Cross-Sub-Project Just-in-Time Defect Prediction on Multi-Repo Projects

Yeongjun Cho, Jung-Hyun Kwon, In-Young Ko
School of Computing
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
{yj_cho, junghyun.kwon, iko}@kaist.ac.kr

***Abstract*—Just-in-time (JIT) defect prediction, which predicts defect-inducing code changes, can provide faster and more precise feedback to developers than traditional module-level defect prediction methods. We find that large-scale projects such as Google Android and Apache Maven divide their projects into multiple sub-projects, in which relevant source code is managed separately in different repositories. Although sub-projects tend to suffer from a lack of the historical data required to build a defect prediction model, the feasibility of applying cross-sub-project JIT defect prediction has not yet been studied. A cross-sub-project model to predict bug-inducing commits in the target sub-project could be built with data from all other sub-projects within the project of the target sub-project, or data from the sub-projects of other projects, as traditional project-level JIT defect prediction methods. Alternatively, we can rank sub-projects and select high-ranked sub-projects within the project to build a filtered-within-project model. In this work, we define a sub-project similarity measure based on the number of developers who have contributed to both sub-projects to rank sub-projects. We extract the commit data from 232 sub-projects across five different projects and evaluate the cost effectiveness of various cross-sub-project JIT defect prediction models. Based on the results of the experiments, we conclude that 1) cross-sub-project JIT defect prediction generally has better cost effectiveness than within-sub-project JIT defect prediction, especially when the sub-projects from the same project are used as training data; 2) in filtered-within-project JIT defect-prediction models, the developer similarity-based ranking can achieve higher cost effectiveness than the other ranking methods; and 3) although a developer similarity-based filtered-within-project model achieves lower cost effectiveness than a within-project model in general, we find that there is room for further improvement to the filtered-within-project model that may outperform the within-project model.**

***Index Terms*—Just-in-time prediction, defect prediction, sub-project**

I. Introduction

The quality assurance (QA) process on large-scale software usually requires a large amount of computing and human resources and time, which may not be affordable for organizations with limited testing resources. Because of the resource limits, an organization could not investigate all modules—files or packages—in a project but still needs to prioritize the inspection of the modules. If they fail to prioritize modules which have any defects inside and assign limited testing resources to others, it will result in a higher chance of defects in the released software. Defect prediction helps developers identify modules that are likely to have defects and provide a list of the modules that need to be treated first to efficiently assign limited resources [1]. To predict the likelihood of defects in each module, most of the defect prediction techniques provide a prediction model that is built based on various metrics, such as complexity [2] and change-history measures [3].

Although a defect prediction technique is helpful for developers to narrow down the modules to inspect, module inspection usually requires many resources because it is difficult for developers to locate a defect inside a suspicious module. To overcome the drawback of the file-level defect prediction technique, recent studies introduced just-in-time (JIT) defect prediction models that output a prediction of whether a *code change* induces defects, rather than a *file*. Because code change level is more fine-grained than the file level, predicting defect-inducing code changes is known to be effective to provide the precise feedback to developers. Moreover, JIT defect prediction technique can provide faster feedback as soon as a change is made into the source code repository [4]–[6].

In general, building a defect prediction model usually requires a large amount of historical data from a project; therefore, it is difficult to build a model for projects that have just started or for legacy systems in which past change history data are not available. A cross-project defect prediction model can be built by utilizing the history data of other projects to predict defects in a project that lacks data. Cross-project defect prediction techniques have been actively studied [7], [8] and existing studies state that when building a cross-project defect prediction model, choosing a set of training data that are closely related to the target project is essential to ensure the performance of defect prediction [8].

We found that many large-scale projects, such as Apache Maven, Google Android, and Samsung Tizen, divide their projects into multiple repositories, called sub-projects. Each sub-project contains files that are relevant to the sub-project's main concerns—like the project core, independent artifacts, and plug-ins—and those files are managed in a separate source code repository.

Each sub-project generally has a lesser number of commits within its repository compared to other monolithic repositories, where all files and commits are managed within a single repository. For instance, Google Android in 2012 had 275

sub-projects and 183 sub-projects that contain less than 100 commits [9]. In addition, there might be changes in the design and architecture of the project, which consequently deprecates some sub-projects and introduces new sub-projects to the project. In the case of Google Android, there were 275 sub-projects in 2012 [9] and the number of sub-projects grew by over one thousand at 2018[1].

Therefore, applying JIT defect prediction to sub-projects could be problematic because it is well known that building a prediction model based on a small amount of data may increase the risk of overfitting and make the prediction less robust to outliers [10]. This motivated us to investigate the feasibility of applying cross-sub-project JIT defect prediction to multi-repo projects.

As Kamei at el. [11] studied at the project level, it might be enough to use all available sub-projects to build a cross-sub-project JIT defect-prediction model without selecting only similar sub-projects to build a model. However, as the sub-project level change history is more fine-grained than that of the project level, we may achieve higher cost effectiveness by filtering out irrelevant data based on fine-grained information about the data. For instance, we find that developers of a target sub-project usually make contributions to multiple sub-projects rather than to a single sub-project of the target project. This inspired us to develop a new similarity metric that measures the similarity between two sub-projects based on the number of authors (developers) who made commits to both sub-projects. Because different developers usually have different defect patterns [12], we expect that using the JIT defect prediction model built from sub-project repositories whose contributors are similar to those of the target project could show better prediction performance than a prediction model that is built from all available repositories.

In this paper, we study the ways of transferring the JIT defect-prediction models to the models that are appropriate for multi-repo projects in terms of cost effectiveness. We establish two research questions (RQs) to check the effectiveness of using the cross-sub-project JIT defect prediction method:

RQ 1 Is the cross-sub-project model trained by sub-projects from the same project more cost effective than that trained by sub-projects from other projects?

To build a cross-sub-project JIT defect prediction model, training data from other sub-projects are required. As training data are among the most important factors to improve a model's cost effectiveness, choosing proper sources of training data is the first concern to build cross-sub-project models. Before selecting sub-projects based on the similarity against the target sub-project, we want to check the cost effectiveness of the models built with sub-projects from the same projects and those built with sub-projects from other projects. If the models built with the sub-projects from the same project perform better than the other models, an organization may not need to spend time collecting commit data from other projects.

[1]https://android.googlesource.com

RQ 2 Which sub-project ranking method performs best in filtered-within-project JIT defect prediction?

In this research question, we build a filtered-within-project JIT defect prediction model that filters low-ranked sub-projects within the project, which are ranked by a score calculated by a rank scoring method. We use four different sub-project ranking algorithms: developer similarity-based ranking by us, domain-agnostic similarity-based ranking by Kamei et al. [11], size-based ranking, and random ranking. We then compare the cost effectiveness of four similarity algorithms with different the numbers of sub-projects selected to find the best-performing ranking algorithm for cross-sub-project JIT defect prediction.

By answering the research questions above, we conclude that 1) a JIT defect prediction model built with the training data from the same project is preferred over a model built with the training data from other projects; 2) the amount of training data is not the only factor that affects the cost effectiveness of cross-sub-project models; 3) a developer similarity that counts the number of developers in both sub-projects is the most preferred way of filtering out irrelevant sub-projects in building filtered-within-project models; and 4) although filtered-within-project models have lower cost effectiveness than within-project models, we notice that there is room for further improvement of the cost effectiveness of filtered-within-project models.

The main contributions of this paper are 1) proposing a sub-project-level defect prediction that can achieve higher cost effectiveness than the traditional project-level within-project method with developer similarity-based filtered-within-project models; and 2) evaluating the cost effectiveness of various cross-sub-project JIT defect prediction models on 232 sub-projects to check its feasibility.

The rest of the paper is organized as follows: Section II describes the experiment settings and Section III presents the design and result of experiments. Section IV provides related works and Section V reports threats to validity. Finally, Section VI conclude this study.

II. Experiment Setting

A. Studied Projects and Sub-projects

We select five open-source projects that are divided into multiple sub-projects. We try to choose projects with different characteristics in terms of programming language, domain, and distribution of the number of commits.

Table I shows the statistics of five projects. For each project, we count the number of sub-projects used in the experiments and the distribution of commit counts in each sub-project. We excluded sub-projects with less than 50 commits because we regard such sub-projects have not enough data to evaluate. During the experiment, we split commits from each sub-project into ten slices for a cross-validation. If a slice only contains clean commits without bug-inducing commits, we cannot evaluate the performance measure for that slice. If there are too many discarded slices, then performance measures with extreme value may occur due to the lack of valid evaluation

results. If a slice has five commits, the probability that the slice has at least one buggy commit could be calculated by $((1 - \text{avg. defect ratio})^5)$. Because the average defect ratio across the five projects we collected is 0.14, this probability is about 0.52. Thus, we can expect that near half of slices will give valid evaluation results from sub-projects with more than 50 commits. For Tizen and Android projects, we use the subset of sub-project within those project because of their large size. For the Tizen project, we use the sub-projects with the prefix of `platform/core/api`. For the Android project, sub-projects with the prefix of `platform/package/apps` are used. We also calculate the ratio of commits with any defect by using the SZZ algorithm [13], which is explained in Section II-C, and calculate the average number of sub-projects contributed per developer, which indicates the feasibility of using information about the developer who contributed in multiple sub-projects within a project to calculate the developer similarity between sub-projects.

B. Change Metric

We used 14 change metrics that are widely adapted in the JIT defect prediction field [4], [14]. Table II shows descriptions of these change metrics. In metric description, the term *subsystem* represents directories that are directly accessible from the root directory.

We apply some modification to those metrics because our study is conducted on the sub-project level, not a project level where those metrics are defined. Originally, developer experience-related metrics are defined under the scope of the project, but we changed this to the scope of the sub-project.

To prevent the multi-collinearity problem in prediction models, we exclude metric values that are correlated with any other metrics. We calculate the Pearson correlation coefficient between each pair of 14 metrics across 344,005 commits. Then, we regard pairs with Pearson correlation coefficient values higher than 0.8 as being correlated and exclude one of the metrics in the correlated pair. As a result, we excluded `ND`, `NF`, `LT`, `NDEV`, `NUC`, and `SEXP` and eight types of metrics are used throughout our experiments. These metrics are listed in bold text in Table II.

C. Labeling

To build and evaluate a prediction model, buggy or clean labels must be specified for each change. Instead of manual labeling, which will cost a lot of time, we use the SZZ algorithm [13] to label bug inducing commits automatically. SZZ algorithm firstly finds bug-fixing changes by inspecting log messages from each change. Then, the algorithm backtracks bug-inducing changes from bug-fixing changes by looking at change history of files which are modified by bug-fixing changes.

D. Pre-processing

1) Sampling Training Data: The number of buggy changes from a source code repository is less than that from a clean one, as shown in the *Defect Ratio* column in TableI. This could be a serious problem, because imbalanced training data could lead to biased prediction results. To deal with this problem, we apply an under-sampling method to the training data. This sampling method randomly removes instances with the majority label until the numbers of instances with the buggy and clean labels are the same.

2) Log Transformation: When investigating the extracted metric values, which are all non-negative, we notice that most of them are highly skewed. To make the distribution of metric values similar to a normal distribution, we apply a logarithm transformation ($\log_2(x+1)$) to all metric values.

E. Prediction Model

There are more than 30 classification model learners used in cross-project defect prediction researches [7]. Because different model learners work better for different datasets, we choose three popular classification model learners that were used by other defect prediction papers. Random forest (RF) [11], logistic regression (LR) [4] and naive Bayes (NB) [15] are selected for our experiments.

Those model learners build a classification model with training instances. An instance consists of the change metric values of a commit and a label whether the commit has any defect. When the change metric values from a new commit are given to a classification model, the model returns a probability that the commit has any defect, so called defect-proneness, or a binary classification whether this commit is buggy or clean. New commits which needs to be inspected for quality assurance can be prioritized by their buggy probability to make suspicious commits checked first. Since we use cost effectiveness as our performance measure, as explained in the next sub-section, we use cost-aware defect-prediction models to consider the cost of investigating a commit. Whereas a general defect-prediction model returns defect-proneness, a cost-aware defect-prediction model returns defect-proneness divided by the number of lines of code [16].

F. Performance Measures

Many previous cross-project defect prediction studies [7] have adapted precision, recall, f1-score, and the area under the receiver operating characteristic curve ($AUCROC$) as performance measures, which are widely used in prediction problems. These performance measures indicate how many testing instances are predicted correctly by a classification model. However, they do not consider the effort for QA testers to inspect the instances predicted as defects, which makes the measures less practical for QA testers [17].

Instead, we use the area under the cost-effectiveness curve ($AUCCE$) [18] as a performance measure in this experiment. This measure considers the effort to investigate the source code, which enables more practical evaluation.

As developers examine the commits that are ordered by the defect-proneness one by one, the total number of LOCs investigated and the total number of defects found will increase. The cost-effectiveness curve, which is a monotonic function, plots changes in the total percentage of LOCs investigated

TABLE I
STATISTICS ABOUT SIZE OF FIVE PROJECTS

Project	# of Sub-projects	Defect Ratio	Avg. # of Contributed Sub-projects per Dev.	# of Commits							
				Sum	*Mean*	*Std.*	*Min.*	*25%*	*50%*	*75%*	*Max.*
Android	45	0.12	2.68	248860	5530.22	7751.91	53.0	568.00	3070.0	6732.00	39449.0
Appium	32	0.21	1.79	18034	563.56	1108.50	51.0	127.50	253.0	541.00	6326.0
Cordova	38	0.16	2.60	23551	619.76	808.88	52.0	197.25	330.5	618.75	3494.0
Maven	69	0.19	6.50	43703	633.38	1288.80	51.0	187.00	296.0	614.00	10344.0
Tizen	48	0.21	2.09	9857	205.35	518.24	52.0	77.25	116.0	153.00	3677.0

TABLE II
DESCRIPTIONS OF CHANGE METRICS

Name	Description
NS	total # of changed subsystems
ND	total # of changed directories
NF	total # of changed files
Entropy	distribution of modified code across files
LA	total # of added code lines
LD	total # of deleted code lines
LT	average # of code lines before the change
FIX	whether containing fix related keyword in a change log
NDEV	average # of developers who touched a file so far
AGE	average # of days passed since the last modification
NUC	total # of unique changes
EXP	# of commits an author made in a sub-project
SEXP	# of commits an author made in a subsystem
REXP	# of commits an author made in a sub-project, weighted by commit time

to the horizontal axis and changes in the total percentage of defects found to the vertical axis. Thus, a higher $AUCCE$ value can be achieved if defect-inducing commits that changed only small amount of source code are investigated earlier.

Although $AUCCE$ values are always between 0 and 1, the maximum and minimum achievable $AUCCE$ values can differ across model learners and sub-projects, so it is difficult to understand the overall performance of defect prediction models. Thus, we normalized the $AUCCE$ value of a prediction model by dividing the value by the $AUCCE$ value of a within-sub-project model. This percentage of $AUCCE$ from the within-sub-project model ($\%WSP_{AUCCE}$)" shows the cost effectiveness a JIT prediction model achieves compared to the within-sub-project model.

III. EXPERIMENTS

A. RQ1: Is the cross-sub-project model trained by sub-projects from the same project more cost effective than that trained by sub-projects from other projects?

1) Design: We evaluate the median $\%WSP_{AUCCE}$ of the two cross-sub-project models across all sub-projects. The first model, a within-project model, is built with commits from sub-projects that belong to the same project as a target sub-project. The other, a cross-project model, is constructed with commits from sub-projects from the other four projects. The goal of both cross-sub-project models—within-project model and cross-project model—are predicting defect-inducing commits within the target sub-project. Fig. 1 shows which training data are selected for two cross-sub-project models and one within-sub-project model. A rectangle represents the commit data of each sub-project and a rounded rectangle represents a project. Sub-projects within a dashed line are selected as training data for each JIT defect prediction model.

In the process of building a JIT defect prediction model, especially when applying the under-sampling method, there is randomness that yields non-deterministic experimental results. Thus, we repeated the experiments 30 times to minimize the effect of randomness. In addition, we statistically tested whether the cost effectiveness of the two models is significantly different. We used the Wilcoxon signed-rank test [19] for the test as the $\%WSP_{AUCCE}$ values are paired between the two models and distribution of the $\%WSP_{AUCCE}$ values are not from a normal distribution. In addition, we calculated effect sizes of the Wilcoxon signed-rank tests to see the $\%WSP_{AUCCE}$ difference in the two models.

2) Result: With 41,760 experimental results (3 model learners $\times$ 2 cross-sub-project models $\times$ 232 sub-projects $\times$ 30 repetitions), Table III shows the median $\%WSP_{AUCCE}$ value of two models on different projects and model learners. The *Diff.* row represents the differences in the median $\%WSP_{AUCCE}$ between the two models and the number in the parenthesis represents the effect size between models. Stars next to the effect size represent that there is a statistically significant difference between the performance measures of the two models. Three stars to one star represent a statistical significance at 99% ($\alpha = 0.01$), 95% ($\alpha = 0.05$) and 90% ($\alpha = 0.1$) confidence intervals, respectively.

As shown in Table III, the within-project model outperformed the cross-project model in terms of the median $\%WSP_{AUCCE}$ value, except in cases when logistic regression and naive Bayes learners are used in the Android project and logistic regression is used in the Cordova project. Although the cross-project models are built with more training data than the within-project models, their cost effectiveness is generally lower than that of the within-project models. It shows an evidence that more training data does not mean better performance. These results show that when applying cross-sub-project JIT defect prediction, an organization may not need to collect change data from other projects because prediction models built with those data may not perform better than those built with change data from sub-projects

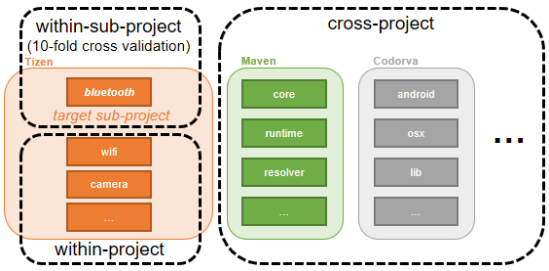

Fig. 1. Training data of within- and cross-project models

within the project. It is possible that filtering training data may increase the cost effectiveness of the cross-sub-project JIT prediction model. In this paper, however, we will focus on the feasibility of using within-project sub-project data and further data utilization would be handled in future work.

Another finding from the results is that the within-project models seem to have higher cost effectiveness than the within-sub-project models. Across all projects, except Android and all three model learners, the within-project models achieved a median $\%WSP_{AUCCE}$ value higher than 1.0, meaning that more than half of the within-project models achieved higher $AUCCE$ than the within-sub-project models. This result is different from the result of Kamei et al. [11] because, in their work, which was conducted at the project level, there was almost no cross-project JIT prediction models that achieved higher $AUCROC$ than the within-project model. We may explain a possible reason for the difference with the number of commits per sub-project. Table I shows that, except in the Android project, where the mean number of commits per sub-project is up to 27 times higher than in other projects, median commit counts (*50%* column) of the sub-project per project are near 500, which is much smaller than the number of commits per project (*Sum* column). We conducted additional experiments to confirm whether cross-sub-projects can be beneficial to sub-projects with a small number of commits, as explained in section I. The Spearman correlation coefficient between the number of commits in a target sub-project and the $\%WSP_{AUCCE}$ value of its within-project model is -0.379, indicating that there is a negative linear relationship. This means that the fewer commits a sub-project has, the greater the improvement in $AUCCE$ that its cross-sub-project models can achieve.

B. RQ2: Which sub-project ranking method performs best in filtered-within-project JIT defect prediction?

1) Design: We see from Section III-A2 that a small amount of training data could lead to better prediction performance. Thus, instead of using all the sub-project data within a project to build a cross-sub-project JIT defect-prediction model, we can filter out sub-projects that are less helpful in defect prediction to improve cost effectiveness of the JIT defect-prediction model. Fig. 2 shows how the training data of such models are selected. First, we calculate a score for each sub-project by using a ranking method. Then, we rank sub-projects by their scores and choose the top N sub-projects as training data for the filtered-within-project model. We use two different sub-project similarity-based ranking methods and two baseline ranking methods for this research question. The first is the developer similarity, which is our proposed method. The developer similarity is calculated with the number of developers who made any commit in both sub-projects. Sub-projects with more contributing developers in the target sub-project achieve higher similarity. As it can be seen in the average number of contributed sub-projects per developer in Table I, people tend to contribute to various sub-project within the project. The second is domain-agnostic similarity, used in the work by Kamei et al. [11]. Domain-agnostic similarity measure between the target sub-project and another sub-project is calculated in this order: 1) calculate the Spearman correlation between label values, which is 1 if a commit introduced any defect or 0 otherwise, and the values of each metric from the other sub-project; 2) select three metrics that achieved the highest Spearman correlation values; 3) calculate the Spearman correlation between each unordered pair of selected metrics for each sub-project. This generates a three-dimensional vector ($\binom{3}{2}$) for each sub-project; 4) Calculate the domain-agnostic similarity by the Euclidean distance between the two vectors. A smaller distance represents greater similarity.

Two baseline ranking methods we used are random and size rank. *Random rank* ranks the sub-projects randomly as a dummy baseline, and *size rank* ranks the sub-projects by the number of commits. The more commits a sub-project has, the higher the rank it achieves. Size ranking is comparable with our proposed ranking method because our method is correlated with the size of the sub-project.

We build cross-sub-project JIT defect prediction models for a target sub-project with commits from 1, 3, 5, 10, and 20 highest ranked sub-projects for each ranking method. Similar

TABLE III
MEDIAN $\%WSP_{AUCCE}$ VALUE OF WITHIN- AND CROSS-PROJECT MODELS ACROSS FIVE PROJECTS AND THREE MODEL LEARNERS

		Model Learner		
Project	**Model**	***LR***	***NB***	***RF***
Android	Cross	1.00	1.14	0.94
	Within	0.99	1.00	1.01
	Diff.	-0.01(-0.17**)	-0.14(-1.00**)	+0.06(+0.54**)
Appium	Cross	1.02	0.95	1.01
	Within	1.05	1.03	1.02
	Diff	+0.02(+0.77**)	+0.09(+0.89**)	+0.01(+0.08*)
Cordova	Cross	1.07	0.88	1.02
	Within	1.06	1.02	1.06
	Diff	-0.01(+0.02)	+0.15(+0.99**)	+0.04(+0.5***)
Maven	Cross	1.00	0.77	0.96
	Within	1.11	1.05	1.11
	Diff	+0.11(+0.98**)	+0.27(+1.00**)	+0.14(+0.96**)
Tizen	Cross	0.80	0.77	0.88
	Within	1.10	1.03	1.08
	Diff	+0.30(+0.99**)	+0.26(+0.98**)	+0.20(+0.96**)
All	Cross	0.99	0.90	0.96
	Within	1.06	1.02	1.05
	Diff	+0.07(+0.65**)	+0.13(+0.65**)	+0.09(+0.76**)

to RQ1, we evaluate the median $\%WSP_{AUCCE}$ values from each cross-sub-project JIT defect prediction model, repeat the experiments 30 times. P-values and effect sizes using the Wilcoxon signed-rank test are also calculated to compare performance of developer similarity based ranking method and other ranking methods.

2) Result: As Table IV shows, when only one sub-project is selected to build a filtered-within-project JIT defect prediction model, the developer similarity ranking method outperforms the domain-agnostic similarity and random ranking in all three learners and outperforms the size ranking in two learners. This result shows that a similarity measure designed for sub-project-level JIT defect prediction can be preferred over a similarity measure that is originally designed for cross-project JIT defect prediction when picking a sub-project to build a cross-sub-project JIT defect-prediction model.

As a prediction model is built with more sub-projects, median cost effectiveness generally increases and the differences of median $\%WSP_{AUCCE}$ value between four ranking methods become smaller. When we conduct experiments with more than 20 sub-projects selected for each model, the cost effectiveness of the various models becomes almost the same, so we do not include them in the table. This may be because as more number sub-projects are selected to build the filtered-within-project JIT defect prediction models, the models are trained with a more similar set of training data.

When we additionally refined our developer similarity ranking methods by not considering developers who barely contributed to the sub-project or by normalizing its value with the total number of developers who contributed in the other sub-project. Table V shows the comparison of performance measures normalization by the number of developers in the other sub-project is applied or not. Since filtering developer does not success in selecting similar sub-projects to get improved performance measures, we do not insert the table for that. However, in case of the normalization, it improved performance measures greatly when Naive Bayes learner is used and 3 to 5 sub-projects are selected. The improved performance measures even exceed the median $\%WSP_{AUCCE}$ value of within-project models which can be found in the "all" row at Table III.

In this research question, we see that the developer similarity-based cross-sub-project JIT defect prediction model is preferable to the other ranking methods. However, we notice that the median values of the performance measure achieved with filtered-within-project models (Table IV) are smaller than those achieved with within-project models (Table III). Still, we find the evidence that there is still room for improving the cost effectiveness of filtered-within-project models over that of within-project models with conducting additional experiments.

IV. Related Work

A. JIT Defect Prediction

Most studies on defect prediction have focused on predicting the defectiveness of software modules—files, packages, or functions—by utilizing project history data [20]. Recently, some research has been performed on just-in-time defect prediction, which predicts software changes that may introduce defects.

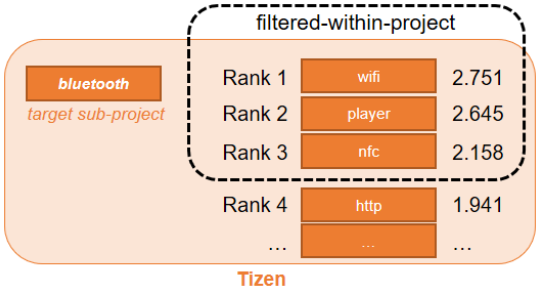

Fig. 2. Training data of filtered-within-project models

Kamei et al. conducted experiments applying just-in-time defect prediction to six large open-source projects and five large commercial projects [4]. To build the defect prediction models, they used 14 change metrics in five different dimensions—diffusion, size, purpose, history, and experience—and used the logistic regression learner. These change metrics are widely used in other just-in-time defect prediction studies [21], [22]. Kim et al. proposed a change level defect prediction method and tested it on 12 open-source software projects [5]. They used a set of text-based metrics that were extracted from the source code, log messages, and file names. In addition, they used metadata such as the change author and commit time. They also considered changes in the complexity of source code caused by commits. There are many other just-in-time defect prediction studies, such as applying deep learning [21] or unsupervised models [22] for JIT defect prediction. However, applying just-in-time defect prediction on projects that consist of sub-projects is not discussed yet.

B. Cross-Project Defect Prediction

Zimmermann et al. defined 30 project characteristics and showed the influence of the similarity of each characteristic between target and predictor projects on a module-level cross-project defect prediction [23]. They considered the project domain, programming language, company, quality assurance tools, and other aspects as project characteristics and concluded that the characteristics of a project that transfers a defect prediction model can influence precision, recall, and accuracy in cross-project defect prediction. However, we notice that this finding is barely applicable in cross-sub-project defect prediction, as sub-projects within the project share many common characteristics. For instance, sub-projects within the same project usually are developed by the same company using the same programming language and development tools. He et al. [8] investigated the feasibility of cross-project defect prediction at the module-level on 10 open-source projects. They concluded that a model trained with data from other projects can achieve higher accuracy than one trained with data from the target project in the best cases. In addition, they said that selecting the training dataset considering the distributional

TABLE IV
MEDIAN $\%WSP_{AUCCE}$ VALUE OF VARIOUS FILTERED-WITHIN-PROJECT MODELS ACROSS FIVE PROJECTS AND THREE MODEL LEARNERS

Model Learner	Ranking Method	# of Sub-projects Selected			
		1	*3*	*5*	*10*
LR	Developer	1.02	1.04	1.05	1.06
	Domain-agnostic	1.01(-0.02, -0.26**)	1.04(-0.01, -0.13**)	1.05(-0.00, -0.05**)	1.05(-0.01, -0.05**)
	Random	0.99(-0.04, -0.38**)	1.03(-0.02, -0.30**)	1.04(-0.01, -0.22**)	1.05(-0.01, -0.18**)
	Size	1.04(+0.02, +0.30**)	1.05(+0.01, -0.13**)	1.05(+0.00, -0.15**)	1.06(-0.00, -0.20**)
NB	Developer	1.03	1.03	1.03	1.03
	Domain-agnostic	1.01(-0.02, -0.25**)	1.04(+0.01, +0.19**)	1.04(+0.01, +0.24**)	1.04(+0.01, +0.24**)
	Random	1.01(-0.02, -0.26**)	1.03(+0.00, -0.01)	1.03(+0.00, +0.04**)	1.03(+0.00, +0.13**)
	Size	1.00(-0.03, -0.33**)	1.01(-0.02, -0.47**)	1.01(-0.02, -0.46**)	1.01(-0.02, -0.30**)
RF	Developer	0.98	1.01	1.02	1.03
	Domain-agnostic	0.96(-0.02, -0.14**)	1.01(-0.00, -0.09**)	1.02(-0.00, -0.09**)	1.03(+0.00, -0.01)
	Random	0.94(-0.04, -0.29**)	0.99(-0.02, -0.30**)	1.01(-0.02, -0.29**)	1.02(-0.01, -0.20**)
	Size	0.96(-0.02, -0.18**)	0.98(-0.03, -0.44**)	0.99(-0.03, -0.45**)	1.02(-0.02, -0.32**)

TABLE V
MEDIAN $\%WSP_{AUCCE}$ VALUE OF NORMALIZED DEVELOPER SIMILARITY-BASED FILTERED-WITHIN-PROJECT MODELS ACROSS FIVE PROJECTS AND THREE MODEL LEARNERS

Model Learner	Developer Similarity	# of Sub-projects Selected			
		1	*3*	*5*	*10*
LR	Not Normalized	1.02	1.04	1.05	1.06
	Normalized	0.95(-0.08, -0.62**)	1.02(-0.03, -0.46**)	1.03(-0.02, -0.38**)	1.05(-0.01, -0.32**)
NB	Not Normalized	1.03	1.03	1.03	1.03
	Normalized	0.99(-0.04, -0.30**)	1.06(+0.03, +0.22**)	1.06(+0.03, +0.24**)	1.04(+0.02, +0.25**)
RF	Not Normalized	0.98	1.01	1.02	1.03
	Normalized	0.92(-0.06, -0.43**)	0.98(-0.03, -0.35**)	1.00(-0.03, -0.34**)	1.02(-0.01, -0.22**)

characteristics of datasets could lead to better cross-project results. However, this characteristic could not be extracted in projects where no historical data exist, which hinders the application of distributional characteristics in cross-project defect prediction. In addition, neither Zimmermann et al. nor He et al. considered JIT defect prediction and the cost effectiveness of the defect-prediction model.

Kamei et al. [11] conducted cross-project defect prediction experiments in a change-level manner. They observed that predicting defects for changes in a target project with a model built with another project's data has lower accuracy than that of a within-project model. However, when the prediction is done with a model built with a larger pool of training data from multiple projects or combining the prediction results from multiple cross-project models, its performance is indistinguishable from that of a within-project model. They also applied domain-aware or domain-agnostic similarity measures between two projects to select a similar project. For the domain-agnostic type, they calculated the Spearman correlation between the metric values within a dataset and used the correlation values to find a similar project. For the domain-aware type, they used a subset of project characteristics proposed by Zimmermann et al. [23] and calculated the Euclidean distance to find a similar project. When these similarity measures are used to pick one project to transfer its JIT defect-prediction model to predict defects in the target project, they concluded that both measures successfully contributed to pick a better-than-average cross-project model. However, when they built a cross-project JIT defect prediction model with training data from multiple similar projects, it barely improved accuracy over a model trained with data from all other projects without filtering irrelevant projects. This work showed that cross-project defect prediction is feasible on JIT defect prediction, but there were no discussions on the sub-project-level JIT defect prediction. Moreover, the cost was not considered of investigating commits to find defects in the evaluation.

V. THREAD TO VALIDITY

A. Construct Validity

For our experiments, we implemented Python scripts to extract the change metric data from the source code repositories to build and test JIT defect prediction models. However, the scripts might have defects that affect the experiments and results. To reduce this threat, we used open-source frameworks and libraries that are commonly used in other studies, such as *scikit-learn* [24]. In addition, we also double-checked our source code and manually inspected extracted change measures for validation.

B. Dataset Quality

We used *CodeRepoAnalyzer* by Rosen et al. [25] to extract change metrics from git repositories. While using this tool, we noticed that it has some bugs. For instance, some extracted metric values were marked as negative, which is incorrect by definition. Although we handled the bugs found in this tool, there could be other bugs that were not found and that could have affected the extracted values.

Although the SZZ algorithm is widely used in JIT defect prediction research [4], [12], it is known that the accuracy of keyword-based labelling methods for bug-inducing commits are limited [26]. We may improve the accuracy of automatic labeling by utilizing bug-repository data [13].

VI. Conclusion

In this paper, we investigated the feasibility of transferring JIT defect prediction models built with data from other sub-projects to predict bug-inducing commits in a target sub-project. We conducted experiments with five projects, which comprise 232 sub-projects in total, and three different model learners. With two research questions, we conclude that 1) a cross-sub-project model has better cost effectiveness than within-sub-project models in general; 2) a cross-sub-project JIT defect prediction model built with data from sub-projects within the targets project has higher cost effectiveness than a JIT defect prediction model built with data from all available sub-projects; 3) the developer similarity-based ranking method is preferable for filtering sub-projects that are irrelevant to the target sub-project; and 4) although a developer similarity-based filtered-within-project model has lower cost effectiveness than a within-project model in general, we further improved the performance of the filtered-within-project model to outperform the within-project model in the best cases. Our contributions include 1) proposing defect prediction at the sub-project level that potentially has better cost effectiveness than traditional within-project models by using new developer-similarity-based filtered-within-project models and 2) initially evaluating the cost effectiveness of various sub-project-level JIT defect prediction models across 232 sub-projects. In future work, We plan to investigate a more polished way to apply filtered-within-project models, such as filtering developers by considering their contributions over various project resources [27] before calculating the developer similarity.

References

[1] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium On*. IEEE, 2004, pp. 417–428.

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

[3] R. Moser, W. Pedrycz, and G. Succi, "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 181–190.

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

[5] S. Kim, E. J. W. Jr, and Y. Zhang, "Classifying Software Changes: Clean or Buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, Mar. 2008.

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

[7] S. Herbold, "A systematic mapping study on cross-project defect prediction," *arXiv:1705.06429 [cs]*, May 2017.

[8] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," *Automated Software Engineering*, vol. 19, no. 2, pp. 167–199, Jun. 2012.

[9] E. Shihab, Y. Kamei, and P. Bhattacharya, "Mining Challenge 2012: The Android Platform," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, ser. MSR '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 112–115.

[10] M. A. Babyak, "What you see may not be what you get: A brief, nontechnical introduction to overfitting in regression-type models," *Psychosomatic Medicine*, vol. 66, no. 3, pp. 411–421, 2004 May-Jun.

[11] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2072–2106, Oct. 2016.

[12] T. Jiang, L. Tan, and S. Kim, "Personalized Defect Prediction," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 279–289.

[13] J. Śliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5.

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

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

[16] T. Mende and R. Koschke, "Effort-Aware Defect Prediction Models," in *2010 14th European Conference on Software Maintenance and Reengineering*, Mar. 2010, pp. 107–116.

[17] Y. Kamei and E. Shihab, "Defect Prediction: Accomplishments and Future Challenges," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, Mar. 2016, pp. 33–45.

[18] 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*, ser. PROMISE '09. New York, NY, USA: ACM, 2009, pp. 7:1–7:10.

[19] F. Wilcoxon, "Individual Comparisons by Ranking Methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[20] J. Nam, "Survey on software defect prediction," *Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep*, 2014.

[21] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep Learning for Just-in-Time Defect Prediction," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, Aug. 2015, pp. 17–26.

[22] W. Fu and T. Menzies, "Revisiting Unsupervised Learning for Defect Prediction," *arXiv:1703.00132 [cs]*, Feb. 2017.

[23] 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*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 91–100.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, and D. Cournapeau, "Scikit-learn: Machine Learning in Python," *MACHINE LEARNING IN PYTHON*, p. 6.

[25] C. Rosen, B. Grawi, and E. Shihab, "Commit Guru: Analytics and Risk Prediction of Software Commits," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 966–969.

[26] T. Hall, D. Bowes, G. Liebchen, and P. Wernick, "Evaluating Three Approaches to Extracting Fault Data from Software Change Repositories," in *Product-Focused Software Process Improvement*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Jun. 2010, pp. 107–115.

[27] G. Gousios, E. Kalliamvakou, and D. Spinellis, "Measuring Developer Contribution from Software Repository Data," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 129–132.

