Effort-Aware Just-in-Time Defect Prediction: Simple Unsupervised Models Could Be Better Than Supervised Models

Yibiao Yang[1], Yuming Zhou[1*], Jinping Liu[1], Yangyang Zhao[1], Hongmin Lu[1], Lei Xu[1], Baowen Xu[1], and Hareton Leung[2]

[1]Department of Computer Science and Technology, Nanjing University, China

[2]Department of Computing, Hong Kong Polytechnic University, Hong Kong, China

ABSTRACT

Unsupervised models do not require the defect data to build the prediction models and hence incur a low building cost and gain a wide application range. Consequently, it would be more desirable for practitioners to apply unsupervised models in effort-aware just-in-time (JIT) defect prediction if they can predict defect-inducing changes well. However, little is currently known on their prediction effectiveness in this context. We aim to investigate the predictive power of simple unsupervised models in effort-aware JIT defect prediction, especially compared with the state-of-the-art supervised models in the recent literature. We first use the most commonly used change metrics to build simple unsupervised models. Then, we compare these unsupervised models with the state-of-the-art supervised models under cross-validation, time-wise-cross-validation, and across-project prediction settings to determine whether they are of practical value. The experimental results, from open-source software systems, show that many simple unsupervised models perform better than the state-of-the-art supervised models in effort-aware JIT defect prediction.

CCS Concepts

•**Software and its engineering** → **Risk management;** Software development process management;

Keywords

Defect, prediction, changes, just-in-time, effort-aware

1. INTRODUCTION

Recent years have seen an increasing interest in just-in-time (JIT) defect prediction, as it enables developers to identify defect-inducing changes at check-in time [7, 13]. A defect-inducing change is a software change (i.e. a single or several consecutive commits in a given period of time) that introduce one or several defects into the source code in a software system [37]. Compared with traditional defect prediction at module (e.g. package, file, or class) level, JIT defect prediction is a fine granularity defect prediction. As stated by Kamei et al. [13], it allows developers to inspect an order of magnitude smaller number of SLOC (source lines of code) to find latent defects. This could provide large savings in effort over traditional coarser granularity defect predictions. In particular, JIT defect prediction can be performed at check-in time [13]. This allows developers to inspect the code changes for finding the latent defects when the change details are still fresh in their minds. As a result, it is possible to find the latent defects faster. Furthermore, compared with conventional non-effort-aware defect prediction, effort-aware JIT defect prediction takes into account the effort required to inspect the modified code for a change [13]. Consequently, effort-aware JIT defect prediction would be more practical for practitioners, as it enables them to find more latent defects per unit code inspection effort. Currently, there is a significant strand of interest in developing effective effort-aware JIT defect prediction models [7, 13].

Kamei et al. [13] leveraged supervised method (i.e. the linear regression method) to build an effort-aware JIT defect prediction model. To the best of our knowledge, this is the first time to introduce effort-aware concept into JIT defect prediction. Their results showed that the proposed supervised model was effective in effort-aware performance evaluation compared with the random model. This work is significant, as it could help find more defect-inducing changes per unit code inspection effort. In practice, however, it is often time-consuming and expensive to collect the defect data (used as the dependent variable) to build supervised models. Furthermore, for many new projects, the defect data are unavailable, in which supervised models are not applicable. Different from supervised models, unsupervised models do not need the defect data to build the defect prediction models. Therefore, for practitioners, it would be more desirable to apply unsupervised models if they can predict defects well. According to recent studies [16, 17, 18, 19, 26, 42], simple unsupervised models, such as the ManualUp model in which modules are prioritized in ascending order according to code size, are effective in the context of effort-aware defect prediction at coarser granularity. Up till now, however, little is known on the practical value of simple unsupervised models in the context of effort-aware JIT defect prediction.

The main contributions of this paper are as follows:

*Corresponding author: zhouyuming@nju.edu.cn.

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950353

(1) We investigate the predictive effectiveness of unsupervised models in effort-aware JIT defect prediction, which is an important topic in the area of defect prediction.

(2) We perform an in-depth evaluation on the simple unsupervised techniques (i.e. the unsupervised models in Section 3.2) under three prediction settings (i.e. cross-validation, time-wise-cross-validation, and across-project prediction).

(3) We compare simple unsupervised models with the state-of-the-art supervised models in recent literature [8, 13]. The experimental results show that many simple unsupervised models perform significantly better than the state-of-the-art supervised models.

The rest of this paper is organized as follows. Section 2 introduces the background on defect prediction. Section 3 describes the employed experimental methodology. Section 4 provides the experimental setup in our study, including the subject projects and the data sets. Section 5 reports in detail the experimental results. Section 6 examines the threats to validity of our study. Section 7 concludes the paper and outlines directions for future work.

2. BACKGROUND

In this section, we first introduce the background on just-in-time defect prediction and then effort-aware defect prediction. Finally, we describe the existing work on the application of unsupervised models in traditional defect prediction.

2.1 Just-in-time Defect Prediction

The origin of JIT (just-in-time) defect prediction can be traced back to Mockus and Weiss [28], who used a number of change metrics to predict the probability of changes to be defect-inducing changes. For practitioners, JIT defect prediction is of more practical value compared with traditional defect predictions at module (e.g. file or class) level. As stated by Kamei et al. [13], the reason is not only because the predictions can considerably narrow down the code to be inspected for finding the latent defects, but also because the predictions can be made at check-in time when the change details are still fresh in the minds of the developers. In particular, in traditional defect predictions, after a module is predicted as defect-prone, it may be difficult to find the specific developer, who is most familiar with the code, to inspect the module to find the latent defects. However, in JIT defect predictions, it is easy to find such a developer to inspect the predicted defect-prone change, as each change is associated with a particular developer.

In the last decade, Mockus and Weisśs study led to an increasing interest in JIT defect predictions. Śliwerski et al. [37] studied defect-inducing changes in two open-source software systems and found that the changes committed on Friday had a higher probability to be defect-inducing changes. Eyolfson et al. [6] studied the influence of committed time and developer experience on the existence of defects in a software change. Their results showed that changes committed between midnight and 4AM were more likely to be defect-inducing than the changes committed between 7AM and noon. Yin et al. [40] studied the bug-fixing changes in several open-source software systems. They found that around 14.2% to 24.8% bug-fixing changes for post-release bugs were defect-inducing and the concurrency defects were the most difficult to be fixed. Kim et al. [15] used numerous features extracted from various sources such as change metadata, source code, and change log messages to build prediction models to predict defect-inducing changes. Their results showed that defect-inducing changes can be predicted at 60% recall and 61% precision on average.

2.2 Effort-aware Defect Prediction

Although the above-mentioned results were encouraging, they did not take into account the effort required for quality assurance when applying the JIT defect prediction models in practice. Arisholm et al. [1] pointed out that, when locating defects, it was important to take into account the cost-effectiveness of using defect prediction models to focus on verification and validation activities. This viewpoint has been recently taken by many module-level defect prediction studies [12, 25, 34, 33, 32, 39, 42]. Inspired by this viewpoint, Kamei et al. applied effort-aware evaluation to JIT defect predictions [13]. In their study, Kamei et al. used the total number of lines modified by a change as a measure of the effort required to inspect a change. In particular, they leveraged linear regression method to build the effort-aware JIT defect prediction model (called the EALR model). In their model, the dependent variable was $Y(x)/Effort(x)$, where $Effort(x)$ was the effort required for inspecting the change x and $Y(x)$ was 1 if x was defect inducing and 0 otherwise. The results showed that, on average, the EALR model can detect 35% of all defect-inducing changes, when using 20% of the effort required to inspect all changes. As such, Kamei et al. believed that effort-aware JIT defect prediction was able to focus on the most risky changes and hence could reduce the costs of developing high-quality software [13].

2.3 Unsupervised Models in Traditional Defect Prediction

In the last decades, supervised models have been the dominant defect prediction paradigm in traditional defect prediction at module (e.g. file or class) level [1, 11, 13, 15, 25, 26, 28, 33, 32, 42, 43, 44]. In order to build a supervised model, we need to collect the defect data such as the number of defects or the labeled information (buggy or not-buggy) for each module. For practitioners, it may be expensive to apply such supervised models in practice. The reason for this is that it is generally time-consuming and costly to collect the defect data. Furthermore, supervised defect prediction models cannot be built if the defect data are unavailable. This is especially true when a new type of projects is developed or when historical defect data have not been collected.

Compared with supervised models, unsupervised models do not need the defect data. Due to this advantage, recent years have seen an increasing effort devoted to apply unsupervised modeling techniques to build defect prediction models [3, 26, 41, 42]. In practice, however, it is more important to know the effort-aware prediction performance of unsupervised defect prediction models. To this end, many researchers investigate whether unsupervised models are still effective when taking into account the effort to inspect the modules that are predicted as "defect-prone". Koru et al. [16, 17, 18, 19] suggested that smaller modules should be inspected first, as more defects would be detected per unit code inspection effort. The reason was that the relationship between module size and the number of defects was found not linear but logarithmic [16, 18], indicating that defect-proneness increased at a slower rate as module size increased. Menzies et al. [26] used the ManualUp model to name the "smaller modules inspected first" strategy by Koru et al. In their experiments,

Menzies et al. did find that the ManualUp model had a good effort-aware prediction performance. Their results were further confirmed by Zhou et al.'s study [42], in which the ManualUp model was found even competitive to the regular supervised logistic regression model. All these studies show that, in traditional defect prediction, unsupervised models perform well under effort-aware evaluation.

3. RRESEARCH METHODOLODY

In this section, we first introduce the investigated independent and dependent variables. Then, we describe the simple unsupervised models under study and present the supervised models which will be used as the baseline models against. Next, we give the research questions. After that, we provide the performance indicators for evaluating the effectiveness of defect prediction models in effort-aware JIT defect prediction. Finally, we give the data analysis method used in this study.

3.1 Dependent and Independent Variables

The dependent variable in this study is a binary variable. If a code change is a defect-inducing change, the dependent variable is set to 1 and 0 otherwise.

Table 1: Summarization 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, i.e. $-\Sigma_{k-1}^{n} p_k log_2 p_k$, where n is the number of files touched by the change and pk is the ratio of the touched code in the k-th file to the total touched code
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	The number of developers that changed the files
AGE	The average time interval (in days) between the last and the change over the files that are touched
NUC	The number of unique last changes to the files
EXP	Developers experience, i.e. the number of changes
REXP	Recent developer experience, i.e. the total experience of the developer in terms of changes, weighted by their age
SEXP	Developer experience on a subsystem, i.e. the number of changes the developer made in the past to the subsystems

The independent variables used in this study consist of fourteen change metrics. Table 1 summarizes these change metrics, including the metric name, the description, and the source. These fourteen metrics can be classified into the following five dimensions: diffusion, size, purpose, history, and experience. The diffusion dimension consists of NS, ND, NF, and Entropy, which characterize the distribution of a change. As stated by Kamei et al. [13], it is believed that a highly distributed change is more likely to be a defect-inducing change. The size dimension leverages LA, LD, and LT to characterize the size of a change, in which a larger change is expected to have a higher likelihood of being a defect-inducing change [30, 36]. The purpose dimension consists of only FIX. In the literature, there is a belief that a defect-fixing change is more likely to introduce a new defect [40]. The history dimension consists of NDEV, AGE, and NUC. It is believed that a defect is more likely to be introduced by a change if the touched files have been modified by more developers, by more recent changes, or by more unique last changes [4, 9, 11, 22]. The experience dimension consists of EXP, REXP, and SEXP, in which the experience of the developer of a change is expected to have a negative correlation to the likelihood of introducing a defect into the code by the change. In other words, if the current change is made by a more experienced developer, it is less likely that a defect will be introduced. Note that, all these change metrics are the same as those used in Kamei et al.'s study.

3.2 Simple Unsupervised Models

In this study, we leverage change metrics to build simple unsupervised models. As stated by Monden et al. [29], to adopt defect prediction models, one needs to consider not only their prediction effectiveness but also the significant cost required for metrics collection and modeling themselves. A recent investigation from Google developers further shows that a prerequisite for deploying a defect prediction model in a large company such as Google is that it must be able to scale to large source repositories [21]. Therefore, we only take into account those unsupervised defect prediction models that have a low application cost (including metrics collection cost and modeling cost) and a good scalability. More specifically, our study will investigate the following unsupervised defect prediction models. For each of the change metrics (except LA, and LD), we build an unsupervised model that ranks changes in descendant order according to the reciprocal of their corresponding raw metric values. This idea is inspired by Koru and Menzies et al.'s finding that smaller modules are proportionally more defect-prone and hence should be inspected first [19, 25]. In our study, we expect that "smaller" changes tend to be more proportionally defect-prone. More formally, for each change metric M, the corresponding model is $R(c) = 1/M(c)$. Here, c represents a change and R is the predicted risk value. For a given system, the changes will be ranked in descendant order according to the predicted risk value R. In this context, changes with smaller change metric values will be ranked higher.

Note that, under each of the above-mentioned simple unsupervised models, it is possible that two changes have the same predicted risk values, i.e. they have a tied rank. In our study, if there is a tied rank according to the predicted risk values, the change with a lower defect density will be ranked higher. Furthermore, if there is still a tied rank according to the defect densities, the change with a larger change size will be ranked higher. In this way, we will obtain simple unsupervised models that have the "worst" predictive performance (theoretically) in effort-aware just-in-time defect prediction [14]. In our study, we investigate the predictive power of those "worst" simple unsupervised models. If our experimental results show that those "worst" simple unsupervised models are competitive to the supervised models, we will have confidence that simple unsupervised models are of practical value for practitioners in effort-aware just-in-time defect prediction.

As can be seen, there are 12 simple unsupervised models, which involve a low application cost and can be efficiently applied to large source repositories.

3.3 The Supervised Models

The supervised models are summarized in Table 2. These supervised models are categorized into six groups: "Function", "Lazy", "Rule", "Bayes", "Tree", and "Ensemble". The supervised models in the "Function" group are the regression models and the neural networks. "Lazy" are the supervised models based on lazy learning. "Rule" and "Tree" respec-

tively represent the rule based and the decision tree based supervised models. "Ensemble" are those supervised ensemble models which are built with multiple base leaners. The Naive Bayes is a probability-based technique. In [13], Kamei et al. using the linear regression model to build the effort-aware JIT defect prediction model (i.e. the EALR model). The EALR model is the state-of-the-art supervised model in effort-aware JIT defect prediction. Besides, we also include other supervised techniques (i.e. the models in Table 2 except the EALR model) as the baseline models. The reasons are two-folds. First, they are the most commonly used supervised techniques in defect prediction studies [8, 10, 20, 23, 25]. Second, a recent literature [8] using most of them (except for the Random Forest) to revisit their impact on the performance of defect prediction.

Table 2: Overview of the supervised models

Family	Model	Abbreviation
Function	Linear Regression	EALR
	Simple Logistic	SL
	Radial basis functions network	RBFNet
	Sequential Minimal Optimization	SMO
Lazy	K-Nearest Neighbour	IBk
Rule	Propositional rule	JRip
	Ripple down rules	Ridor
Bayes	Naïve Bayes	NB
Tree	J48	J48
	Logistic Model Tree	LMT
	Random Forest	RF
Ensemble	Bagging	BG+LMT, BG+NB, BG+SL, BG+SMO, and BG+J48
	Adaboost	AB+LMT, AB+NB, AB+SL, AB+SMO, and AB+J48
	Rotation Forest	RF+LMT, RF+NB, RF+SL, RF+SMO, and RF+J48
	Random Subspace	RS+LMT, RS+NB, RS+SL, RS+SMO, and RS+J48

In this study, we use the same method as Kamei et al. [13] to build the EALR model. As stated in Section 2.2, Y(x)/Effort(x) was used as the dependent variable in the EALR model. For the other supervised models, we use the same method as Ghotra et al. [8]. More specifically, Y(x) was used as the dependent variable for these supervised models. Consistent with Ghotra et al. [8], we use the same parameters to build these supervised models. For example, the K-Nearest Neighbor requires K most similar training example for classifying an instance. In [8], Ghotra et al. found that K = 8 performed best than other options (i.e. 2, 4, 6, and 16). As such, we also use K = 8 to build the K-Nearest Neighbor. In EALR model, Kamei et al. used the under sampling method to deal with the imbalanced data set and then removed the most highly correlated factors to deal with collinearity. In consistence with Kamei et al.'s study [13], we use exactly the same method to deal with imbalanced data set and collinearity.

3.4 Research Questions

We investigate the following three research questions to determine the practical value of simple unsupervised models:

***RQ1**: How well do simple unsupervised models predict defect-inducing changes when compared with the state-of-the-art supervised models in cross-validation?*

***RQ2**: How well do simple unsupervised models predict defect-inducing changes when compared with the state-of-the-art supervised models in time-wise-cross-validation?*

***RQ3**: How well do simple unsupervised models predict defect-inducing changes when compared with the state-of-the-art supervised models in across-project prediction?*

The purposes of $RQ1$, $RQ2$, and $RQ3$ are to compare simple unsupervised models with the state-of-the-art supervised models with respect to three different prediction settings (i.e. the cross-validation, time-wise-cross-validation, and across-project prediction) to determine how well they predict defect-inducing changes. Since unsupervised models do not leverage the buggy or not-buggy label information to build the prediction models, they are not expected to perform better than the supervised models. However, if unsupervised models are not much worse than the supervised models, it is still a good choice for practitioners to apply them because they have a lower building cost, a wider application range, and a higher efficiency. To the best of our knowledge, little is currently known on these research questions from the viewpoint of unsupervised models in the literature. Our study attempts to fill this gap by an in-depth investigation into simple unsupervised models in the context of effort-aware JIT defect prediction.

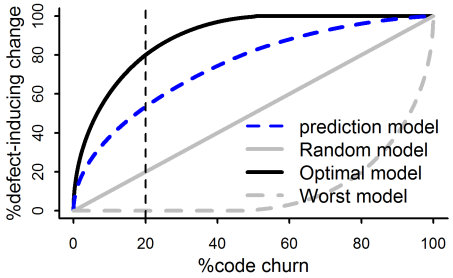

Figure 1: Code-churn-based Alberg diagram

3.5 Performance Indicators

When evaluating the predictive effectiveness of a JIT defect prediction model, we take into account the effort required to inspect those changes predicted as defect-prone to find whether they are defect-inducing changes. Consistent with Kamei et al. [13], we use the code churn (i.e. the total number of lines added and deleted) by a change as a proxy of the effort required to inspect the change. In [13], Kamei et al. used ACC and P_{opt} to evaluate the effort-aware performance for the EALR model. ACC denotes the recall of defect-inducing changes when using 20% of the entire effort required to inspect all changes to inspect the top ranked changes. P_{opt} is the normalized version of the effort-aware performance indicator originally introduced by Mende and Koschke [24]. The P_{opt} is based on the concept of the "code-churn-based" Alberg diagram.

Figure 1 is an example "code-churn-based" Alberg diagram showing the performances of a prediction model m. In this diagram, the x-axis and y-axis are respectively the cumulative percentage of code churn of the changes (i.e. the percentage of effort) and the cumulative percentage of defect-inducing changes found in selected changes. To compute P_{opt}, two additional curves are included: the "optimal" model and the "worst" model. In the "optimal" model and the "worst" model, changes are respectively sorted in decreasing and ascending order according to their actual defect densities. According to [13], P_{opt} can be formally defined as:

$$P_{opt}(m) = 1 - \frac{Area(optimal) - Area(m)}{Area(optimal) - Area(worst)}$$

Here, Area(optimal) and Area(worst) is the area under the curve corresponding to the best and the worst model, respectively. Note that both ACC and P_{opt} are applicable to supervised models as well as unsupervised models.

3.6 Data Analysis Method

Figure 2 provides an overview of our data analysis method. As can be seen, in order to obtain an adequate and realistic assessment, we examine the three RQs under the following three prediction settings: 10 times 10-fold cross-validation, time-wise-cross-validation, and across-project prediction.

10 times 10-fold cross-validation is performed within the same project. At each 10-fold cross-validation, we first randomize the data set. Then, we divide the data set into 10 parts of approximately equal size. After that, each part is used as a testing data set to evaluate the effectiveness of the prediction model built on the remainder of the data set (i.e. the training data set). The entire process is then repeated 10 times to alleviate possible sampling bias in random splits. Consequently, each model has $10 \times 10 = 100$ prediction effectiveness values.

Time-wise-cross-validation is also performed within the same project, in which the chronological order of changes is considered. This is the method followed in [37]. For each project, we first rank the changes in chronological order according to the commit date. Then, all changes committed within the same month are grouped into the same part. Assume the changes in a project are grouped into n parts, we use the following approach for the time-wise prediction. We build a prediction model m on the combination of part i and part i+1 and then apply m to predict changes in part i+4 and i+5 ($1 \leq i \leq n-5$). As such, each training set and test set will have changes committed with two consecutive months. The reasons for this setting are four-fold. First, the release cycle of most projects is typically $6 \sim 8$ weeks [5]. Second, it can make sure that each training set and test set will have a gap of two months. Third, two consecutive months can make sure that each training set will have enough instances, which is important for supervised models. Forth, it allows us to have enough runs for each project. If a project has changes of n months, this method will produce $n-5$ prediction effectiveness values for each model.

Across-project prediction is performed across different projects. We use a model trained on one project (i.e. the training data set) to predict defect-proneness in another project (i.e. the testing data set) [33, 43]. Given n projects, this method will produce $n \times (n-1)$ prediction effectiveness values for each model. In this study, we use six projects as the subject projects. Therefore, each prediction model will produce $6 \times (6-1) = 30$ prediction effectiveness values.

Note that, the unsupervised models only use the change metrics in testing data to build the prediction models. In this study, we also apply cross-validation, time-wise-cross-validation, and across-project prediction settings to the unsupervised models. This allows the unsupervised models to use the same testing data as those supervised models, thus making a fair comparison on their prediction performance.

When investigating $RQ1$, $RQ2$, and $RQ3$, we use the B-H corrected p-values from the Wilcoxon signed-rank test to examine whether there is a significant difference in the prediction effectiveness between the unsupervised and supervised models. In particular, we use the Benjamini-Hochberg (BH) corrected p-values to examine whether a difference is statistically significant at the significance level of 0.05 [2]. If the statistical test shows a significant difference, we then use the Cliff's δ to examine whether the magnitude of the difference is practically important from the viewpoint of practical application [1]. By convention, the magnitude of the difference is considered trivial ($|\delta| < 0.147$), small ($0.147 \leq |\delta| < 0.33$), moderate ($0.33 \leq |\delta| < 0.474$), or large ($\geq 0.474$) [35].

Furthermore, similar to Ghotra et al. [8], we use Scott-Knott test [14, 27] to group the supervised and unsupervised prediction models to examine whether there exist some models outperform others. The Scott-Knott test uses hierarchical cluster analysis method to divide the prediction models into two groups according to the mean performance (i.e. the P_{opt} and the ACC with respect to different runs for each model). If the difference between the divided groups is statistically significant, Scott-Knott will recursively divide each group into two different groups. The test will terminate when groups can no longer divided into statistically distinct groups.

4. EXPERIMENTAL SETUP

In this section, we first introduce the subject projects and then describe the data sets collected from these projects.

4.1 Subject Projects

In this study, we use the same open-source subject projects as used in Kamei et al.'s study [13]. More specifically, we use the following six projects to investigate the predictive power of simple unsupervised models in effort-aware JIT defect prediction: Bugzilla (BUG), Columba (COL), Eclipse JDT (JDT), Eclipse Platform (PLA), Mozilla (MOZ), and PostgreSQL (POS). Bugzilla is a well-known web-based bug tracking system. Columba is a powerful mail management tool. Eclipse JDT is the Eclipse Java Development Tools, which is a set of plug-ins that add the capabilities of a full-featured Java IDE to the Eclipse platform. Mozilla is a well-known and widely used open-source web browser. PostgreSQL is a powerful, open-source object-relational database system. As stated by Kamei et al. [13], these six projects are large, well-known, and long-lived projects, which cover a wide range of domains and sizes. In this sense, it is appropriate to use these projects to investigate simple unsupervised models in JIT defect prediction.

4.2 Data Sets

The data sets from these six projects used in this study are shared by Kamei et al. and are available online. As mentioned by Kamei et al. [13], these data are gathered by combining the change information mined from the CVS repositories of these projects with the corresponding bug reports. More specifically, the data for Bugzilla and Mozilla were gathered from the data provided by MSR 2007 Mining Challenge. The data for the Eclipse JDT and Platform were gathered from the data provided by the MSR 2008 Mining Challenge. The data for Columba and PostgreSQL were gathered from the official CVS repository.

Table 3 summarizes the six data sets used in this study. The first column and the second column are respectively the subject data set name and the period of time for collecting the changes. The third to the sixth columns respectively report the total number of changes, the percentage of defect-inducing changes, the average LOC per change, and the number of files modified in a code change. As can be seen, for each data set, defects concentrated in a small percentage

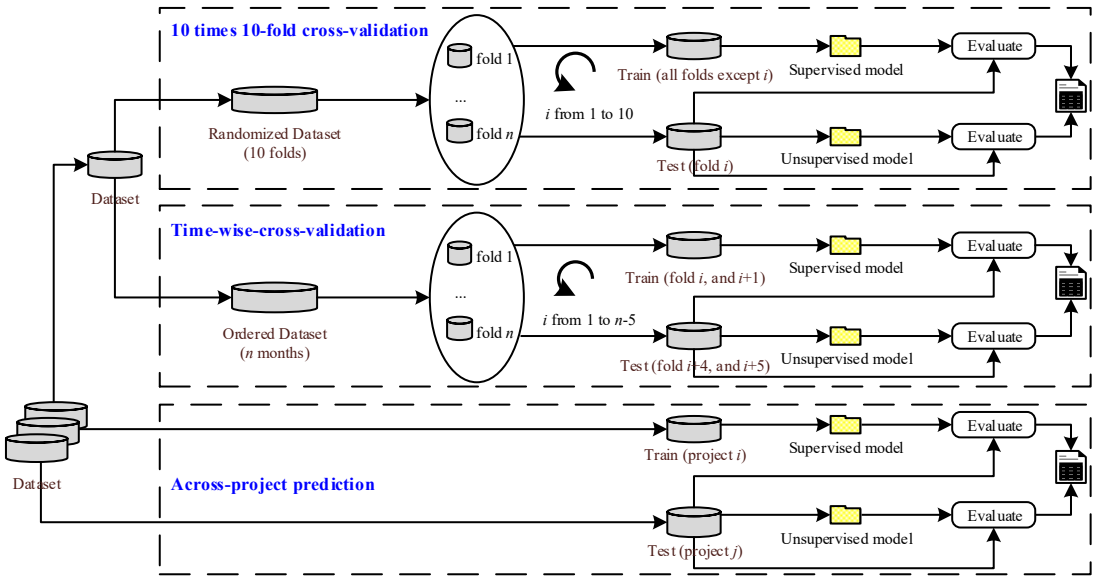

Figure 2: Overview of the three prediction settings

Table 3: Summarization of studied data sets

Project	Period	#change	%defect-inducing change	mean LOC per change	#modified files per change
BUG	08/1998-12/2006	4620	36%	37.5	2.3
COL	11/2002-07/2006	4455	14%	149.4	6.2
JDT	05/2001-12/2007	35386	14%	71.4	4.3
PLA	20/2001-12/2007	64250	5%	72.2	4.3
MOZ	01/2000-12/2006	98275	25%	106.5	5.3
POS	07/1996-05/2010	20431	20%	101.3	4.5

of changes (around 5% $\sim$ 36% of all changes).

5. EXPERIMENTAL RESULTS

In this section, we report the experimental results.

5.1 10 Times 10-fold Cross-validation

Figure 3 and Figure 4 respectively employ the box-plot to describe the distributions of P_{opt} and ACC obtained from 10 times 10-fold cross-validation for the supervised models and simple unsupervised models for the overall result over the six data sets. For each model, the box-plot shows the median (the horizontal line within the box), the 25th percentile (the lower side of the box), and the 75th percentile (the upper side of the box). In Figure 3 and Figure 4, the horizontal dotted lines respectively represent the median performance of the best supervised model. In particular, there are blue, red, and black box-plots. A blue box indicates that: (1) the corresponding simple model performs significantly better than the best supervised model according to the Wilcoxon signed-rank test (i.e. the BH corrected p-value is less than 0.05); and (2) the magnitude of the difference between the corresponding simple model and the best supervised model is not trivial according to Cliff's δ (i.e. $|\delta| \geq 0.147$). A red box indicates that: (1) the corresponding simple model performs significantly worse than the best supervised model; and (2) the magnitude of the difference between the corresponding simple model and the best supervised model is not trivial. A black box indicates that: (1) the difference between the corresponding simple model and the best supervised model is not significant; or (2) the magnitude of the difference between the corresponding simple model and the best supervised model is trivial.

From Figure 3 and Figure 4, we have the following observations. First, according to P_{opt}, the best supervised model is the EALR model, which performs significantly better than all the other supervised models. However, the ND/EXP/REXP/SEXP unsupervised models have a performance similar to the EALR model and the NF/Entropy/LT-/NDEV/AGE/NUC unsupervised models perform significantly better than the EALR model. Second, according to ACC, the best supervised model is also the EALR model, which performs significantly better than the other supervised models except the RBFN model. However, the NDEV/NUC unsupervised models perform similar to the EALR model and the NF/Entropy/LT/AGE unsupervised models perform significantly better than the EALR model.

Figure 5 and Figure 6 respectively present the results from Scott-Knott test. In Figure 5 and Figure 6, the y-axis is the average performance. The blue labels indicate simple unsupervised models. The dotted lines represent groups divided by the Scott-Knott test. All models are ordered by their mean ranks over the six different projects. As can be seen, all models in the first group are the unsupervised models. The best supervised model (i.e. the EALR model) is in the second group. This indicates that, those unsupervised models in the first group significantly outperform the best supervised model.

Table 4 (a) and Table 5 (b) respectively summarize the median P_{opt} and ACC for the best supervised model (i.e. the EALR model) and the best four simple unsupervised models obtained from 10 times 10-fold cross-validation. In each table, for each simple unsupervised model, we show how often it performs significantly better (denoted by "$\surd$") or worse (denoted by "$\times$") than the best supervised model by the Wilcoxon's signed-rank test. The row "AVG" reports the average median over the six data sets. The row "W/T/L" reports

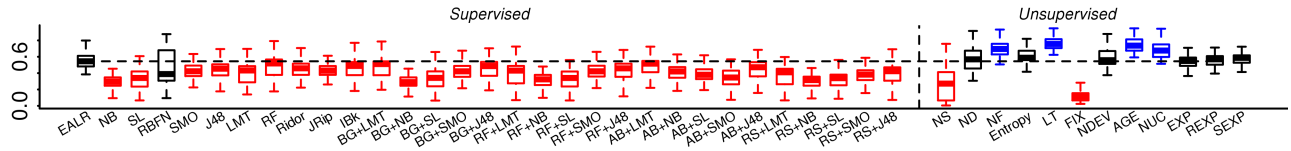

Figure 3: 10 times 10-fold cross-validation: performance comparison in terms of P_{opt}

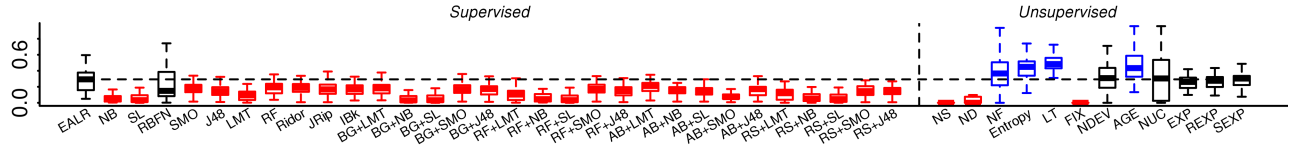

Figure 4: 10 times 10-fold cross-validation: performance comparison in terms of ACC

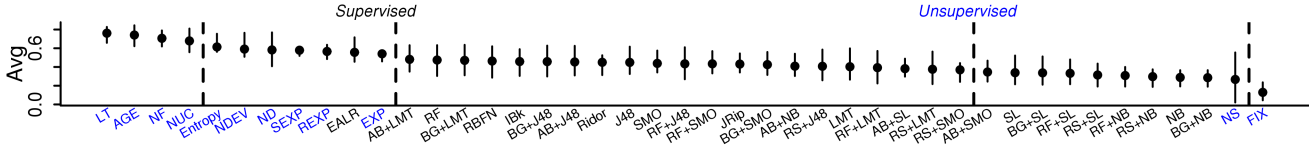

Figure 5: Scott-Knott test for 10 times 10-fold cross-validation in terms of P_{opt}

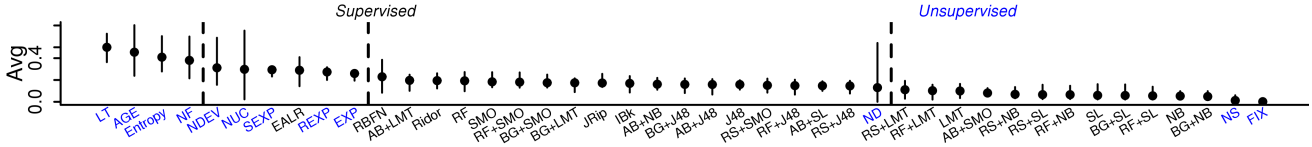

Figure 6: Scott-Knott test for 10 times 10-fold cross-validation in terms of ACC

the number of data sets for which the simple unsupervised model obtains a better, equal, and worse performance than the best supervised model. In particular, an entry in deep gray background indicates a large improvement and an entry in light gray background indicates a moderate improvement in terms of the Cliff's δ.

Table 4: 10 times 10-fold cross-validation: best supervised model vs. unsupervised models ($RQ1$)

(a) P_{opt}

	EALR	NF	NUC	LT	AGE
BUG	0.705	0.673	0.670	0.754	0.748√
COL	0.576	0.811√	0.822√	0.842√	0.856√
JDT	0.511	0.716√	0.679√	0.781√	0.747√
PLA	0.551	0.698√	0.640√	0.748√	0.707√
MOZ	0.453	0.614√	0.552√	0.644√	0.621√
POS	0.491	0.727√	0.714√	0.805√	0.768√
AVG	0.548	0.706	0.679	0.762	0.741
W/T/L	-	5/1/0	5/1/0	5/1/0	6/0/0

(b) ACC

	EALR	NF	Entropy	LT	AGE
BUG	0.384	0.256	0.267	0.477	0.413
COL	0.394	0.618	0.608√	0.626√	0.687√
JDT	0.217	0.370√	0.445√	0.523√	0.476√
PLA	0.303	0.369√	0.470√	0.459√	0.385√
MOZ	0.146	0.213√	0.326√	0.375√	0.244√
POS	0.281	0.416	0.366	0.519√	0.464√
AVG	0.287	0.374	0.414	0.497	0.445
W/T/L	-	3/3/0	4/2/0	5/1/0	5/1/0

From Table 4, we have the following observations. First, according to P_{opt}, the best four simple unsupervised models have average median P_{opt} ranging from 0.679 to 0.762, thus exhibiting a 24% to 39% improvement over the EALR model (average median P_{opt} = 0.548). In particular, the AGE unsupervised model performs significantly better than the EALR model in all six data sets and the magnitudes of the difference are large in five data sets. Second, according to ACC, the best four simple unsupervised models have average median ACC ranging from 0.374 to 0.497, thus exhibiting a 30% to 73% improvement over the EALR model (average median ACC = 0.287). In other words, when using the same effort to inspect the changes, these simple unsupervised models can detect 30% to 74% more defect-inducing changes than the best supervised model.

Overall, the above observations suggest that simple unsupervised models could perform better when compared with the state-of-the-art supervised models in effort-aware JIT defect prediction under 10 times 10-fold cross-validation.

5.2 Time-wise Cross-validation

Figure 7 and Figure 8 respectively employ the box-plot to describe the distributions of the P_{opt} and the ACC obtained from time-wise-cross-validation.

From Figure 7 and Figure 8, we have the following observations. First, according to P_{opt}, the best supervised model in the time-wise-cross-validation is the EALR model, which outperforms all the other supervised models. However, the Entropy/NDEV/SEXP unsupervised models have a performance similar to the EALR model and the NF/LT/AGE/NUC unsupervised models perform significantly better than the EALR model. Second, according to ACC, again, the best supervised model is the EALR model. However, the NF/Entropy/NDEV/REXP/SEXP models have a performance similar to the EALR model and the LT/AGE unsupervised models perform significantly better than the EALR model.

Figure 9 and Figure 10 respectively present the results from the Scott-Knott test for the supervised and unsupervised models with respect to P_{opt} and ACC obtained from time-wise-cross-validation. From Figure 9, we can see that both the first group and second group consist of two simple unsupervised models. The best supervised model (i.e. the EALR model) is in the third group, in which six unsupervised models are also included. This indicates that many simple unsupervised models are similarly to or even better than the best supervised model in terms of P_{opt}. From Figure 10, we can see that that the first group consists of two simple unsu-

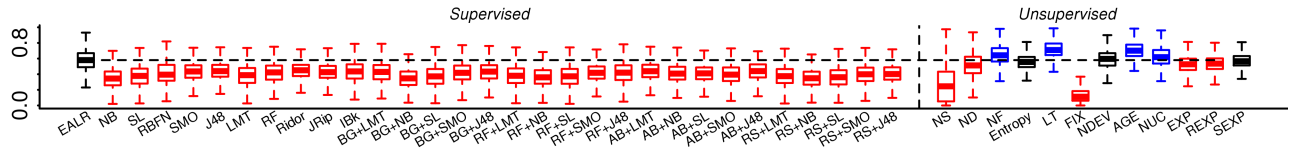

Figure 7: Time-wise-cross-validation: performance comparison in terms of P_{opt}

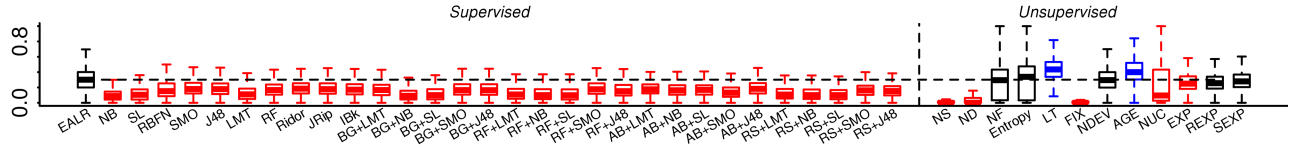

Figure 8: Time-wise-cross-validation: performance comparison in terms of ACC

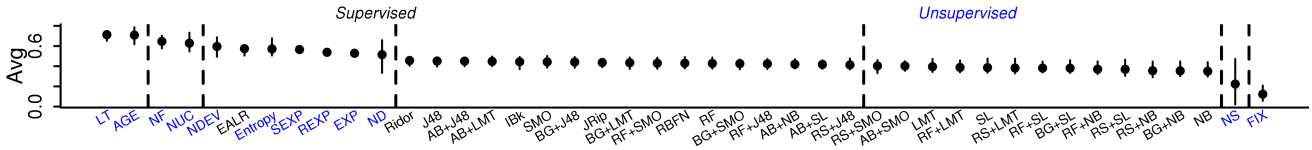

Figure 9: Scott-Knott test under time-wise-cross-validation in terms of P_{opt}

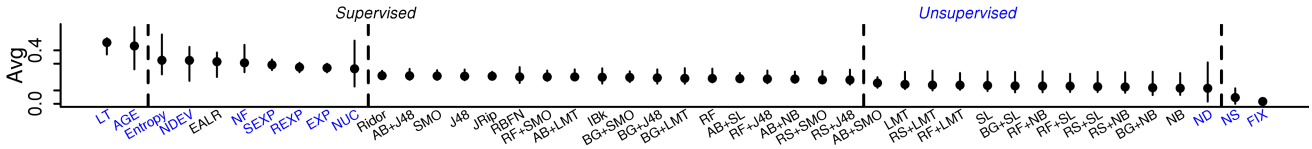

Figure 10: Scott-Knott test under time-wise-cross-validation in terms of ACC

pervised models. The best supervised model (i.e. the EALR model) is in the second group, in which seven unsupervised models are also included. This indicates that many simple unsupervised models are similarly to or even better than the best supervised model in terms of ACC.

Table 5: Time-wise-cross-validation: best supervised model vs. unsupervised models ($RQ2$)

	(a) P_{opt}			(b) ACC		
	EALR	LT	AGE	EALR	LT	AGE
BUG	0.594	0.721√	0.661√	0.286	0.449√	0.375√
COL	0.619	0.732√	0.786√	0.400	0.440√	0.568√
JDT	0.590	0.709√	0.685√	0.323	0.452√	0.408√
PLA	0.583	0.717√	0.709√	0.305	0.432√	0.429√
MOZ	0.498	0.651√	0.638√	0.180	0.363√	0.280√
POS	0.600	0.742√	0.731√	0.356	0.432√	0.426√
AVG	0.581	0.712	0.702	0.308	0.428	0.414
W/T/L	-	6/0/0	6/0/0	-	6/0/0	6/0/0

Table 5 (a) and Table 5 (b) respectively summarize the median P_{opt} and ACC for the best supervised model (i.e. the EALR model) and the best two simple unsupervised models obtained from time-wise-cross-validation. As can be seen, the LT/AGE unsupervised models perform significantly better than the EALR model in all six data sets, regardless of whether P_{opt} or ACC is considered. As indicated by the cells in grey background, there is a moderate to large improvement in most cases in terms of the Cliff's δ. According to P_{opt}, the best two simple unsupervised models exhibit more than 21% improvement over the EALR model (average median $P_{opt} = 0.581$). According to ACC, the best two simple unsupervised models exhibit more than 34% improvement over the EALR model (average median $ACC = 0.308$). In other words, when using the 20% effort to inspect all the changes, these simple unsupervised models can detect 34% more defect-inducing changes than the EALR model.

Overall, the above observations suggest that simple unsupervised models could perform better when compared with the state-of-the-art supervised models in effort-aware JIT defect prediction under time-wise-cross-validation.

5.3 Across-project Prediction

Figure 11 and Figure 12 respectively employ the box-plot to describe the distributions of P_{opt} and ACC obtained from across-project prediction for the supervised models and simple unsupervised models over the six data sets.

From Figure 11 and Figure 12 , we have the following observations. First, according to P_{opt}, the EARL model performs significantly better than all the other supervised models. However, the ND/EXP/REXP unsupervised models have a performance similar to the EALR model and the NF/Entropy/LT/NDEV/AGE/NUC/SEXP unsupervised models perform significantly better than the EALR model. Second, according to ACC, the EALR model is also perform significantly than all the other supervised models. However, the NDEV/NUC/EXP/REXP/SEXP unsupervised models have a performance similar to the EALR model and the NF/Entropy/LT/AGE unsupervised models perform significantly better than the EALR model.

Figure 13 and Figure 14 respectively present the results from Scott-Knott test for the supervised and unsupervised models with respect to P_{opt} and ACC obtained from across-project prediction. From Figure 13, we can see that all the four models in the first group are simple unsupervised models. The best supervised model (i.e. the EALR model) is in the second group, in which six simple unsupervised models are also included. This indicates that many simple unsupervised models are similarly to or even better than the best supervised model in terms of P_{opt}. From Figure 14, we can see that the models in the first, second, and third groups are all simple unsupervised models. The best supervised model is in the fourth group, in which four simple unsupervised models are also included. This indicates that many simple unsupervised models are similarly to or even better than the best supervised model in terms of ACC.

Table 6 shows the P_{opt} and ACC for the best supervised

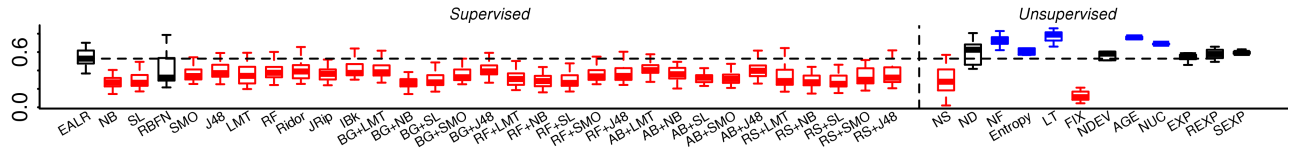

Figure 11: Across-project prediction: performance comparison in terms of P_{opt}

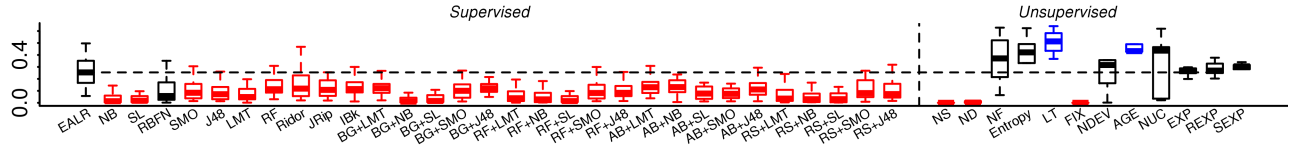

Figure 12: Across-project prediction: performance comparison in terms of ACC

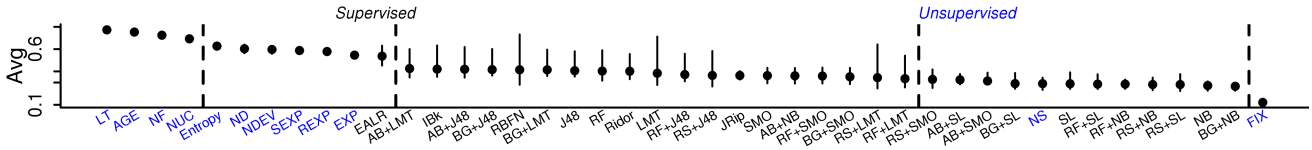

Figure 13: Scott-Knott test under across-project prediction in terms of P_{opt}

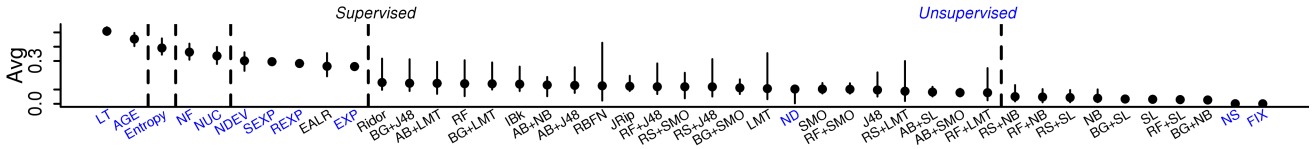

Figure 14: Scott-Knott test under across-project prediction in terms of ACC

model (i.e. the EALR model) and the best two simple unsupervised models obtained from across-project prediction. The first and the second columns are respectively the training project and the testing project. The row "AVG" reports for each prediction model the average P_{opt} and ACC.

Table 6: Across-project prediction: best supervised vs. unsupervised models ($RQ3$)

Train	Test	EALR		LT		AGE	
		P_{opt}	ACC	P_{opt}	ACC	P_{opt}	ACC
BUG	COL	0.701	0.497	0.858	0.641	0.868	0.702
	JDT	0.592	0.281	0.815	0.582	0.769	0.490
	PLA	0.695	0.429	0.770	0.494	0.740	0.427
	MOZ	0.563	0.219	0.660	0.367	0.622	0.240
	POS	0.603	0.349	0.810	0.533	0.762	0.432
COL	BUG	0.633	0.336	0.726	0.435	0.758	0.432
	JDT	0.418	0.113	0.815	0.582	0.769	0.490
	PLA	0.506	0.239	0.770	0.494	0.740	0.427
	MOZ	0.476	0.148	0.660	0.367	0.622	0.240
	POS	0.426	0.167	0.810	0.533	0.762	0.432
JDT	BUG	0.674	0.376	0.726	0.435	0.758	0.432
	COL	0.564	0.378	0.858	0.641	0.868	0.702
	PLA	0.555	0.277	0.770	0.494	0.740	0.427
	MOZ	0.502	0.168	0.660	0.367	0.622	0.240
	POS	0.514	0.299	0.810	0.533	0.762	0.432
PLA	BUG	0.685	0.367	0.726	0.435	0.758	0.432
	COL	0.564	0.363	0.858	0.641	0.868	0.702
	JDT	0.458	0.152	0.815	0.582	0.769	0.490
	MOZ	0.498	0.168	0.660	0.367	0.622	0.240
	POS	0.478	0.230	0.810	0.533	0.762	0.432
MOZ	BUG	0.619	0.35	0.726	0.435	0.758	0.432
	COL	0.482	0.247	0.858	0.641	0.868	0.702
	JDT	0.367	0.057	0.815	0.582	0.769	0.490
	PLA	0.414	0.173	0.770	0.494	0.740	0.427
	POS	0.382	0.135	0.810	0.533	0.762	0.432
POS	BUG	0.629	0.351	0.726	0.435	0.758	0.432
	COL	0.617	0.439	0.858	0.641	0.868	0.702
	JDT	0.459	0.159	0.815	0.582	0.769	0.490
	PLA	0.544	0.261	0.770	0.494	0.740	0.427
	MOZ	0.497	0.159	0.660	0.367	0.622	0.240
AVG		0.537	0.263	0.773	0.509	0.753	0.454

From Table 6, when comparing the simple unsupervised models against the EALR model, we have the following observations. First, the LT unsupervised model has a larger performance value in all entries for both P_{opt} and ACC. Second, the AGE model has a larger performance value in all entries for P_{opt}. For ACC, except one entry (i.e. the underlined 0.427), the AGE model also has a larger performance value. On average (see the "AVG" row), the two best simple unsupervised models exhibit more than 40% improvement in terms of P_{opt} and exhibit more than 70% improvement in terms of ACC.

Overall, the above observations suggest that simple unsupervised models could be better than the state-of-the-art supervised models in effort-aware JIT defect prediction under across-project prediction.

Table 7: Overall performance: simple unsupervised models vs. the best supervised model

	CV		TW-CV		AP	
	P_{opt}	ACC	P_{opt}	ACC	P_{opt}	ACC
NS	×	×	×	×	×	×
ND	≈	×	×	×	≈	×
NF	√	√	√	≈	√	√
Entropy	√	√	≈	≈	√	√
LT	√	√	√	√	√	√
FIX	×	×	×	×	×	×
NDEV	√	≈	≈	≈	√	≈
AGE	√	√	√	√	√	√
NUC	√	≈	√	×	√	≈
EXP	≈	≈	×	×	≈	≈
REXP	≈	≈	×	≈	≈	≈
SEXP	≈	≈	≈	≈	√	≈

√: (1) significantly better and (2) the magnitude is not trivial
×: (1) significantly worse and (2) the magnitude is not trivial
≈: (1) the difference is not significant or (2) magnitude is trivial

Table 7 summarizes the main results from Section 5.1, Section 5.2, and Section 5.3. As can be seen, the best two simple unsupervised models are the LT/AGE models as they perform significantly better than the best supervised model under all of the three prediction settings in terms of both P_{opt} and ACC. In addition, the NF/Entropy/NDEV/SEXP models perform similarly to or better than the best supervised model. Thus, we have a strong evidence to support that many simple unsupervised models perform well compared

with the state-of-the-art supervised models in effort-aware JIT defect prediction (indicated by P_{opt} and ACC). The above finding is very surprising, as it is in contrast with our original expectation that the-state-of-the-art supervised models should have a better performance. The reason for this expectation is that the state-of-the-art supervised models exploiting the defect data (i.e. the label information) to build the prediction model. From the above finding, we believe that it is a good choice for practitioners to apply simple unsupervised models in practice due to the low building cost and the wide application range.

6. THREATS TO VALIDITY

In this section, we analyze the most important threats to the construct, internal, and external validity of our study.

6.1 Construct Validity

The dependent variable used in this study is a binary variable indicating whether a change is defect-inducing. Our study used the data sets provided online by Kamei el al. [13]. As stated by Kamei et al. [13], they used the commonly used SZZ algorithm [37] to discover defect-inducing changes. However, the discovered defect-inducing changes may be incomplete, which is a potential threat to the construct validity of the dependent variable. Thus, the approaches to recovering missing links [31, 38] are required to improve the accuracy of the SZZ algorithm. Indeed, this is an inherent problem to most, if not all, studies that discover defect-inducing changes by mining software repositories, not unique to us. Nonetheless, this threat needs to be eliminated by using complete defect data in the future work. The independent variables used in this study are the commonly used change metrics. For their construct validity, previous research has investigated the degree to which they accurately measure the concepts they purport to measure [13]. In particular, each change metric has a clear definition and can be easily collected. In this sense, the construct validity of the independent variables should be acceptable.

6.2 Internal Validity

There are two potential threats to the internal validity. The first potential threat is from the specific cut-off value used for the performance indicator (i.e. the ACC). In our study, 0.2 is used as the cut-off value for the computation for recall of defect-inducing changes. The reasons are two-fold. First, 0.2 was used in Kamei et al.'s study, this enable us to directly compare our result with them. Second, 0.2 is the most commonly used cut-off value in the literature. However, it is unknown whether our conclusion depends on the chosen cut-off value. To eliminate this potential threat, we re-run all the analyses using the following typical cut-off values: 0.05, 0.10, and 0.15. We found our conclusion remained no change. The second potential threat is from the gap between the training and the test set in time-wise-cross-validation. In this study, we use two months as the gap. However, it is unknown whether our result depend on this gap. In order to eliminate this potential threat, we re-run all the analysis using the other gaps (2, 4, 6, and 12). We found our conclusion remained no change.

6.3 External Validity

The most important threat to the external validity of this study is that our results may not be generalized to other systems. In our experiments, we use six long-lived and widely used open-source software systems as the subject systems. The experimental results drawn from these subject systems are quite consistent. Furthermore, the data sets from these systems are large enough to draw statistically meaningful conclusions. We believe that our study makes a significant contribution to the software engineering body of empirical knowledge about effort-aware JIT defect prediction. Nonetheless, we do not claim that our findings can be generalized to all systems, as the subject systems under study might not be representative of systems in general. To mitigate this threat, there is a need to use a wide variety of systems to replicate our study in the future.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we perform an empirical study to investigate the predictive power of simple unsupervised models in effort-aware JIT defect prediction. Our experimental results from six industrial-size systems show that many simple unsupervised models perform well in predicting defect-inducing changes. In particular, contrary to the general expectation, we find that several simple unsupervised models perform better than the state-of-the-art supervised model reported by Kamei et al. [13]. The experimental results from the investigated six subject systems are quite consistent, regardless of whether 10 times 10-fold cross-validation, time-wise-cross-validation, or across-project prediction is considered. Our findings have important implications. For practitioners, simple unsupervised models are attractive alternative to supervised models in the context of effort-aware JIT defect prediction, as they have a lower building cost and a wider application range. This is especially true for those projects whose defect data are expensive to collect or even unavailable. For researchers, we strongly suggest that future JIT defect prediction research should use our simple unsupervised models as the baseline models for comparison when a novel prediction model is proposed.

Our study only investigates the actual usefulness of simple unsupervised models in effort-aware JIT defect prediction for open-source software systems. It is unclear whether they can be applied to closed-source software systems. In the future, an interesting work is hence to extend our current study to closed-source software systems.

8. REPEATABILITY

We provide all datasets and R scripts that used to conduct this study at http://ise.nju.edu.cn/yangyibiao/jit.html

9. ACKNOWLEDGMENTS

We are very grateful to Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi for sharing their data sets. This enables us to conduct this study. This work is supported by the National Key Basic Research and Development Program of China (2014CB340702), the National Natural Science Foundation of China (61432001, 91418202, 61272082, 61300051, 61321491, 61472178, and 91318301), the Natural Science Foundation of Jiangsu Province (BK20130014), the Hong Kong Competitive Earmarked Research Grant (PolyU5219/06E), the HongKong PolyU Grant (4-6934), and the program A for Outstanding PhD candidate of Nanjing University.

10. REFERENCES

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

[2] Y. Benjamini and Y. Hochberg. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300, Jan. 1995.

[3] P. Bishnu and V. Bhattacherjee. Software Fault Prediction Using Quad Tree-Based K-Means Clustering Algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 24(6):1146–1150, June 2012.

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

[5] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein. Time variance and defect prediction in software projects. *Empirical Software Engineering*, 17(4-5):348–389, Nov. 2011.

[6] J. Eyolfson, L. Tan, and P. Lam. Do Time of Day and Developer Experience Affect Commit Bugginess? MSR '11, pages 153–162, New York, NY, USA, 2011. ACM.

[7] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi. An Empirical Study of Just-in-time Defect Prediction Using Cross-project Models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 172–181, New York, NY, USA, 2014. ACM.

[8] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, volume 1, pages 789–800, May 2015.

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

[10] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, Nov. 2012.

[11] A. E. Hassan. Predicting Faults Using the Complexity of Code Changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

[12] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan. Revisiting Common Bug Prediction Findings Using Effort-aware Models. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

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

[14] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan. Prioritizing the Devices to Test Your App on: A Case Study of Android Game Apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 610–620, New York, NY, USA, 2014. ACM.

[15] S. Kim, E. Whitehead, and Y. Zhang. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, Mar. 2008.

[16] A. Koru, D. Zhang, K. El Emam, and H. Liu. An Investigation into the Functional Form of the Size-Defect Relationship for Software Modules. *IEEE Transactions on Software Engineering*, 35(2):293–304, Mar. 2009.

[17] A. Koru, D. Zhang, and H. Liu. Modeling the Effect of Size on Defect Proneness for Open-Source Software. pages 115–124, May 2007.

[18] A. G. Koru, K. E. Emam, D. Zhang, H. Liu, and D. Mathew. Theory of relative defect proneness. *Empirical Software Engineering*, 13(5):473–498, Oct. 2008.

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

[20] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, July 2008.

[21] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr. Does Bug Prediction Support Human Developers? Findings from a Google Case Study. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 372–381, Piscataway, NJ, USA, 2013. IEEE Press.

[22] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura. An analysis of developer metrics for fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 18:1–18:9, New York, NY, USA, 2010. ACM.

[23] T. Mende and R. Koschke. Revisiting the Evaluation of Defect Prediction Models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, PROMISE '09, pages 7:1–7:10, New York, NY, USA, 2009. ACM.

[24] T. Mende and R. Koschke. Effort-aware defect prediction models. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 107–116, Washington, DC, USA, 2010. IEEE Computer Society.

[25] T. Menzies, J. Greenwald, and A. Frank. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, Jan. 2007.

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

[27] N. Mittas and L. Angelis. Ranking and Clustering Software Cost Estimation Models through a Multiple Comparisons Algorithm. *IEEE Transactions on Software Engineering*, 39(4):537–551, Apr. 2013.

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

[29] A. Monden, T. Hayashi, S. Shinoda, K. Shirai, J. Yoshida, M. Barker, and K. Matsumoto. Assessing the Cost Effectiveness of Fault Prediction in Acceptance Testing. *IEEE Transactions on Software Engineering*, 39(10):1345–1357, Oct. 2013.

[30] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 284–292, New York, NY, USA, 2005. ACM.

[31] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Multi-layered Approach for Recovering Links Between Bug Reports and Fixes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 63:1–63:11, New York, NY, USA, 2012. ACM.

[32] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 432–441, Piscataway, NJ, USA, 2013. IEEE Press.

[33] F. Rahman, D. Posnett, and P. Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 61:1–61:11, New York, NY, USA, 2012. ACM.

[34] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. BugCache for Inspections: Hit or Miss? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 322–331, New York, NY, USA, 2011. ACM.

[35] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, pages 1–33, 2006.

[36] Y. Shin, A. Meneely, L. Williams, and J. Osborne. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, Nov. 2011.

[37] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, May 17, 2005*. ACM, 2005.

[38] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. ReLink: Recovering Links Between Bugs and Changes. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 15–25, New York, NY, USA, 2011. ACM.

[39] Y. Yang, Y. Zhou, H. Lu, L. Chen, Z. Chen, B. Xu, H. Leung, and Z. Zhang. Are Slice-Based Cohesion Metrics Actually Useful in Effort-Aware Post-Release Fault-Proneness Prediction? An Empirical Study. *IEEE Transactions on Software Engineering*, 41(4):331–357, Apr. 2015.

[40] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How Do Fixes Become Bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 26–36, New York, NY, USA, 2011. ACM.

[41] S. Zhong, T. M. Khoshgoftaar, and N. Seliya. Unsupervised learning for expert-based software quality estimation. In *Proceedings of the Eighth IEEE International Conference on High Assurance Systems Engineering*, HASE'04, pages 149–155, Washington, DC, USA, 2004. IEEE Computer Society.

[42] Y. Zhou, B. Xu, H. Leung, and L. Chen. An in-depth study of the potentially confounding effect of class size in fault prediction. *ACM Trans. Softw. Eng. Methodol.*, 23(1):10:1–10:51, Feb. 2014.

[43] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 91–100, New York, NY, USA, 2009. ACM.

[44] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–, Washington, DC, USA, 2007. IEEE Computer Society.