. In summary, CBS+ almost dominates EALR, since it can find more defective changes, while achieving comparable results in terms of *PCI@20%* and *IFA*.

As a summary, when compared with EALR, CBS+ detects about 15% - 26% more defective changes, while keeping the number of context switches and initial false alarms close to those of EALR.

Then we compare the evaluation results of CBS+ with those of LT. Tables 10 - 12 present the comparison results of LT and CBS+ for three different evaluation settings.

– For *time-wise cross-validation*, CBS+ wins in terms of *Recall* in two out the six projects and finds about 4% more defective changes on average across the six projects, which slightly improves the average *Recall* by 9%. As for each of the other four evaluation measures, CBS+ wins in at least five out of the six projects.
– For *10-times 10-fold cross-validation*, LT wins in terms of *Recall* in one out of the six projects while CBS+ wins in terms of *Recall* in three out the six projects. On average, CBS+ can find 4% more defective changes. Also, CBS+ wins in terms of all the other four evaluation measures for every project.
– For *cross-project validation*, the *Recall* achieved by LT and CBS+ may vary a lot for different projects. For example, LT can find 15% more defective changes in Bugzilla project, while CBS+ can find 20% more defective changes in Mozilla project. On average, the *Recall* achieved by CBS+ is quite close to that by LT. On the other hand, CBS+ significantly outperforms LT by a substantial margin in terms of *PCI@20%* and *IFA*, which

Table 13 Comparison results of OneWay and CBS+ for time-wise cross-validation

Time-wise cross-validation										
Project	Recall		Precision		F1-score		PCI@20%		IFA	
	OW	CBS+	OW	CBS+	OW	CBS+	OW	CBS+	OW	CBS+
BUG	0.362	0.452	0.394	**0.452(M)**	0.369	0.448	0.396	0.375	4	**1(M)**
COL	0.561	0.488	0.227	**0.339(L)**	0.315	**0.390(M)**	0.649	**0.366(L)**	24	**0(L)**
JDT	0.422	0.452	0.117	**0.214(L)**	0.183	**0.299(L)**	0.553	**0.301(L)**	32	**1(L)**
PLA	0.407	**0.524(L)**	0.110	**0.207(L)**	0.167	**0.304(L)**	0.537	**0.368(L)**	201	**0(L)**
MOZ	0.327	**0.440(L)**	0.041	**0.101(L)**	0.074	**0.159(L)**	0.449	**0.252(L)**	113	**4(L)**
POS	0.451	0.441	0.224	**0.372(L)**	0.294	**0.389(L)**	0.555	**0.323(L)**	14	**1(L)**
AVG	0.422	0.466	0.186	0.281	0.234	0.332	0.523	0.331	65	1
Wins	0	2	0	6	0	5	0	5	0	6

Table 14 Comparison results of OneWay and CBS+ for 10-times 10-fold cross-validation

10-times 10-fold cross-validation										
Project	Recall		Precision		F1-score		PCI@20%		IFA	
	OW	CBS+	OW	CBS+	OW	CBS+	OW	CBS+	OW	CBS+
BUG	0.494	**0.572(L)**	0.370	**0.503(L)**	0.424	**0.533(L)**	0.479	**0.419(L)**	36	**0(L)**
COL	**0.659(L)**	0.519	0.275	**0.461(L)**	0.385	**0.487(L)**	0.738	**0.354(L)**	96	**0(L)**
JDT	0.531	0.530	0.116	**0.238(L)**	0.190	**0.329(L)**	0.679	**0.323(L)**	94	**0(L)**
PLA	0.467	**0.609(L)**	0.114	**0.247(L)**	0.183	**0.351(L)**	0.605	**0.361(L)**	527	**0(L)**
MOZ	0.371	**0.458(L)**	0.034	**0.112(L)**	0.063	**0.180(L)**	0.577	**0.212(L)**	521	**2(L)**
POS	0.508	0.497	0.191	**0.444(L)**	0.285	**0.470(L)**	0.685	**0.278(L)**	25	**0(L)**
AVG	0.505	0.531	0.183	0.334	0.255	0.392	0.627	0.325	217	0
Wins	1	3	0	6	0	6	0	6	0	6

suggests that CBS+ significantly reduces the amount of false alarms and the amount of changes required to be inspected for each project.

As a summary, when compared with LT, the number of defective changes detected by CBS+ is comparable to LT's result, while CBS+ significantly reduces context switches and number of initial false alarms before first success.

When compared with EALR, CBS+ significantly improves the average Recall by 47% - 108% for three different evaluation settings, while keeping the results of PCI@20% and IFA close to those of EALR. When compared with LT, CBS+ achieves comparable results in terms of Recall, but it significantly reduces context switches and false alarms before first success.

8 Discussion

8.1 Comparison with Fu and Menzies's Work

Most recently, Fu and Menzies (2017) also revisited unsupervised model in effort-aware JIT defect prediction. They replicated Yang et al.'s work (2016) 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:

Table 15 Comparison results of OneWay and CBS+ for cross-project validation

Cross-project validation										
Project	Recall		Precision		F1-score		PCI@20%		IFA	
	OW	CBS+	OW	CBS+	OW	CBS+	OW	CBS+	OW	CBS+
BUG	0.432	0.285	0.392	0.621	0.411	0.391	0.405	0.169	445	0
COL	0.702	0.566	0.273	0.402	0.393	0.470	0.786	0.431	962	0
JDT	0.582	0.549	0.114	0.214	0.190	0.308	0.737	0.369	695	0
PLA	0.494	0.551	0.115	0.238	0.186	0.333	0.634	0.340	5004	0
MOZ	0.367	0.569	0.033	0.058	0.061	0.104	0.576	0.518	4911	6
POS	0.533	0.508	0.180	0.364	0.270	0.424	0.739	0.349	167	2
AVG	0.518	0.505	0.185	0.316	0.252	0.338	0.646	0.363	2031	1
Winner	-		**CBS+ (Medium)**		**CBS+ (Medium)**		**CBS+ (Large)**		**CBS+ (Large)**	

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

We also compare our approach CBS+ with OneWay. Tables 13 - 15 present the comparison results of CBS+ and OneWay (denoted as "OW" in the table).

- For *time-wise cross-validation*, when compared with OneWay, CBS+ wins in terms of *Recall* in two out of the six projects and achieves comparable results in the other four projects. CBS+ wins in terms of *Precision*, *F1-score* and *IFA* in at least five out of the six projects.
- For *10-times 10-fold cross-validation*, OneWay wins in terms of *Recall* in one out of the six projects, while CBS+ wins in terms of *Recall* in three out of the six projects. On average, CBS+ slightly performs better in terms of *Recall*. Also, CBS+ wins in terms of the other evaluation measures in all of the six projects.
- For *cross-project validation*, when compared with OneWay, CBS+ achieves comparable results in terms of *Recall* and significantly outperforms OneWay by a substantial margin in terms of the other evaluation measures.

8.2 Comparison between CBS+ and CBS

In this paper, we propose CBS+, which is an extended version of the supervised model (i.e., CBS) proposed in our preliminary work (Huang et al., 2017b). Although we have discussed the major differences between CBS+ and CBS in section 4, we are also interested to compare the performance of CBS+ and CBS.

Tables 16 - 18 present the comparison results of CBS and CBS+ for three different evaluation settings. The results show that CBS performs worse in

Table 16 Comparison results of CBS and CBS+ for time-wise cross-validation

Time-wise cross-validation										
Project	Recall		Precision		F1-score		PCI@20%		IFA	
	CBS	CBS+	CBS	CBS+	CBS	CBS+	CBS	CBS+	CBS	CBS+
BUG	0.438	0.452	0.473	0.452	0.442	0.448	0.368	0.375	1	1
COL	0.464	0.488	0.352	0.339	0.390	0.390	0.364	0.366	8	**0(L)**
JDT	0.453	0.452	0.216	0.214	0.297	0.299	0.302	0.301	11	**1(L)**
PLA	0.515	0.524	0.207	0.207	0.304	0.304	0.369	0.368	20	**0(L)**
MOZ	0.435	0.440	0.098	0.101	0.156	0.159	0.252	0.252	28	**4(L)**
POS	0.444	0.441	0.376	0.372	0.387	0.389	0.321	0.323	4	**1(M)**
AVG	0.458	0.466	0.287	0.281	0.329	0.332	0.329	0.331	12	1
Wins	0	0	0	0	0	0	0	0	0	5

Table 17 Comparison results of CBS and CBS+ for 10-times 10-fold cross-validation

10-times 10-fold cross-validation										
Project	Recall		Precision		F1-score		PCI@20%		IFA	
	CBS	CBS+	CBS	CBS+	CBS	CBS+	CBS	CBS+	CBS	CBS+
BUG	0.567	0.572	0.500	0.503	0.531	0.533	0.419	0.419	3	**0(M)**
COL	0.517	0.519	0.459	0.461	0.491	0.487	0.354	0.354	13	**0(L)**
JDT	0.531	0.530	0.237	0.238	0.329	0.329	0.323	0.323	42	**0(L)**
PLA	0.604	0.609	0.245	0.247	0.350	0.351	0.362	0.361	87	**0(L)**
MOZ	0.450	0.458	0.110	0.112	0.177	0.180	0.213	0.212	113	**2(L)**
POS	0.497	0.497	0.443	0.444	0.470	0.470	0.279	0.278	32	**0(L)**
AVG	0.528	0.531	0.332	0.334	0.391	0.392	0.325	0.325	48	0
Wins	0	0	0	0	0	0	0	0	0	6

Table 18 Comparison results of CBS and CBS+ for cross-project validation

Cross-project validation										
Project	Recall		Precision		F1-score		PCI@20%		IFA	
	CBS	CBS+	CBS	CBS+	CBS	CBS+	CBS	CBS+	CBS	CBS+
BUG	0.284	0.285	0.615	0.621	0.389	0.391	0.170	0.169	7	0
COL	0.570	0.566	0.404	0.402	0.473	0.470	0.431	0.431	164	0
JDT	0.549	0.549	0.213	0.214	0.307	0.308	0.370	0.369	570	0
PLA	0.550	0.551	0.238	0.238	0.332	0.333	0.340	0.340	1370	0
MOZ	0.561	0.569	0.057	0.058	0.103	0.104	0.519	0.518	1815	6
POS	0.508	0.508	0.364	0.364	0.424	0.424	0.350	0.349	174	2
AVG	0.504	0.505	0.315	0.316	0.338	0.338	0.363	0.363	683	1
Winner	-		-		-		-		**CBS+(Large)**	

terms of *IFA* (e.g., the average *IFA* achieved by CBS in cross-project setting is more than 600), since it simply ranks small changes first. In general, when compared with CBS, CBS+ achieves almost the same results in terms of *Recall*, *Precision*, *F1-score* and *PCI@20%*, while significantly and substantially reduces *IFA* in every evaluation setting. The average *IFA* achieved by CBS+ is no more than 1 in every evaluation setting.

8.3 Evaluating Results with P_{opt}

In Yang et al. (2016) and Fu and Menzies (2017)'s work, they also evaluated their models using the evaluation measure P_{opt}, which is based on the concept of the Alberg diagram (Arisholm et al., 2010). An Alberg diagram (see Figure 2 for an example) shows the relationship between the *Recall* achieved by

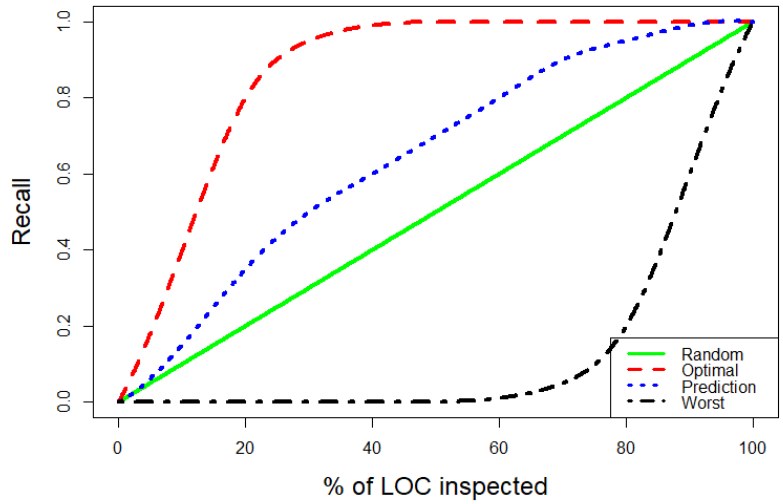

Fig. 2 An example of the relationship between *Recall* and the inspection cost for different prediction models

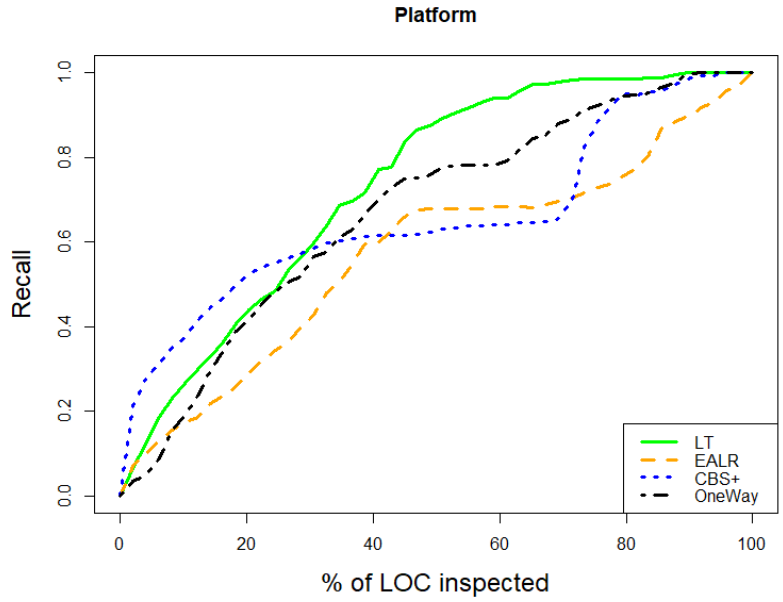

Fig. 3 The Alberg diagram of different prediction models when evaluated with the Platform project under time-wise cross-validation

Table 19 The average P_{opt} and P_{opt}@20% achieved by different prediction models under different evaluation settings

Evaluation Setting	Measure	Model			
Evaluation Setting	Measure	LT	EALR	CBS+	OneWay
Time-Wise	Popt	**0.712**	0.572	0.646	0.699
	Popt@20%	0.327	0.265	**0.426**	0.312
10-Times 10-Fold	Popt	0.762	0.569	0.663	**0.766**
	Popt@20%	0.358	0.249	**0.484**	0.378
Cross-Project	Popt	0.773	0.516	0.637	**0.780**
	Popt@20%	0.354	0.189	**0.453**	0.364

a prediction model and the amount of inspected LOC. The Optimal line in Figure 2 represents the case that all changes are sorted in ascending order by defect-density. The Worst line in the figure represents the case that all changes are sorted in descending order by defect-density. The actual prediction model should outperform the Random model and try to get close to the Optimal model. Given a prediction model m, its P_{opt} can be formally defined as:

$$P_{opt}(m) = 1 - \frac{Area(Optimal) - Area(m)}{Area(Optimal) - Area(worst)} \quad (7)$$

The function $Area(x)$ calculates the area under the curve corresponding to the prediction model x. Thus, a larger P_{opt} value means a smaller difference between the prediction model and the optimal model. In general, the output of each prediction model is a ranking list of changes, and P_{opt} evaluates the overall ranking in terms of *Recall*. We did not evaluate our model using P_{opt} in our previous work (Huang et al., 2017b) because it considers the performance of a prediction model when all the changes need to be inspected, which conflicts with the "effort aware" scenario considered in our study. However, we can adapt P_{opt} to only consider the area under the curve when the inspection cost does not exceed 20% of the total LOC, and we call it as P_{opt}@20%. Thus, to conduct a comprehensive comparison, in this paper, we evaluate the performance of LT, EALR, CBS+ and OneWay in terms of P_{opt} and P_{opt}@20%.

Table 19 presents the average results across the six projects for each prediction model under different evaluation settings. In general, LT and OneWay performs better in terms of P_{opt}, while CBS+ performs better in terms of P_{opt}@20%. The reason is that, using CBS+, developers can find more defects at the early stage but they would waste a lot of effort to inspect large changes at the middle stage of the whole inspection process. As an example, we plot the Alberg diagram of different prediction models when evaluated with the Platform project under time-wise cross-validation, as shown in Figure 3. From this figure, we can see that the *Recall* of CBS+ increases faster than the other models in the early stage (i.e., when inspecting less than 25% of all LOC). However, when we continuously increase the inspection cost (i.e., from 25% to 70%), the *Recall* of CBS+ increases much slower. Finally, in the next stage

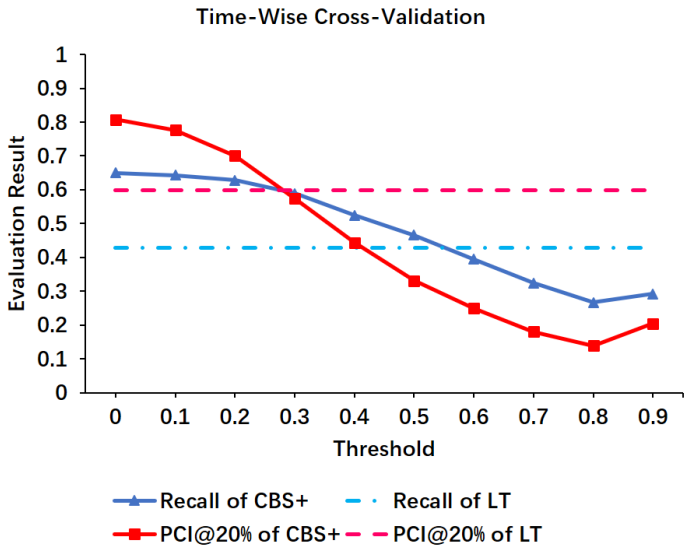

Fig. 4 The average Recall and PCI@20% of CBS+ when varying the threshold λ for time-wise cross-validation

(i.e., from 70% to 80%), the *Recall* of CBS+ is boosted again and almost 90% of all defective changes would have been inspected after this stage.

Such phenomenon is caused by a key step in CBS+, in which it divides all changes into two ranking lists: the first list contains the changes that are predicted as defective, while the second list contains the changes that are predicted as non-defective. Thus, in the early stage, following CBS+ recommendation, developers can inspect a lot of small but highly defect-prone changes, which contributes to the increase of *Recall*. However, it is unavoidable that some changes that are very large would also be put into the first ranking list, since larger changes are more likely to be defective (Kamei et al., 2013). Although these large changes are ranked at the bottom of the first ranking list, they still have a higher priority than those changes in the second ranking list. Thus, they would eventually be inspected in the middle stage of the inspection process, which requires a lot of effort cost. Finally, when we start inspecting the second list of changes, the small changes with high defect-proneness are also inspected first, which helps to boost the *Recall* again.

As a summary, CBS+ achieves a better P_{opt} if we only consider a limited amount of effort cost (e.g., inspecting 20% of all LOC), while LT and OneWay achieve a better P_{opt} if all changes need to be inspected.

8.4 The Tradeoff Between *Recall* and *PCI@20%*

One of the key steps in CBS+ is to divide all changes into two lists. The first list contains the changes that are predicted as defective, while the second list

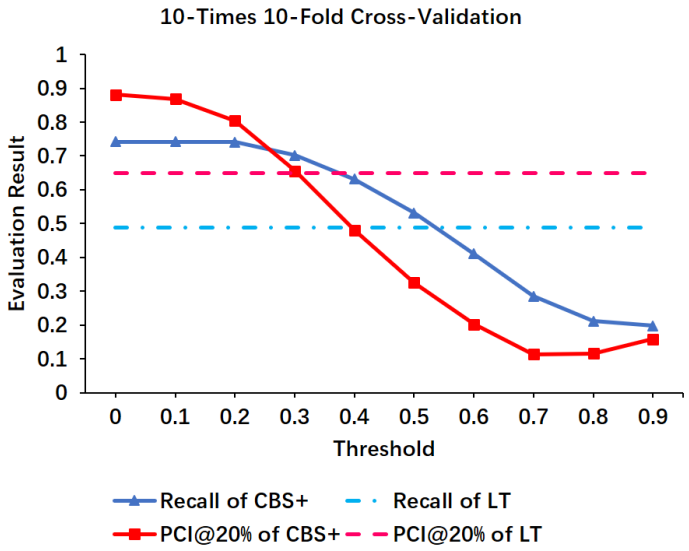

Fig. 5 The average Recall and PCI@20% of CBS+ when varying the threshold λ for 10-times 10-fold cross-validation

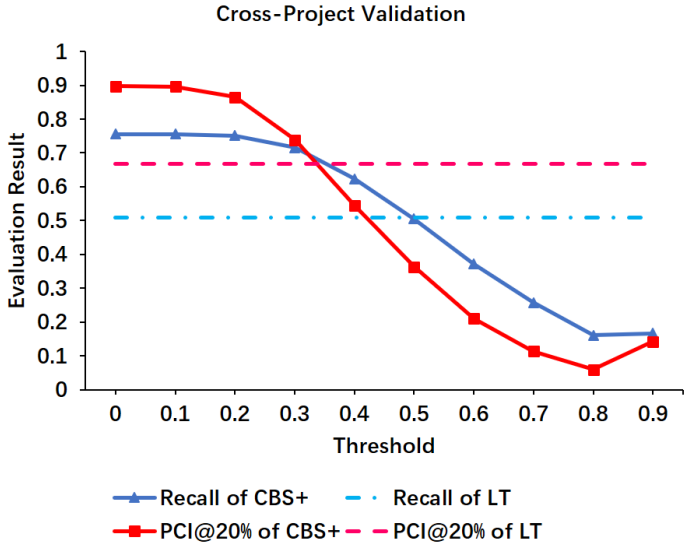

Fig. 6 The average Recall and PCI@20% of CBS+ when varying the threshold λ for cross-project validation

Table 20 The average results achieved by SBS for three different settings, compared with the results achieved by CBS+ with λ set to 0 (denoted as CBS*)

Measure	Model	Evaluation Setting		
		Time-Wise	10-Times 10-Fold	Cross-Project
Recall	SBS	**0.659**	0.725	0.749
	CBS*	0.650	**0.742**	**0.756**
Precision	SBS	0.171	0.179	0.181
	CBS*	**0.181**	**0.185**	**0.183**
F1-score	SBS	0.261	0.277	0.281
	CBS*	**0.270**	**0.285**	**0.284**
PCI@20%	SBS	0.855	0.891	0.903
	CBS*	**0.807**	**0.881**	**0.897**
IFA	SBS	68	320	3203
	CBS*	**1**	**0**	**1**

contains the changes that are predicted as non-defective. Thus, the contents of the two lists are directly decided by the threshold value λ, which is used to decide whether a change is defective given its defect proneness predicted by the underlying logistic classifier. Note that changes in the first ranking list are always ranked higher than changes in the second ranking list. Thus, the threshold value λ directly decides the inspection order. Given a limited budget, only changes that appear early in the second list would be inspected.

For example, when we set $\lambda = 0.9$, only a few changes would be predicted as defective and put into the first ranking list. In such case, many of them would be large changes (i.e., they modified many LOC) since large changes are more likely to be predicted with a high defect-proneness. Inspecting these large changes would cost a lot of effort while they just contribute a little to *Recall*. Thus, we would achieve a low *Recall* in this case, since we are only allowed to inspect 20% of all the modified LOC.

On the other hand, if we set $\lambda = 0.1$, most of the changes would be predicted as defective and put into the first ranking list. In this case, we are able to first inspect many small but also relatively defect-prone changes. Thus, we could spend most of our budget to inspect changes with higher defect-density, which helps us to get a much higher *Recall*. However, the drawback is that we need to inspect a lot of changes, which means the *PCI@20%* would be rather high.

In general, a lower threshold would lead to more changes being identified as defective, and vice versa. If there are more candidate changes in the defective list, then we would be able to inspect more changes after the sorting operation and the *PCI@20%* would increase. As a result, the *Recall* would also increase since more inspected changes bring more chances to find defective changes. Although the optimal model should achieve a high *Recall* while keeping the *PCI@20%* as low as possible, our model allows developers to balance the tradeoff between *Recall* and *PCI@20%* by tuning the threshold. For

example, if developers are not sensitive to frequent context switches, it would be acceptable to sacrifice *PCI@20%* and achieve a higher *Recall* by setting a lower threshold. Note that both LT and EALR are not tunable, since LT is just sorting by a fixed metric, while EALR also predicts a fixed score for each change which is used for sorting.

To investigate the impact of different threshold settings on the effectiveness of CBS+, we measure the *Recall* and *PCI@20%* of CBS+ when varying the threshold λ from 0 to 0.9 with a step of 0.1. Note that we would get the same results when λ is set to 0 or 1, since all changes are predicted as the same labels in both cases (i.e., defective when $\lambda = 0$ or non-defective when $\lambda = 1$). Thus, we omit the results when λ is set to 1.

Figures 4 - 6 plot the corresponding results for three different evaluation settings. As a comparison, we also plot the *Recall* and *PCI@20%* of LT in these figures. In general, when increasing the threshold λ, both *Recall* and *PCI@20%* would decrease. One exception is that the *PCI@20%* no longer decreases when λ is set to 0.9. In this case, only a few changes would be predicted as defective changes for inspection and there would be much budget left for inspecting those changes that are predicted as non-defective. By performing these additional inspections, we can find more defective changes with the budget left; however, this sacrifices both *PCI@20%* (i.e., we inspected more changes) and *Precision* (i.e., more changes that are non-defective would be inspected). This highlights the tradeoff between *Recall*, *PCI@20%* and *Precision*.

Besides, in RQ3, we have shown that CBS+ with the default threshold setting (i.e., $\lambda = 0.5$) can significantly outperforms LT in terms of *PCI@20%* and achieves comparable results in terms of *Recall*. If we set a lower threshold (i.e., $\lambda = 0.4$), then CBS+ would significantly outperforms LT in terms of both *Recall* and *PCI@20%*, which means that CBS+ would dominate LT in this case. Thus, the results show that CBS+ is more flexible in use. For example, suppose there is a list of new changes that are not labeled as defective or not. The project team could set a target value of *PCI@20%* that is acceptable for developers, based on historical data or the preference of developers. Since calculating *PCI@20%* does not require the true label of each change, we can easily adjust the value of λ to help developers find as many defective changes as possible while satisfying the requirement of *PCI@20%*. Another possibility is to employ an automatic parameter tuning algorithm to decide a suitable value of λ, which we leave as future work.

Another interesting finding is that when λ is set to 0, CBS+ can achieve a high *Recall* in every evaluation setting (i.e., the *Recall* ranges between 0.65 and 0.75), which is much better than those achieved by both supervised and unsupervised models in previous studies (Kamei et al., 2013; Yang et al., 2016). In this case, all changes would be predicted as defective and sorted together by the ratio between a change's defect proneness and its size. Intuitively, this sorting operation can also be done in an unsupervised way, where all changes are directly sorted in descending order by a change's size. We denote this special unsupervised model as SBS (i.e., Sort-By-Size). The only difference between SBS and Yang et al.'s unsupervised model is that SBS chooses a

change's size (i.e., LA+LD) as the metric for sorting while Yang et al. chooses the metric LT for sorting. Table 20 presents the average results achieved by SBS for three different settings, compared with the results achieved by CBS+ with λ set to 0 (denoted as CBS*). The results show that both CBS* and SBS achieve a high *Recall* for three different settings. In general, CBS* performs slightly better than SBS in terms of *Precision*, *F1-score* and *PCI@20%*, and performs much better in terms of *IFA*. However, we still argue that both CBS* and SBS are not practical in use even though they can achieve a high *Recall*, since they require developers to inspect at least 80% of all changes, which goes against the requirement of "effort-aware".

More recently, Zhou et al. (2018) reported that simple unsupervised models can achieve a prediction performance comparable or even superior to most of the existing supervised models in cross-project defect prediction. Specifically, they evaluated the performance of two unsupervised models, namely ManualDown and ManualUp. ManualDown considers larger modules as more defect-prone, while ManualUp considers smaller modules as more defect-prone. These two models are based on the observation of many previous studies, which pointed out that larger modules tend to have more defects but have lower defect density (Nagappan et al., 2006a,b; Nam and Kim, 2015; Thongmak and Muenchaisri, 2003). Thus, considering the effort-aware setting, the ManualUp model would find more defective modules and achieve a high *Recall* for cross-project defect prediction. This model can also be easily applied for effort-aware just-in-time defect prediction, and its working process is exactly the same with SBS (i.e., sort changes by size and inspect smaller changes first). However, as discussed in the previous paragraph, although ManualUp (or SBS) achieves a high *Recall*, it requires developers to inspect at least 80% of all changes, which goes against the requirement of "effort-aware".

8.5 Improving the underlying classifier of CBS+

By default, we use logistic regression (denoted as Logistic) which was originally used by Kamei et al. (2013) for defect prediction as the underlying classifier of CBS+. However, as pointed out by Ghotra et al. (2015), the choice of different classification techniques would have a significant impact on the performance of defect prediction models. Thus, if we choose a different classification technique to build the underlying classifier of CBS+, we might get different results. To investigate this impact, we examine the performance of CBS+ with five different underlying classifiers, including Random Forest (RF), Sequential Minimal Optimization (SMO), k-Nearest Neighbor (kNN), J48 and Naive Bayes (NB). These classifiers are widely used in previous software engineering studies and they belong to different families of classification techniques. For example, SMO belongs to the family of support vector machines (SVMs), while J48 belongs to the family of decision trees.

Table 21 presents the average results across the six projects for each prediction model. The results show that kNN and SMO performs badly in terms of

Table 21 Performance of CBS+ with different underlying classifiers

Measure	Model	Evaluation Setting		
		Time-Wise	10-Times 10-Fold	Cross-Project
Recall	Logistic	0.466	0.531	0.505
	RF	**0.476**	**0.582**	0.**546**
	SMO	0.448	0.520	0.483
	kNN	0.474	0.542	0.518
	J48	0.468	0.555	0.545
	NB	0.405	0.411	0.382
Precision	Logistic	0.281	0.334	0.316
	RF	0.267	0.331	0.301
	SMO	0.287	0.332	0.325
	kNN	0.238	0.279	0.251
	J48	0.266	0.332	0.308
	NB	**0.299**	**0.348**	**0.354**
F1-score	Logistic	**0.332**	0.392	0.338
	RF	0.325	**0.404**	0.350
	SMO	0.325	0.384	0.338
	kNN	0.295	0.350	0.314
	J48	0.309	0.398	**0.353**
	NB	0.323	0.355	0.331
PCI@20%	Logistic	0.331	0.325	0.363
	RF	0.368	0.355	0.389
	SMO	0.321	0.328	0.342
	kNN	0.414	0.403	0.435
	J48	0.372	0.339	0.381
	NB	**0.271**	**0.243**	**0.235**
IFA	Logistic	**1**	**0**	**1**
	RF	**1**	1	6
	SMO	13	71	659
	kNN	17	70	572
	J48	**1**	4	43
	NB	7	1	2

IFA and they cannot outperform Logistic in terms of other evaluation measures by a substantial margin. Although NB achieves the best results in terms of *Precision* and *PCI@20%* for all evaluation settings, it sacrifices a lot in terms of *Recall*. J48 performs relatively bad in terms of *IFA* under cross-project setting, but it achieves comparable results in terms of other evaluation measures when compared with Logistic. RF is the most prominent one among all the selected classifiers. It achieves the best *Recall* for all evaluation settings and achieves comparable results in terms of *IFA* when compared with Logistic. The only drawback of RF is that it sacrifices a bit in terms of *PCI@20%*.

In summary, we recommend future research to use Logistic or RF as the underlying classifier of CBS+. Besides, our implementation of these classi-

fiers is based on Weka (Hall et al., 2009), and we directly use the default hyper-parameter setting provided by Weka API. Although previous studies, e.g., Tantithamthavorn et al. (2016) and Agrawal and Menzies (2018), have shown that tuning the classifier's hyper-parameters would achieve significant improvements for defect prediction, we leave this as a future work because of the following reasons:

1. In our work, there are multiple evaluation metrics, and it is difficult to decide which evaluation metric is the most important to be the optimization target of a tuning algorithm. Thus, the hyper-parameter tuning problem would become a multi-objective optimization problem, which is more complex than the optimization problem for a single evaluation metric considered in many previous studies. Designing and experimenting with algorithms for this problem is beyond the scope of this paper.
2. The key point in our paper is to investigate whether using supervised models is still a good choice when compared with unsupervised models. We have demonstrated that supervised models (e.g., EALR and CBS+) outperform unsupervised models in terms of *PCI@20%* and *IFA*. Thus, we have sufficiently explored this key point. If tuning is done well, the only differences in the results are the deltas between the performance of the supervised and unsupervised models. Besides, while tuning the hyper-parameters would achieve better results for supervised models, the tuning process would be highly time consuming and make supervised models less practical.

8.6 The impact of inspecting too many changes

In this paper, we propose the evaluation measure *PCI@20%* and we argue that if a prediction model requires developers to inspect too many changes, the frequent context switch between different changes may increase the actual time and effort spent. To further investigate the impact of inspecting too many changes in practice, we conducted a survey with professional developers from two IT companies in China, named Insigma Global Service[3], and Hengtian[4]. Insigma Global Service is an outsourcing company which has more than 500 employees, and it mainly does outsourcing projects for Chinese vendors (e.g., Chinese commercial banks, Alibaba, and Baidu). Hengtian is also an outsourcing company which has more than 2,000 employees, and it mainly does outsourcing projects for US and European corporations (e.g., State Street Bank, Cisco, and Reuters). We sent emails to 141 developers who had experience in code review and we received 54 replies. These 54 developers vary in job roles, such as testers, front-end developers, back-end developers, algorithm engineers, mobile app developers, etc. The years of their experience in software development vary from 2 years to 8 years, with an average of 4.3 years. Our survey has two questions, as shown below:

[3] "Insigma Global Service," http://www.insigmaservice.com/.
[4] "Hengtian," http://www.hengtiansoft.com/.

- Q1: Suppose a number of code changes are supposed to be potentially defective, and we divide these changes into two groups, namely Group A and Group B. There are 30 changes in Group A and these changes *in total* modified 300 LOC. There are 10 changes in Group B and these changes *in total* also modified 300 LOC. For each group, we assign one developer to inspect these changes one by one. Among the following options, which one do you think is more reasonable?
  1) The time cost of Group A is substantially larger than Group B
  2) The time cost of Group A is almost the same with Group B
  3) The time cost of Group B is substantially larger than Group A
- Q2: Can you explain why for your answer in Q1? (optional)

Note that in Q1, the number of changes in Group A is triple as many as that of Group B, which follows our finding in RQ1, i.e., LT requires developers to inspect more than twice or even triple as many changes as those required by using EALR.

Among the responses, 37 (69%) of the developers chose the first option (i.e., Group A requires more effort cost). 21 of them described their reasons, and we summarized four major reasons, as follows:

- Additional cost of context switches, e.g.,
  - *When inspecting multiple changes, for each change, you need to open and close different files, and you have to quickly switch your mind from one function to another one which might be completely different. These pieces of time should not be ignored.*
  - *I never continuously review that number of changes in a single day. Wouldn't that be quite boring? I don't think I have that patience to check many changes that are just "possibly defective"...*
- Additional cost of inspecting a broader range of affected files, e.g.,
  - *Group A has more changes and different change may affect different files. When reviewing a code change, aside from the modified LOC, you also need to check corresponding functions or files. Thus, the real number of code to be reviewed in Group A should be much larger than Group B.*
- Additional cost of bug localization, e.g.,
  - *The purpose of code review is to find defects and improve code quality. Since you already told me these changes might have bugs, I would be more careful to locate the bug for each change. Also, I don't think the cost of locating a bug is simply related to the number of modified LOC. It is possible to spent more time to locate the bug for a small change from my experience.*
- Additional cost of dealing conflicts or dependencies, e.g.,
  - *Group A has more changes involved. Some of them may have conflicts or dependencies. When you rollback a defective change, you might need to address these conflicts or dependencies and this would introduce much more time cost.*

6 (11%) of the developers chose the second option (i.e., the two groups almost have the same effort cost). 3 of them described their reasons, and we summarized two major reasons, as follows:

- It is hard to judge the real cost, e.g.,
  - *The information given in Q1 is not enough to estimate the time cost. I don't think the cost of code review can be simply estimated with LOC...*
  - *I'm not sure how to compare the cost... I see that Group A has more changes to review. However, sometimes I spend more time to review a small set of changes while sometimes I don't.*
- Both groups have additional cost, e.g.,
  - *Group A has more changes to review and this may require checking a lot of different files or functions. Group B has less changes but each change modified more LOC and this may require more effort to understand the logic of each change.*

11 (20%) of the developers chose the third option (i.e., Group A requires more effort cost). 6 of them described their reasons, and we summarized two major reasons, as follows:

- The major cost comes from program comprehension, e.g.,
  - *It would be more difficult to understand the logic of the corresponding code if a change modified more LOC. I think this would be the major cost for code review. Especially, if the number of modified LOC is really large, the review cost would be much much larger.*
  - *IMHO, I don't like reviewing a change that modified too many LOC. It would be time consuming to understand what it has done and locate where the bug is.*
- The additional cost of inspecting the code context of a single large change, e.g.,
  - *If a change modified many LOC, these LOC might affect different parts of the file. To understand the modified code, you also need to check the code nearby and this would increase the time cost of code review.*

As a summary, the majority of the developers agree that inspecting too many changes would introduce additional effort cost. Another interesting finding is that a lot of developers pointed out that the real effort cost to review a code change cannot be simply estimated with the number of LOC modified by the change. However, almost all of the current studies in the literature assume that the effort cost is linearly correlated with change size. Thus, we recommend future research to design new quantitative criteria to better capture the real effort cost involved in code review.

8.7 Implications

8.7.1 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.

8.7.2 Implications for Researchers

Both the studies by Yang et al. (2016) and Kamei et al. (2013) 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.

9 Threats To Validity

9.1 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 (Yang et al., 2016). 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.

9.2 External Validity

The external validity relates to the quality and generalizability of our dataset. Our dataset is originally provided by Kamei et al., who used the SZZ algorithm (Śliwerski et al., 2005) to automatically build the ground truth labels of all changes. However, as reported by the authors of the SZZ algorithm and other related studies (Kim et al., 2006; da Costa et al., 2017; Neto et al., 2018), in some cases, the SZZ algorithm cannot accurately locate the change that truly introduces the bug. As a result, the SZZ algorithm itself may introduce false positives (i.e., non-defective changes marked as defective) and miss a number of truly defective changes. Thus, the errors introduced by the SZZ algorithm may affect the results of our study. On the other hand, the dataset contains 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.

9.3 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 (Arisholm et al., 2007; Rahman et al., 2012; Jiang et al., 2013; Shihab et al., 2013; Valdivia Garcia and Shihab, 2014; Huang et al., 2017a). 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.

10 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. (2013) and Yang et al. (2016) for providing us the datasets and source code used in their study. Finally, to enable other researchers replicate and extend our study, we have published the replication package in Zenodo[5]. This research was partially supported by the National Key Research and Development Program of China (2018YFB1003904) and NSFC Program (No. 61602403).

[5] https://zenodo.org/record/1432582#.W6YyU2gzaUl

References

Abdi H (2007) Bonferroni and šidák corrections for multiple comparisons. Encyclopedia of measurement and statistics 3:103–107

Agrawal A, Menzies T (2018) Is better data better than better data miners?: on the benefits of tuning smote for defect prediction. In: Proceedings of the 40th International Conference on Software Engineering, ACM, pp 1050–1061

Arisholm E, Briand LC, Fuglerud M (2007) Data mining techniques for building fault-proneness models in telecom java software. In: The 18th IEEE International Symposium on Software Reliability (ISSRE'07), IEEE, pp 215–224

Arisholm E, Briand LC, Johannessen EB (2010) A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. Journal of Systems and Software 83(1):2–17

Cliff N (1996) Ordinal methods for behavioral data analysis. Lawrence Erlbaum Associates

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

D'Ambros M, Lanza M, Robbes R (2010) An extensive comparison of bug prediction approaches. In: Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, IEEE, pp 31–41

Fu W, Menzies T (2017) Revisiting unsupervised learning for defect prediction. In: Proceedings of the 2017 25th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, p to appear

Ghotra B, McIntosh S, Hassan AE (2015) 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, pp 789–800

Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Transactions on software engineering 26(7):653–661

Guo PJ, Zimmermann T, Nagappan N, Murphy B (2010) Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In: Software Engineering, 2010 ACM/IEEE 32nd International Conference on, IEEE, vol 1, pp 495–504

Gyimothy T, Ferenc R, Siket I (2005) Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Transactions on Software engineering 31(10):897–910

Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The weka data mining software: an update. ACM SIGKDD explorations newsletter 11(1):10–18

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

Hamill M, Goseva-Popstojanova K (2009) Common trends in software fault and failure data. IEEE Transactions on Software Engineering 35(4):484–496

Han J, Pei J, Kamber M (2011) Data mining: concepts and techniques. Elsevier

Hassan AE (2009) Predicting faults using the complexity of code changes. In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, pp 78–88

Hintze JL, Nelson RD (1998) Violin plots: a box plot-density trace synergism. The American Statistician 52(2):181–184

Huang Q, Shihab E, Xia X, Lo D, Li S (2017a) Identifying self-admitted technical debt in open source projects using text mining. Empirical Software Engineering pp 1–34

Huang Q, Xia X, Lo D (2017b) Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In: Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on, IEEE

Jiang T, Tan L, Kim S (2013) Personalized defect prediction. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, IEEE, pp 279–289

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

Kim S, Zimmermann T, Pan K, James Jr E, et al. (2006) Automatic identification of bug-introducing changes. In: null, IEEE, pp 81–90

Kim S, Whitehead Jr EJ, Zhang Y (2008) Classifying software changes: Clean or buggy? IEEE Transactions on Software Engineering 34(2):181–196

Kochhar PS, Xia X, Lo D, Li S (2016) Practitioners' expectations on automated fault localization. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, pp 165–176

Koru AG, Zhang D, El Emam K, Liu H (2009) An investigation into the functional form of the size-defect relationship for software modules. IEEE Transactions on Software Engineering 35(2):293–304

Koru G, Liu H, Zhang D, El Emam K (2010) Testing the theory of relative defect proneness for closed-source software. Empirical Software Engineering 15(6):577–598

Li PL, Herbsleb J, Shaw M, Robinson B (2006) 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, pp 413–422

Matsumoto S, Kamei Y, Monden A, Matsumoto Ki, Nakamura M (2010) An analysis of developer metrics for fault prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, ACM, p 18

Mende T, Koschke R (2010) Effort-aware defect prediction models. In: Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on, IEEE, pp 107–116

Menzies T, Di Stefano JS (2004) How good is your blind spot sampling policy. In: High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE

International Symposium on, IEEE, pp 129–138
Menzies T, Milton Z, Turhan B, Cukic B, Jiang Y, Bener A (2010) Defect prediction from static code features: current results, limitations, new approaches. Automated Software Engineering 17(4):375–407
Meyer AN, Fritz T, Murphy GC, Zimmermann T (2014) Software developers' perceptions of productivity. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, pp 19–29
Mockus A, Weiss DM (2000) Predicting risk of software changes. Bell Labs Technical Journal 5(2):169–180
Moser R, Pedrycz W, Succi G (2008) 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, pp 181–190
Munson JC, Khoshgoftaar TM (1992) The detection of fault-prone programs. IEEE Transactions on Software Engineering 18(5):423–433
Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, IEEE, pp 284–292
Nagappan N, Ball T, Murphy B (2006a) Using historical in-process and product metrics for early estimation of software failures. In: Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on, IEEE, pp 62–74
Nagappan N, Ball T, Zeller A (2006b) Mining metrics to predict component failures. In: Proceedings of the 28th international conference on Software engineering, ACM, pp 452–461
Nam J, Kim S (2015) Clami: Defect prediction on unlabeled datasets (t). In: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, IEEE, pp 452–463
Neto EC, da Costa DA, Kulesza U (2018) The impact of refactoring changes on the szz algorithm: An empirical study. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 380–390
Ostrand TJ, Weyuker EJ, Bell RM (2004) Where the bugs are. In: ACM SIGSOFT Software Engineering Notes, ACM, vol 29, pp 86–96
Parnin C, Orso A (2011) Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 international symposium on software testing and analysis, ACM, pp 199–209
Purushothaman R, Perry DE (2005) Toward understanding the rhetoric of small source code changes. IEEE Transactions on Software Engineering 31(6):511–526
Rahman F, Devanbu P (2013) How, and why, process metrics are better. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, pp 432–441
Rahman F, Posnett D, Devanbu P (2012) Recalling the imprecision of cross-project defect prediction. In: Proceedings of the ACM SIGSOFT 20th In-

ternational Symposium on the Foundations of Software Engineering, ACM, p 61

Shihab E, Hassan AE, Adams B, Jiang ZM (2012) 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, p 62

Shihab E, Ihara A, Kamei Y, Ibrahim WM, Ohira M, Adams B, Hassan AE, Matsumoto Ki (2013) Studying re-opened bugs in open source software. Empirical Software Engineering 18(5):1005–1042

Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: ACM sigsoft software engineering notes, ACM, vol 30, pp 1–5

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2016) Automated parameter optimization of classification techniques for defect prediction models. In: Proceedings of the 38th International Conference on Software Engineering, ACM, pp 321–332

Thongmak M, Muenchaisri P (2003) Predicting faulty classes using design metrics with discriminant analysis. In: Software Engineering Research and Practice, pp 621–627

Turhan B, Menzies T, Bener AB, Di Stefano J (2009) On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering 14(5):540–578

Valdivia Garcia H, Shihab E (2014) Characterizing and predicting blocking bugs in open source projects. In: Proceedings of the 11th working conference on mining software repositories, ACM, pp 72–81

Wilcoxon F (1945) Individual comparisons by ranking methods. Biometrics bulletin 1(6):80–83

Xia X, Bao L, Lo D, Li S (2016a) "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, pp 267–278

Xia X, Lo D, Pan SJ, Nagappan N, Wang X (2016b) Hydra: Massively compositional model for cross-project defect prediction. IEEE Transactions on Software Engineering 42(10):977–998

Xia X, Lo D, Wang X, Yang X (2016c) Collective personalized change classification with multiobjective search. IEEE Transactions on Reliability 65(4):1810–1829

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

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

Yang X, Lo D, Xia X, Sun J (2017) Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. Information and Software Tech-

nology 87:206–220
Yang Y, Zhou Y, Liu J, Zhao Y, Lu H, Xu L, Xu B, Leung H (2016) 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, pp 157–168
Yin Z, Yuan D, Zhou Y, Pasupathy S, Bairavasundaram L (2011) How do fixes become bugs? In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ACM, pp 26–36
Zhou Y, Yang Y, Lu H, Chen L, Li Y, Zhao Y, Qian J, Xu B (2018) How far we have progressed in the journey? an examination of cross-project defect prediction. ACM Transactions on Software Engineering and Methodology (TOSEM) 27(1):1
Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B (2009) 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, pp 91–100