Code churn: A neglected metric in effort-aware just-in-time defect prediction

Jinping Liu*†, Yuming Zhou*‡, Yibiao Yang*‡, Hongmin Lu*, and Baowen Xu*
*State Key Laboratory for Novel Software Technology, Nanjing University, China
†School of Computer Science and Telecommunication Engineering, Jiangsu University
‡Corresponding author: {zhouyuming, yangyibiao}@nju.edu.cn

***Abstract*—Background: An increasing research effort has devoted to just-in-time (JIT) defect prediction. A recent study by Yang et al. at FSE'16 leveraged individual change metrics to build unsupervised JIT defect prediction model. They found that many unsupervised models performed similarly to or better than the state-of-the-art supervised models in effort-aware JIT defect prediction. Goal: In Yang et al.'s study, code churn (i.e. the change size of a code change) was neglected when building unsupervised defect prediction models. In this study, we aim to investigate the effectiveness of code churn based unsupervised defect prediction model in effort-aware JIT defect prediction. Methods: Consistent with Yang et al.'s work, we first use code churn to build a code churn based unsupervised model (CCUM). Then, we evaluate the prediction performance of CCUM against the state-of-the-art supervised and unsupervised models under the following three prediction settings: cross-validation, time-wise cross-validation, and cross-project prediction. Results: In our experiment, we compare CCUM against the state-of-the-art supervised and unsupervised JIT defect prediction models. Based on six open-source projects, our experimental results show that CCUM performs better than all the prior supervised and unsupervised models. Conclusions: The result suggests that future JIT defect prediction studies should use CCUM as a baseline model for comparison when a novel model is proposed.**

Keywords-Defect; prediction; changes; just-in-time; code churn; unsupervised models; supervised models

I. INTRODUCTION

Defect prediction techniques can play an important role in producing high quality software. However, most prior studies on defect prediction are conducted at module level (a module could be a package or a file) [3, 4, 7, 8, 10, 11, 12]. After applying module-level defect prediction, developers still have to spend considerable time to review or test those identified defect-prone modules. Therefore, recent years have seen an increasing research effort devoted to just-in-time (JIT) defect prediction. The JIT defect prediction aims to predict the probability of changes to be defect-inducing changes. A defect inducing-change is a software change that introduces one or several defects into a software system. Compared with prior studies, JIT defect prediction is a fine granularity prediction. According to Kamei et al. [5], JIT defect prediction allows developers to inspect the code changes for finding the defects when the design details are still fresh in their minds. Since code changes could be small code areas, it might provide effort savings over coarser grained predictions [5].

Kamei et al. [5] introduced the concept of effort-aware into JIT defect prediction. Based on linear regression, they built an effort-aware supervised JIT defect prediction model called EALR. They found that using only 20% effort it would take to inspect all changes, the EALR model could identify 35% defect-inducing changes. Recently, Yang et al. [9] leveraged individual change metrics to build unsupervised JIT defect prediction models. They reported that many unsupervised models performed similarly to or even better than the state-of-the-art JIT supervised defect prediction model EALR. However, in Yang et al.'s study, code churn (i.e. the change size of a code change), the basic change size metric, was neglected when building unsupervised defect prediction models.

In this study, we aim to investigate the effectiveness of code churn based unsupervised defect prediction model in effort-aware JIT defect prediction. To this end, we first use code churn to build an unsupervised JIT defect prediction model called CCUM. Then, we evaluate the prediction performance of CCUM against the state-of-the-art supervised and unsupervised models under the same three prediction settings as used in Yang et al.'s study: cross-validation, time-wise cross-validation, and cross-project prediction. Based on the same six open-source software projects, our experimental results show that CCUM performs better than all the state-of-the-art supervised and unsupervised models in effort-aware JIT defect prediction. Based on this finding, we strongly suggest that future JIT defect prediction studies should use CCUM as a baseline model for comparison. This will help develop better effective JIT defect prediction models.

The contributions of this paper are summarized as follows. First, we propose to use code churn to build an unsupervised model CCUM in the context effort-aware JIT defect prediction. CCUM is simple to apply in practice but has been neglected in prior studies. Second, we conduct an in-depth investigation on the prediction effectiveness of CCUM. In our experiment, we compare CCUM against the state-of-the-art supervised and unsupervised JIT defect prediction models including Kamei et al.'s EALR model [5], Yang et al.'s unsupervised models [9], Yang et al.'s TLEL model [29], and Fu et al.'s OneWay model [32]. Our work provides valuable data for both researchers and practitioners. For researchers, we provide a new baseline model that should be used in future JIT defect prediction studies. For practitioners, we provide a simple yet effective defect prediction model in effort-aware JIT defect prediction.

The rest of this paper is organized as follows. Section II introduces the background of defect prediction. Section III

978-1-5090-4039-1/17 $31.00 © 2017 IEEE
DOI 10.1109/ESEM.2017.8

describes the experimental setup in our study, including the research questions under study, the data sets used, performance indicators, and data analysis method. Section IV reports in detail the experimental results. Section V examines the threats to validity of our study. Section VI concludes the paper and outlines directions for future work.

II. BACKGROUND

In this section, we first introduce the background on supervised and unsupervised defect prediction models. Then, we describe related works in just-in-time (JIT) defect prediction. Finally, we introduce the application of code churn in defect prediction at the module level.

A. Supervised vs. unsupervised defect prediction

In the last decades, supervised models have ever been widely used in traditional defect prediction at module (e.g. file or class) level [1, 2, 5, 18, 20, 23, 25, 27, 28]. When building a supervised model, there is a need to use the defect data (e.g. the buggy or not buggy label information) for each module in a training set [15]. For practitioners, however, it may be not easy to collect such data, because not only it is a time-consuming work but also it is usually difficult to validate whether the collecting labels are correct [17].

Compared with supervised models, unsupervised models do not require the defect data when building the prediction models. In other words, there is no need to collect history label information. Therefore, unsupervised models incur a lower building cost and gain a wider application range [9]. As such, unsupervised models would be more desirable if they perform well. Recently, an increasing time and effort is devoted into applying unsupervised modeling techniques to build defect prediction models [3, 16, 18, 25].

Since supervised models leverage the prior knowledge on software metrics and defects in a training set but unsupervised models do not, there are good reasons to suppose that supervised models perform better in defect prediction. However, Yang et al. observed that simple unsupervised models could be better than supervised models [9]. A possible explanation for this phenomenon is that existing supervised models do not sufficiently make use of the prior knowledge on software metrics and defects.

B. Just-in-time defect prediction

The origin of JIT defect prediction can be traced back to Mockus and Weiss's work [20]. They used a number of change metrics to predict whether a change was a defect-inducing change. Compared with traditional defect predictions at module level, JIT defect prediction can narrow down the checking area and hence can shorten the working time to deal with defects. A more beneficial factor is that JIT can make it easier to find the code developer who is appropriate for dealing with the defects. In the last decades, there is an increasing interest in JIT defect predictions. Śliwerski et al. [21] examined the properties of defect-inducing changes in the Eclipse and Mozilla projects. They found that, the larger a change was, the more likely it would induce a defect. Eyolfson et al. [35] studied the correlation between commit bugginess and the time of day of the commit, the day of week of the commit, and the experience and commit frequency of the commit authors. Yin et al. [22] presented a comprehensive characteristic studies on incorrect bug-fixes from large operating system code bases.

Kamei et al. [5] applied effort-aware evaluation in JIT defect predictions. In their study, churn (i.e. the total number of lines modified) was used as a measure of the effort required to inspect a change. In particular, they used a linear regression method to build the effort-aware JIT defect prediction model (called the EALR model). In EALR, Y(x)/Effort(x) was acted as the dependent variable, 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. According to EALR, 35% of all defect-inducing changes could be detected when using only 20% of the total inspection effort. As such, Kamei et al. concluded that effort-aware JIT defect prediction was able to focus on the most risky changes and hence helped reduce the costs of developing quality software.

Yang et al. [9] performed an empirical study to investigate the predictive power of simple unsupervised models in effort-aware JIT defect prediction. In particular, contrary to the general expectation, Yang et al. [9] found that several simple unsupervised models performed better than the state-of-the-art supervised model EALR reported by Kamei et al. [5]. Later, Fu et al. [32] challenged Yang et al.'s study [9]. They found that not all unsupervised learners work better than supervised learners on all six data sets for different evaluation measures. They investigated how to pick the best models before deploying them to a project. Consequently, Fu et al. proposed OneWay, a supervised learner built on the idea of simple unsupervised learners. More recently, Yang et al. [29] applied a two-layer ensemble learning (TLEL) technique to JIT defect prediction. In the inner layer, bagging and decision tree were combined to build a random forest model. In the outer layer, stacking was used to ensemble many different random forest models. In their experiment, Yang et al. compared TLEL with three baseline models, i.e. Deeper [26], DNC [24], and MKEL [19]. The experimental results showed that on average TLEL could discover over 70% of the buggy changes by reviewing only 20% of the lines of code. Note that, all the above three studies used the same data sets from the following six projects shared by Kamei et al. [5]: Bugzilla, Columba, Eclipse JDT, Mozilla, Eclipse Platform, and PostgreSQL.

C. Code churn in defect prediction

In the last decade, code churn has been one of the representative process metrics to build defect prediction models at module level. In the literature, for a given module, the code churn is usually defined as the sum of added lines of code and deleted lines of code. Elbaum et al. [36] reported that code churn was closely associated with defects. Nagappan et al. [37, 38] used code churn metrics to predict system defect density. Shin et al. [39] evaluated code churn and complexity metrics as indicators of software vulnerabilities. Their results showed that code churn performed better in vulnerability prediction. Indeed, code churn has been widely used in module-level defect

prediction. Currently, many development environments such as Visual Studio [30] support the automated collection of code churn. However, to the best of our knowledge, code churn is neglected in JIT defect prediction. Although previous studies have demonstrated the usefulness of code churn in module-level defect prediction, it is unknown whether code churn is useful in JIT defect prediction.

III. EXPERIMENTAL SETUP

In this section, we first describe CCUM, a code churn based unsupervised model. Then, we introduce the state-of-the-art supervised and unsupervised models used as the baseline models in our study. Next, we describe the research questions and present the data sets used in our experiments. After that, we describe the performance indicators used to evaluate the effectiveness of defect prediction models. Finally, we report the data analysis method.

A. Code churn based unsupervised model

In this study, we leverage the code churn of a software change to build an unsupervised defect prediction model at change level. For each change, the corresponding code churn is the sum of lines of code added and deleted during the change. More specifically, we build the unsupervised model CCUM that sorts changes in descendant order according to the reciprocal of their corresponding raw metric value of code churn. This is consistent with Yang et al.'s study to build unsupervised models [9]. According to [9], the idea to build this kind of unsupervised models was inspired by Koru and Menzies et al.'s finding that smaller modules are proportionally more defect-prone and should be inspected first [19, 25]. In our study, we expect that changes with "smaller" change size tend to be more proportionally defect-prone. More formally, the unsupervised model CCUM is $R(c) = 1/Churn(c)$. Here, c represents a change, Churn(c) is the code churn of change c, and R is the predicted risk value. For a given system, all changes will be ranked in descendant order according to the predicted risk value R. In this content, changes with smaller code churn will be ranked higher.

B. Baseline models

In this study, on the one hand, we compare CCUM against the following state-of-the-art supervised JIT defect prediction models: EALR [5], TLEL [29], and OneWay [32]. EALR is a linear regression based effort-aware JIT defect prediction model. TLEL is a two-layer ensemble learning based JIT defect prediction model. OneWay is supervised JIT defect prediction model built on the idea of simple unsupervised learner prediction. On the other hand, we compare CCUM against the following state-of-the-art unsupervised JIT defect prediction models: LT and AGE [9]. The LT model ranks changes in ascendant order according to the lines of code in the associated files before changed, while the AGE model ranks changes in ascendant order according to the age of the last change. In [9], Yang et al.'s results showed that LT and Age were the best two unsupervised models in JIT defect prediction.

C. Research questions

We investigate the following four research questions to determine the practical value of the code churn model CCUM.

- *RQ1: (CCUM vs. EALR) How well does CCUM predict defect-inducing changes when compared with Kamei et al.'s supervised model EALR [5]?*
- *RQ2: (CCUM vs. TLEL) How well does CCUM predict defect-inducing changes when compared with the Yang et al.'s supervised model TLEL [29]?*
- *RQ3: (CCUM vs. OneWay) How well does CCUM predict defect-inducing changes when compared with Fu et al.'s supervised model OneWay [32]?*
- *RQ4: (CCUM vs. Other unsupervised models) How well does CCUM predict defecting changes when compared with Yang et al.'s unsupervised models [9]?*

The purposes of the above research questions are to compare CCUM with the state-of-the-art supervised and unsupervised models proposed in recent studies [5, 29, 9, 32]. The answers to these research questions would help determine how well the code churn model CCUM predicts defect-inducing changes in JIT defect prediction.

D. Data sets

In order to make a fair comparison with the state-of-the-art supervised and unsupervised models, we use the same six open-source software projects to conduct the investigation. These projects include Bugzilla (BUG), Columba (COL), Eclipse JDT (JDT), Eclipse Platform (PLA), Mozilla (MOZ), and PostgreSQL (POS). Table I summarizes the six data sets used in this study. The first column is the project name and the second column is the period for collecting the changes. The third and fourth columns are respectively the total number of defect-inducing changes and the total number of software changes. The last column is the percent of defect-inducing changes in each software project.

TABLE I. THE SIX DATA SETS USED IN THIS STUDY

Project	Period	#defect-inducing changes	#changes	%defect-inducing changes
BUG	1998/08/26~2006/12/16	1696	4620	36.71%
COL	2002/11/25~2006/07/27	1361	4455	30.55%
JDT	2001/05/02~2007/12/31	5089	35386	14.38%
MOZ	2000/01/01~2006/12/17	5149	98275	5.24%
PLA	2001/05/02~2007/12/31	9452	64250	14.71%
POS	1996/07/09~2010/05/15	5119	20431	25.06%

Table II lists the fourteen change metrics in the six data sets, including the dimension, the metric name, and the description. These metrics are grouped into the following five dimensions: diffusion, size, purpose, history, and experience. The diffusion dimension assumes that a highly distributed change is more likely to be a defect-inducing change. The size dimension assumes that a larger change has a higher likelihood of being a defect-inducing change. The purpose dimension assumes that a defect-fixing change is more likely to introduce a new defect. The history dimension

assumes 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. The experience dimension assumes that the experience of the developer of a change has a negative correlation to the likelihood of introducing a defect into the code by the change.

TABLE II. SUMMARIZATION OF CHANGE METRICS

Dimension	Metric	Description
Diffusion	NS	Number of touched subsystems
	ND	Number of touched directories
	NF	Number of touched files
	Entropy	Distribution across the touched files
Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code in a file before the change
Purpose	FIX	Whether or not the change is a defect fix
History	NDEV	Number of developers that changed the files
	AGE	The average time interval (in days) between the last and the current change over the files that are touched
	NUC	Number of unique last changes to the files
Experience	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 (in years)
	SEXP	Developer experience on a subsystem, i.e. the number of changes the developer made in the past to the subsystems

E. Performance indicators

The following performance indicators will be involved when we compare CCUM against the baseline models: ACC, P_{opt}, Precision, Recall, F1, and PofB20.

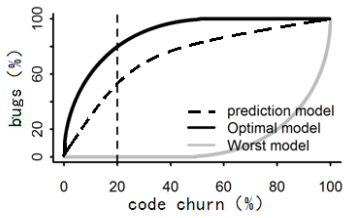

Figure 1. Performance indicator of P_{opt}

We will use ACC and P_{opt} when comparing CCUM against EALR (RQ1). In [5], ACC and P_{opt} were used to evaluate the effectiveness of EALR. 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 based on "code-churn-based" Alberg diagram as shown in Fig. 1. 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 bugs found in selected changes. To compute P_{opt}, two additional curves are included: the "optimal" model and the "worst" model. In the "optimal" model, changes are sorted in decreasing order according to their actual defect densities. In the "worst" model, changes are sorted in ascending order according to their actual bug densities.

According to [5], P_{opt} can be formally described as:

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

Here, Area (optimal) and Area (worst) is the area under the curves corresponding to the best model and the worst model, respectively.

We will use PofB20, precision, recall, and F1 when comparing CCUM against TLEL (RQ2). In [29], PofB20, precision, recall, and F1 were used to evaluate the effectiveness of TLEL. PofB20 is another name of ACC. F1 is a commonly- used indicator to evaluate classification performance. It combines precision and recall, and can be derived from a confusion matrix. If a change is correctly classified as "defect-inducing", it is a true positive (TP); if a change is misclassified as "defect-inducing", it is a false positive (FP). Similarly, there are false negatives (FN) and true negatives (TN). Precision is the ratio of the number of correctly predicted "defect-inducing" changes to the number of the changes predicted as "defect-inducing" (i.e. precision = TP / (TP+FP)). Recall is the ratio of the number of correctly predicted "defect-inducing" changes to the actual number of "defect-inducing" changes (i.e. recall = TP / (TP+FN). F1 is a harmonic mean of precision and recall: F1 = 2 × recall × precision / (recall + precision).

We will use ACC, P_{opt}, precision, recall, and F1 when comparing CCUM against OneWay (RQ3). The reason is that these indicators were used to evaluate OneWay in [32]. Based on the similar reason, we will use ACC and P_{opt} when comparing CCUM against LT/AGE (RQ4).

F. Data analysis method

As stated in Section III.C, we will investigate four research questions (RQs) to compare CCUM with the baseline models. In our study, the involved prediction settings include 10 times 10-fold cross-validation, time-wise cross-validation, and cross-project cross-validation.

- 10 times 10-fold cross-validation. This validation is performed within each project. At each time, the data set is divided to 10 parts of approximately equal size after randomized. Each part will be used as the testing set and all the others be training set. Therefore, for each model, 100 prediction values will be obtained.
- Time-wise cross-validation. This validation is also performed within each project. Compared with 10 times 10-fold cross-validation, there are two large differences. First, the changes in chronological order according to the commit date are grouped into the same part within the same month. Second, the testing set is not immediately following the training set, i.e. there is a gap between the two sets. According to [9], the part i and part i+1 will be combined a training set and the part i+4 and i+5 ($1 \le i \le n\_5$) will be combined as the test set.

If a project has changes of n months, for each model, n−5 prediction values will be obtained.

- Across-project cross-validation. This validation is performed across different projects. In this context, one project acts as a training set while another one acts as the testing set. Given n projects, for each model, n × (n−1) prediction values will be obtained. In this study, we use six projects as the subject projects. Therefore, each model will produce 30 prediction values.

In order to make a fair comparison, we use the same data sets, performance indicators, and prediction settings as used in previous studies. When comparing CCUM against EARL (RQ1), we will use 10 times 10-fold cross-validation, time-wise cross-validation, and across-project cross-validation. When comparing CCUM against TLEL (RQ2), we only use 10 times 10-fold cross-validation because time-wise and cross-project cross-validations are not used when evaluating TLEL in [29]. When comparing CCUM against OneWay (RQ3), we use the same time-wise cross-validation setting as used in [32]. When comparing CCUM against LT/AGE, we will use 10 times 10-fold cross-validation, time-wise cross-validation, and across-project cross-validation.

We use the Benjamini-Hochberg (BH) corrected p-values [33] from the Wilcoxon signed-rank test to examine whether there is a statistically significant difference in the prediction effectiveness between CCUM and the baseline models at the significance level of 0.05. Furthermore, we use the Cliff 's δ to examine whether the magnitude of the difference is practically important from the viewpoint of practical application. By convention, the magnitude of the difference is considered trivial ($|\delta| < 0.147$), small ($0.147 \le |\delta| < 0.33$), moderate ($0.33 \le |\delta| < 0.474$), or large ($|\delta| \ge 0.474$) [32, 34].

IV. EXPERIMENTAL RESULTS

In this section, we present the experimental results of CCUM in JIT defect prediction when compared with the state-of-the-art supervised and unsupervised models.

RQ1: CCUM vs. EALR

Table III summarizes for the CCUM and EALR models the ACC and P_{opt} scores over the six projects in 10 times 10-fold cross-validation. For each project, the third and fourth columns respectively report the median performance score and the standard deviation for EALR and CCUM. In particular, the entry with bold indicates that the corresponding model is the winner according to the median performance value. The fifth column reports the effect size measured by Cliff 's. The sixth column reports the BH-corrected p-value from the Wilcoxon signed-rank test. Note that, for EALR, the performance scores are slightly different from those performance scores presented in [9]. The reason is that the performance scores are indeed from 1 time 10-fold cross-validation rather than from 10 times 10-fold cross validation.

As can be seen, in terms of ACC and P_{opt}, CCUM performs significantly better than EALR, regardless of which project is taken into account. Furthermore, the effect sizes indicated by the Cliff 's δ are large over all the six projects. This is true for both ACC and P_{opt}. When looking at the mean median performance scores over the six projects, we can find that EALR has an ACC of 0.291 and a P_{opt} of 0.555. Compared to the random model (ACC = 0.2 and P_{opt} = 0.5), EALR only leads to a 46% improvement on ACC and 11% improvement. This is contrast to a 268% improvement on ACC and a 79% improvement on P_{opt} lead by CCUM. The core observation from Table III is that CCUM performs substantially better than EALR substantially in all data sets.

TABLE III. CCUM VS. EALR IN 10 TIMES 10-FOLD CROSS-VALIDATION

Indicator	Project	EALR	CCUM	Cliff's δ	P-value
ACC	BUG	0.415±0.110	**0.785±0.049**	1	<0.001
	COL	0.405±0.099	**0.812±0.080**	1	<0.001
	JDT	0.208±0.103	**0.745±0.088**	1	<0.001
	MOZ	0.148±0.025	**0.605±0.024**	1	<0.001
	PLA	0.300±0.072	**0.765±0.060**	1	<0.001
	POS	0.271±0.065	**0.705±0.068**	1	<0.001
	Mean	0.291±0.079	**0.736±0.061**	—	—
P_{opt}	BUG	0.717±0.061	**0.929±0.015**	1	<0.001
	COL	0.598±0.079	**0.937±0.024**	1	<0.001
	JDT	0.494±0.098	**0.885±0.033**	1	<0.001
	MOZ	0.456±0.02	**0.812±0.013**	1	<0.001
	PLA	0.55±0.025	**0.893±0.023**	1	<0.001
	POS	0.517±0.066	**0.902±0.021**	1	<0.001
	Mean	0.555±0.058	**0.893±0.022**	—	—

Table IV and Table V respectively summarize for the CCUM and EALR models the results of ACC and P_{opt} over the six projects in time-wise cross-validation and in across-project cross-validation. We can observe that CCUM exhibits an absolute advantage over EALR, regardless of whether ACC and P_{opt} is taken into account. According to p-value, the difference between CCUM and EALR is significant. According to Cliff 's , the magnitude of the difference between CCUM and EALR is practically important.

TABLE IV. CCUM VS. EALR IN TIME-WISE CROSS-VALIDATION

Indicator	Project	EALR	CCUM	Cliff's δ	P-value
ACC	BUG	0.286±0.218	**0.697±0.181**	0.835	<0.001
	COL	0.400±0.146	**0.720±0.129**	0.932	<0.001
	JDT	0.323±0.135	**0.637±0.109**	0.915	<0.001
	MOZ	0.180±0.130	**0.582±0.095**	0.941	<0.001
	PLA	0.305±0.158	**0.704±0.119**	0.949	<0.001
	POS	0.356±0.163	**0.615±0.166**	0.838	<0.001
	Mean	0.308±0.158	**0.659±0.133**	—	—
P_{opt}	BUG	0.596±0.199	**0.909±0.075**	0.917	<0.001
	COL	0.619±0.15	**0.907±0.065**	0.970	<0.001
	JDT	0.590±0.114	**0.837±0.052**	0.958	<0.001
	MOZ	0.498±0.112	**0.802±0.045**	0.959	<0.001
	PLA	0.583±0.111	**0.868±0.050**	0.984	<0.001
	POS	0.600±0.146	**0.866±0.079**	0.899	<0.001
	Mean	0.581±0.139	**0.865±0.061**	—	—

Combining the results from Table III, Table IV, and Table V, we conclude that CCUM is substantially better than ELAR according to ACC and P_{opt}.

TABLE V. CCUM VS. EALR IN ACROSS-PROJECT CROSS-VALIDATION

Train	Test	ACC		P_{opt}	
		EALR	*CCUM*	*EALR*	*CCUM*
BUG	COL	0.497	**0.836**	0.701	**0.948**
	JDT	0.281	**0.761**	0.592	**0.897**
	MOZ	0.219	**0.605**	0.563	**0.813**
	PLA	0.429	**0.774**	0.695	**0.899**
	POS	0.349	**0.726**	0.603	**0.908**
COL	BUG	0.336	**0.79**	0.633	**0.932**
	JDT	0.113	**0.761**	0.418	**0.897**
	MOZ	0.148	**0.605**	0.476	**0.813**
	PLA	0.239	**0.774**	0.506	**0.899**
	POS	0.167	**0.726**	0.426	**0.908**
JDT	BUG	0.376	**0.79**	0.674	**0.932**
	COL	0.378	**0.836**	0.564	**0.948**
	MOZ	0.168	**0.605**	0.502	**0.813**
	PLA	0.277	**0.774**	0.555	**0.899**
	POS	0.299	**0.726**	0.514	**0.908**
MOZ	BUG	0.35	**0.79**	0.619	**0.932**
	COL	0.247	**0.836**	0.482	**0.948**
	JDT	0.057	**0.761**	0.367	**0.897**
	PLA	0.173	**0.774**	0.414	**0.899**
	POS	0.135	**0.726**	0.382	**0.908**
PLA	BUG	0.367	**0.79**	0.685	**0.932**
	COL	0.363	**0.836**	0.564	**0.948**
	JDT	0.152	**0.761**	0.458	**0.897**
	MOZ	0.168	**0.605**	0.498	**0.813**
	POS	0.23	**0.726**	0.478	**0.908**
POS	BUG	0.351	**0.79**	0.629	**0.932**
	COL	0.439	**0.836**	0.617	**0.948**
	JDT	0.159	**0.761**	0.459	**0.897**
	MOZ	0.159	**0.605**	0.497	**0.813**
	PLA	0.261	**0.774**	0.544	0.899
Mean		0.263	**0.749**	0.537	**0.900**
P-value		<0.001	—	<0.001	—
Cliff's δ		1	—	1	—

RQ2: CCUM vs. TLEL

Yang et al.'s model [29] TLEL is a recently proposed supervised JIT defect prediction model. TLEL is much more complex than EALR and hence is more difficult to apply in practice. In [29], Yang et al. performed 100 times 10-fold cross-validation to evaluate the effectiveness of TLEL and reported the average PofB20, precision, recall, and F1 over the six projects. In their experiment, they did not evaluate the performance TLEL in time-wise cross-validation and across-project cross-validation. In contrast, CCUM is a very simple unsupervised JIT defect prediction model. It is very interesting to know how well CCUM performs when compared with the complex supervised model TLEL.

Table VI summarizes for the CCUM and TLEL models the average PofB20, precision, recall, and F1 over the six projects. For the CCUM model, we compute precision, recall, and F1 at the effort = 20% point. In this context, for CCUM, PofB20 (i.e. ACC) is equal to recall. As can be seen, according to F1, CCUM performs worse than TLEL (the p-value is not significant). However, according to PofB20, there is no statically significant difference between CCUM and TLEL. Therefore, we conclude that, in effort-aware JIT defect prediction, CCUM performs similarly to TLEL.

TABLE VI. CCUM VS TLEL IN 10-FOLD CROSS-VALIDATION

Project	TLEL				CCUM			
	PofB20	*Precision*	*Recall*	*F1*	*PofB20*	*Precision*	*Recall*	*F1*
BUG	61.67	**0.6239**	0.7592	**0.685**	**78.47**	0.3302	**0.7847**	0.464
COL	58.85	**0.5122**	0.7433	**0.606**	**81.25**	0.2704	**0.8125**	0.405
JDT	72.55	**0.2934**	0.7348	**0.419**	**74.48**	0.1192	**0.7448**	0.205
MOZ	**82.40**	**0.1579**	**0.7775**	**0.262**	60.51	0.1251	0.6051	0.215
PLA	**77.08**	**0.3142**	**0.7748**	**0.447**	76.46	0.0354	0.7646	0.067
POS	70.64	**0.4986**	**0.7697**	**0.605**	**71.63**	0.1990	0.7163	0.311
Mean	70.53	**0.4000**	**0.7599**	**0.504**	**73.80**	0.1799	0.7380	0.278
P-value	0.438	0.031	0.843	0.313	—	—	—	—
Cliff's δ	0.222	–0.722	–0.056	–0.722	—	—	—	—

RQ3: CCUM vs. OneWay

In [32], Fu et al. evaluated OneWay in time-wise cross-validation and reported the median ACC, P_{opt}, precision, recall, and F1 over the six projects. In particular, they computed precision, recall, and F1 at the effort = 20% point.

Table VII summarizes for the CCUM and OneWay models the average ACC, P_{opt}, precision, and F1 over the six projects. Since we use the effort = 20% point to compute precision, recall, and F1, recall is equal to ACC. For the simplicity of presentation, we do not present the results for recall. As can be seen, on the six projects, CCUM has a 57% = (0.659 0.42) / 0.42 ACC improvement and a 25% = (0.865 0.69) / 0.69 P_{opt} improvement over OneWay. In other words, when using the 20% effort to inspect all the changes, the CCUM model can detect 57% more defect-inducing changes than the OneWay model. According to the p-value, CCUM performs significantly better than OneWay in terms of ACC and P_{opt}. Furthermore, according to Cliff 's , CCUM is substantially better than OneWay in terms of ACC and P_{opt}. In terms of F1, OneWay performs a little better than CCUM (0.23 vs. 0.151). However, there is no statistically significant difference in F1 between CCUM and OneWay. Based on the above results, we conclude that, in effort-aware JIT defect prediction, CCUM performs better than OneWay.

TABLE VII. CCUM VS. ONEWAY IN TIME-WISE CROSS-VALIDATION

Project	OneWay				CCUM			
	ACC	*P_{opt}*	*F1*	*Precision*	*ACC*	*P_{opt}*	*F1*	*Precision*
BUG	0.36	0.65	**0.35**	**0.39**	**0.697**	**0.909**	0.264	0.238
COL	0.56	0.76	0.17	0.11	**0.720**	**0.907**	**0.224**	**0.179**
JDT	0.42	0.7	0.08	0.04	**0.637**	**0.837**	**0.119**	**0.077**
MOZ	0.33	0.62	**0.18**	**0.12**	**0.582**	**0.802**	0.037	0.021
PLA	0.41	0.69	**0.32**	**0.25**	**0.704**	**0.868**	0.117	0.074
POS	0.44	0.74	**0.29**	**0.23**	**0.615**	**0.866**	0.144	0.111
Mean	0.42	0.69	**0.23**	**0.19**	**0.659**	**0.865**	0.151	0.116
P-value	0.031	0.031	0.156	0.156	—	—	—	—
Cliff's δ	1	1	–0.5	–0.389	—	—	—	—

When considering RQ1, RQ2 and RQ3 together, we can conclude that, according to effort-aware performance indicators, CCUM performs better than the state-of-art supervised JIT defect prediction models. This indicates that CCUM is a good choice for practitioners if they hope to find defect-inducing changes under limited inspection effort.

RQ4: CCUM vs. Other unsupervised models

In [9], Yang et al. proposed a number of unsupervised models, including NS, ND, Ent (Entropy), LT, FIX, NDE (NDEV), AGE, NUC, EXP, REX (REXP), and SEX(SEXP). Yang et al. evaluated these models in 10-fold cross-validation, time-wise cross-validation, and across-project cross validation. Of these unsupervised models, according to their results, the LT and AGE model were the best two models. In the following, we will compare CCUM against these models.

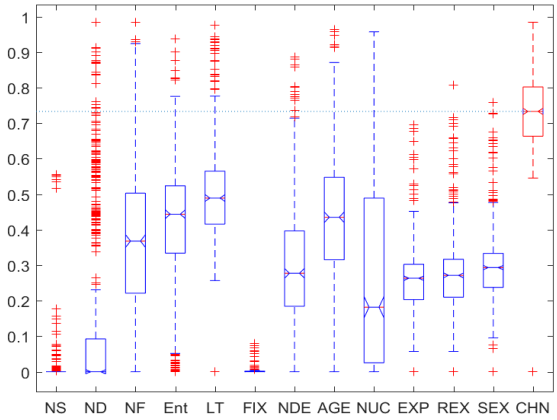

Figure 2. ACC of six data-sets in 10 times 10-fold cross-validation

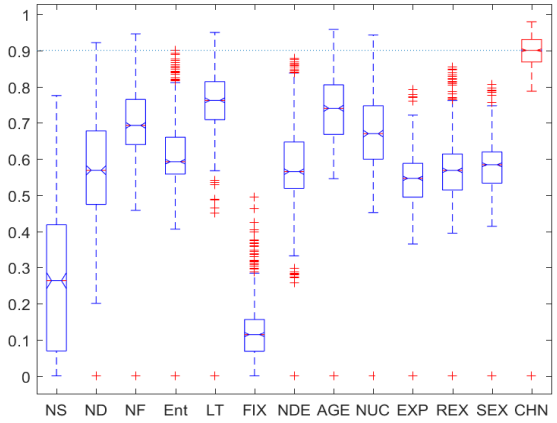

Figure 3. P_{opt} of six datasets in 10 times 10-fold cross-validation

As shown in Fig 2., the red box is high above all the other boxes, i.e. CCUM is the best one among these unsupervised models in terms of ACC. As shown in Fig. 3, the box of CCUM still occupies the highest position. Therefore, with the visualization results, we conclude that CCUM performs better than Yang et al.'s unsupervised models.

Table VIII summarizes the median P_{opt} and ACC for LT, AGE, and CCUM in 10 times 10-fold cross-validation. Note that, here, Cliff(LT) is the cliff 's between CCUM and LT, while Cliff(AGE) is the cliff 's between CCUM and AGE. As can be seen, CCUM performs significantly better than the other two unsupervised models, regardless of whether P_{opt} or ACC is taken into account.

Fig. 4 shows the box-plots for ACC over the six data sets in time-wise cross-validation, while Fig. 5 shows the box-plots for P_{opt} in time-wise cross-validation. From these two figures, we can see that the red box (i.e. CCUM) is much higher above than any other boxes. Except CCUM, LT and AGE are the next best two models, consistent with Yang et al. 's experimental results [9]. We can observe that no box can reach the media line of the box of the CCUM model.

TABLE VIII. CCUM VS LT/AGE IN 10 TIMES 10-FOLD CROSS-VALIDATION

		LT	Cliff's δ (LT)	AGE	Cliff's δ (AGE)	CCUM
ACC	BUG	0.437±0.164	1.000	0.427±0.126	0.999	**0.785±0.049**
	COL	0.623±0.145	0.718	0.675±0.121	0.651	**0.812±0.08**
	JDT	0.572±0.114	0.746	0.474±0.138	0.867	**0.745±0.088**
	MOZ	0.365±0.035	0.983	0.232±0.058	0.994	**0.605±0.024**
	PLA	0.489±0.089	1.000	0.426±0.11	1.000	**0.765±0.06**
	POS	0.504±0.098	0.839	0.427±0.111	0.943	**0.705±0.068**
	Mean	0.498±0.108	0.881	0.443±0.111	0.909	**0.736±0.061**
	p-value	<0.001	—	<0.001	—	—
P_{opt}	BUG	0.732±0.084	1.000	0.752±0.063	1.000	**0.929±0.015**
	COL	0.827±0.069	0.876	0.846±0.053	0.876	**0.937±0.024**
	JDT	0.793±0.060	0.760	0.749±0.068	0.907	**0.885±0.033**
	MOZ	0.659±0.024	0.990	0.621±0.029	1.000	**0.812±0.013**
	PLA	0.756±0.052	1.000	0.725±0.059	1.000	**0.893±0.023**
	POS	0.796±0.050	0.928	0.751±0.066	0.983	**0.902±0.021**
Mean		0.760±0.056	0.926	0.741±0.056	0.961	**0.893±0.022**
P-value		<0.001	—	<0.001	—	—

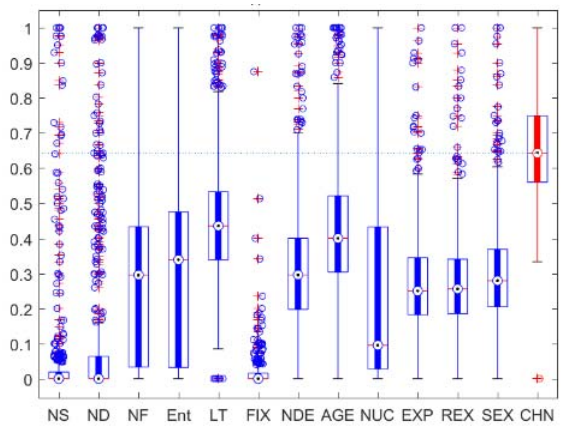

Figure 4. ACC of six data sets in time-wise cross-validation

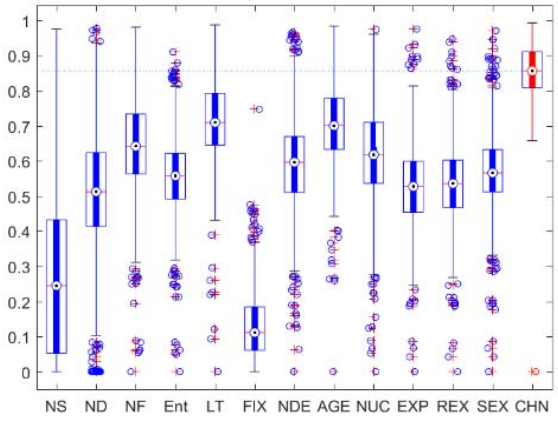

Figure 5. P_{opt} of six datasets in time-wise cross-validation

Table IX summarizes the median ACC and P_{opt} for LT, AGE, and CCUM in time-wise cross-validation. In Yang et al. 's study [9], LT and AGE are the best simple unsupervised models. Therefore, we further compare CCUM against LT and AGE. As can be seen, the p-value indicates that CCUM performs significantly better than LT and AGE. In addition, the Cliff 's indicates that CCUM is substantially better than LT and AGE in terms of ACC and P_{opt}.

Table X summarizes the median ACC and P_{opt} for LT, AGE, and CCUM in across-project cross-validation. As shown in Table X, CCUM continues to be the best model. The difference between CCUM and LT/AGE is significant and the effect size measured by Cliff 's is large.

TABLE IX. CCUM VS LT/AGE IN TIME-WISE CROSS-VALIDATION

		LT	Cliff's δ (LT)	AGE	Cliff's δ (AGE)	CCUM
ACC	BUG	0.449±0.224	0.570	0.375±0.275	0.603	**0.697±0.181**
	COL	0.440±0.215	0.654	0.568±0.18	0.532	**0.720±0.129**
	JDT	0.452±0.131	0.754	0.408±0.166	0.725	**0.637±0.109**
	MOZ	0.363±0.113	0.855	0.28±0.167	0.864	**0.582±0.095**
	PLA	0.432±0.168	0.900	0.429±0.178	0.938	**0.704±0.119**
	POS	0.432±0.221	0.486	0.426±0.183	0.587	**0.615±0.166**
Mean		0.428±0.179	0.703	0.414±0.192	0.708	**0.659±0.133**
P-value		<0.001	—	<0.001	—	—
P_{opt}	BUG	0.721±0.165	0.748	0.661±0.164	0.723	**0.909±0.075**
	COL	0.732±0.146	0.657	0.786±0.104	0.621	**0.907±0.065**
	JDT	0.709±0.084	0.771	0.685±0.097	0.730	**0.837±0.052**
	MOZ	0.651±0.07	0.868	0.638±0.107	0.879	**0.802±0.045**
	PLA	0.717±0.075	0.919	0.709±0.083	0.945	**0.868±0.050**
	POS	0.742±0.134	0.564	0.731±0.101	0.657	**0.866±0.079**
Mean		0.712±0.112	0.755	0.702±0.109	0.759	**0.865±0.061**
P-value		<0.001	—	<0.001	—	—

Combining the results from Table VIII, Table IX, and Table X, we conclude that CCUM is substantially better than LT/AGE according to ACC and P_{opt}. In other words, CCUM is substantially better than the unsupervised models in [9].

V. THREATS TO VALIDITY

We compare CCUM with the state-of-the-art JIT defect prediction models. There are three possible threats to the validity of our study.

- Data sets. We only used six projects to compare CCUM against the baseline models. It is possible that our conclusions could not be generalized to other projects. This threat needs to be reduced by using more projects to conduct the experiment in the future work.
- Performance indicators. When evaluating the cost effectiveness of a model, we used performance indicators such as PofB20 at specific cut points. It is possible that the cut points have an influence on our conclusions. Therefore, in the future work, there is a need to use more comprehensive performance indicators to compare CCUM against the baseline models.
- Prediction settings. In time-wise cross-validation, the gap between the training and testing data sets was set to two consecutive months. The gap may have an unknown influence on our conclusions.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we perform an empirical study to investigate the predictive power of the unsupervised model of code churn (CCUM) in effort-aware JIT defect prediction. Our results from six industrial-size systems show that CCUM performs better than the state-of-the-art supervised models (including EALR reported by Kamei et al. [5] , TLEL reported by Yang et al. [29] , and OneWay reported by Fu et al. [32]) and better than the simple unsupervised models reported by Yang et al. [9]. Although CCUM is very effective in effort JIT defect prediction, it is neglected in existing studies. Due the low building cost and high prediction effectiveness, it is a good choice for practitioners to apply CCUM in practice.

TABLE X. CCUM VS LT/AGE IN ACROSS-PROJECT CROSS-VALIDATION

Train	Test	ACC			P_{opt}		
		LT	*AGE*	*CCUM*	*LT*	*AGE*	*CCUM*
BUG	COL	0.641	0.702	**0.836**	0.858	0.868	**0.948**
	JDT	0.582	0.490	**0.761**	0.815	0.769	**0.897**
	MOZ	0.367	0.240	**0.605**	0.660	0.622	**0.813**
	PLA	0.494	0.427	**0.774**	0.770	0.740	**0.899**
	POS	0.533	0.432	**0.726**	0.810	0.762	**0.908**
COL	BUG	0.435	0.432	**0.790**	0.726	0.758	**0.932**
	JDT	0.582	0.490	**0.761**	0.815	0.769	**0.897**
	MOZ	0.367	0.240	**0.605**	0.660	0.622	**0.813**
	PLA	0.494	0.427	**0.774**	0.770	0.740	**0.899**
	POS	0.533	0.432	**0.726**	0.810	0.762	**0.908**
JDT	BUG	0.435	0.432	**0.790**	0.726	0.758	**0.932**
	COL	0.641	0.702	**0.836**	0.858	0.868	**0.948**
	MOZ	0.367	0.240	**0.605**	0.660	0.622	**0.813**
	PLA	0.494	0.427	**0.774**	0.770	0.740	**0.899**
	POS	0.533	0.432	**0.726**	0.810	0.762	**0.908**
MOZ	BUG	0.435	0.432	**0.790**	0.726	0.758	**0.932**
	COL	0.641	0.702	**0.836**	0.858	0.868	**0.948**
	JDT	0.582	0.490	**0.761**	0.815	0.769	**0.897**
	PLA	0.494	0.427	**0.774**	0.770	0.740	**0.899**
	POS	0.533	0.432	**0.726**	0.810	0.762	**0.908**
PLA	BUG	0.435	0.432	**0.790**	0.726	0.758	**0.932**
	COL	0.641	0.702	**0.836**	0.858	0.868	**0.948**
	JDT	0.582	0.490	**0.761**	0.815	0.769	**0.897**
	MOZ	0.367	0.240	**0.605**	0.660	0.622	**0.813**
	POS	0.533	0.432	**0.726**	0.810	0.762	**0.908**
POS	BUG	0.435	0.432	**0.790**	0.726	0.758	**0.932**
	COL	0.641	0.702	**0.836**	0.858	0.868	**0.948**
	JDT	0.582	0.490	**0.761**	0.815	0.769	**0.897**
	MOZ	0.367	0.240	**0.605**	0.660	0.622	**0.813**
	PLA	0.494	0.427	**0.774**	0.770	0.740	**0.899**
Mean		0.509	0.454	**0.749**	0.773	0.753	**0.900**
P-value		<0.001	<0.001	—	<0.001	<0.001	—
Cliff's δ		0.944	0.944	—	0.889	0.944	—

Our study only investigates the effectiveness of code churn in open-source software systems. It is unclear whether our conclusion is applicable to closed-source software. Therefore, in the future, an interesting work is to use closed-source software to investigate the usefulness of code churn in effort-aware JIT defect prediction.

ACKNOWLEDGMENT

This work is partially supported by the National Key Basic Research and Development Program of China (2014CB340702), the National Natural Science Foundation of China (61432001, 61472178, 61702256), and the National Natural Science Foundation of Jiangsu Province (BK20130014 and BK20170652).

REFERENCES

[1] S. Kim, E. Whitehead, Y. Zhang. Classifying software changes: clean or buggy? IEEE Transactions on Software Engineering, 2008, 34(2): 181-196.

[2] A. E. Hassan. Predicting faults using the complexity of code changes. ICSE 2009: 78-88.

[3] A. Meneely, L. Williams, W. Snipes, J. Osborne. Predicting failures with developer networks and social network analysis. FSE 2008: 13-23.

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

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

[6] T. Fukushima, Y. Kamei, S. Mcintosh, K. Yamashita, N. Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. MSR 2014: 172-181.

[7] N. Bettenburg, M. Nagappan, A. E. Hassan. Think locally, act globally: improving defect and effort prediction models. MSR 2012: 60-69.

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

[9] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, H. Leung. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. FSE 2016: 157-168.

[10] N. Nagappan, T. Ball, A. Zeller, Mining metrics to predict component failures. ICSE 2006: 452-461.

[11] L. Cheung, R. Roshandel, N. Medvidovic, L. Golubchik. Early prediction of software component reliability. ICSE 2008: 111-120.

[12] S. Kim, T. Zimmermann, E. J. Whitehead Jr., A. Zeller. Predicting faults from cached history. ICSE 2007:489–498.

[13] E. Giger, M. D'Ambros, M. Pinzger, H.C. Gall. Method-level bug prediction. ESEM 2012: 171-180.

[14] S. Kim, H. Zhang, R. Wu, L. Gong, Dealing with noise in defect prediction. ICSE 2011: 481-490.

[15] H. Valpola. Bayesian ensemble learning for nonlinear factor analysis. cta Polytechnic Scandinavia. Mathematics and Computing Series No.108, pp. 26-27

[16] S. Zhong,T.Khoshgoftaar, N.Seliya. Unsupervised learning for expert-based software quality estimation. HASE 2004: 149-155.

[17] K. Herzig, S. Just, A. Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. ICSE 2013: 392-401.

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

[19] T.Wang, Z. Zhang, X. Jing, L. Zhang. Multiple kernel ensemble learning for software defect prediction. Automated Software Engineering, 2015, 23(4): 569-590.

[20] Mockus, A. and Weiss, D.M. Predicting risk of software changes[J]. Bell Labs Technical Journal, 2000, 5(2): 169-180.

[21] J. Liwerski, T. Zimmermann, A. Zeller. 2005. When do changes induce fixes. MSR 2005, 30(4): 1-5.

[22] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, L. Bairavasundaram. How do fixes become bugs. FSE 2011: 26-36.

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

[24] S. Wang, X. Yao, Using Class Imbalance Learning for Software Defect Prediction[J]. IEEE Transactions on Reliability, 2013, 62(2): 434-443.

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

[26] X. Yang, D. Lo, X. Xia, Y. Zhang, J. Sun. Deep learning for just-in-time defect prediction. QRS 2015:17–26.

[27] F. Rahman, D. Posnett, P. Devanbu. Recalling the "imprecision" of cross-project defect prediction. FSE 2012.

[28] F. Rahman, P. Devanbu. How, and why, process metrics are better. ICSE 2013: 432-441.

[29] X. Yang, D. Lo, X. Xia, J. Sun. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. Information and Software Technology, 2017, accepted.

[30] Visual Studio, Microsoft. https://www.visualstudio.com/zh-hans/

[31] M. Tan, L. Tan, S. Dara, C. Mayeux. Online defect prediction for imbalanced data. ICSE 2015:99-108

[32] W. Fu, T. Menzies. Revisiting unsupervised learning for defect prediction. arXiv:1703.00132 [cs.SE], 2017.

[33] Y. Benjamini, Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. Journal of the royal statistical society. 1995, 57(1): 289-300

[34] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, L. Devine. Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices. SAIR 2006.

[35] J. Eyolfson, T. Lin, P. Lam. Do time of day and developer experience affect commit bugginess. MSR 2011: 153-162.

[36] J.C. Munson, S.G. Elbaum. Code churn: a measure for estimating the impact of code change. ICSM 1998: 24-31.

[37] N. Nagappan, T. Ball. Use of relative code churn measures to predict system defect density. ICSE 2005: 284-292.

[38] N. Nagappan, T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. ESEM 2007: 364-373.

[39] Y. Shin, A. Meneely, L. Williams, J.A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Transactions on Software Engineering, 2011, 37(6): 772-787.