An Empirical Study of Just-in-Time Defect Prediction using Cross-Project Models

Takafumi Fukushima[1], Yasutaka Kamei[1], Shane McIntosh[2],
Kazuhiro Yamashita[1], and Naoyasu Ubayashi[1]
[1]Principles of Software Languages Group (POSL), Kyushu University, Japan
[2]Software Analysis and Intelligence Lab (SAIL), Queen's University, Canada
[1]{f.taka, yamashita}@posl.ait.kyushu-u.ac.jp, {kamei, ubayashi}@ait.kyushu-u.ac.jp,
[2]mcintosh@cs.queensu.ca

ABSTRACT

Prior research suggests that predicting defect-inducing changes, i.e., Just-In-Time (JIT) defect prediction is a more practical alternative to traditional defect prediction techniques, providing immediate feedback while design decisions are still fresh in the minds of developers. Unfortunately, similar to traditional defect prediction models, JIT models require a large amount of training data, which is not available when projects are in initial development phases. To address this flaw in traditional defect prediction, prior work has proposed cross-project models, i.e., models learned from older projects with sufficient history. However, cross-project models have not yet been explored in the context of JIT prediction. Therefore, in this study, we empirically evaluate the performance of JIT cross-project models. Through a case study on 11 open source projects, we find that in a JIT cross-project context: (1) high performance within-project models rarely perform well; (2) models trained on projects that have similar correlations between predictor and dependent variables often perform well; and (3) ensemble learning techniques that leverage historical data from several other projects (e.g., voting experts) often perform well. Our findings empirically confirm that JIT cross-project models learned using other projects are a viable solution for projects with little historical data. However, JIT cross-project models perform best when the data used to learn them is carefully selected.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Process metrics*

General Terms

Management, Measurement

Keywords

Empirical study, software quality

MSR'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2863-0/14/05...$15.00
http://dx.doi.org/10.1145/2597073.2597075

1. INTRODUCTION

Software Quality Assurance (SQA) activities, such as code inspection and unit testing are standard practices for improving software quality prior to official release. However, software teams have limited testing resources, and must wisely allocate them to minimize the risk of incurring *post-release defects*, i.e., defects that appear in official software releases. For this reason, a plethora of software engineering research is focused on prioritizing SQA activities [18]. For example, defect prediction techniques are often used to prioritize modules (i.e., files or packages) based on their likelihood of containing post-release defects [1, 18]. Using these techniques, practitioners can allocate limited SQA resources to the most defect-prone modules.

However, recent work shows that traditional prediction models often make recommendations at a granularity that is too coarse to be applied in practice [12, 30]. For example, since the largest files or packages are often the most defect-prone [16], they are often suggested by traditional defect models for further inspection. Yet, carefully inspecting large files or packages is not practical for two reasons: (1) the design decisions made by when the code was initially produced may be difficult to for a developer to recall or recover; and (2) it may not be clear which developer should perform the inspection tasks, since many developers often work on the same files or packages [14].

To address these flaws in traditional defect prediction, prior work has proposed change-level defect prediction models, i.e., models that predict the code changes that are likely to introduce defects [12, 14, 22, 30, 31]. The advantages of change-level predictions are that: (1) the predictions are made at a fine granularity, since changes often impact only a small area of the code; and (2) the predictions can be easily assigned, since each change has an author who can perform the inspection while design decisions are still fresh in their mind. We refer to change-level defect prediction as "Just-In-Time (JIT) defect prediction" [12].

Despite the advantages of JIT defect models, like all prediction models, they require a large amount of historical data in order to train a model that will perform well [36]. However, in practice, training data may not be available for projects in the initial development phases, or for legacy systems that have not archived historical data. To overcome this, prior work has proposed *cross-project defect prediction models*, i.e., models trained using historical data from other projects [36].

While studies have shown that cross-project defect prediction models can perform well at the file-level [2, 20], cross-project defect prediction using JIT models remains largely unexplored. We, therefore, set out to empirically study the performance of JIT cross-project defect models. Using data from 11 open source projects, we test JIT cross-project models trained using three techniques, and address the following research questions:

(RQ1) Are high performance within-project models also high performance cross-project models? JIT models with strong within-project performance rarely perform well in a cross-project context.

(RQ2) Does similarity in the correlation between predictor and dependent variables indicate high performance cross-project models? Projects with similar correlation values between the predictors and the dependent variable tend to perform well in a cross-project context.

(RQ3) Do ensemble techniques improve cross-project prediction performance? Using the ensemble technique like "voting experts" [32] yields models that often perform well in a cross-project context.

Furthermore, we show that the combination of ensemble techniques with the carefully selected training projects based on predictor-dependent variable similarity yields models that outperform the other models learned using the studied techniques in isolation.

Paper organization. The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 motivates the three research questions that structure our study. Section 4 describes the setting of our empirical study, while Section 5 describes the results. Section 6 evaluates the combination of similarity (RQ2) and ensemble techniques (RQ3), as well as the impact of model threshold on prediction performance. Section 7 discloses the threats to the validity of our findings. Finally, Section 8 draws conclusions.

2. BACKGROUND AND RELATED WORK

In this section, we describe the related work with respect to traditional, JIT, and cross-project defect prediction.

2.1 Traditional Defect Prediction

Traditional defect prediction models describe the relationship between module metrics (e.g., SLOC and McCabe's Cyclomatic complexity) as predictor variables and a module status (defect-prone or not) as a response variable. In other words, given a module, a traditional defect model classifies it as either defect-prone or not.

Various techniques are used to build defect models, such as logistic regression and random forest. Many prior studies focus on the evaluation of prediction performance for additional modeling techniques like linear discriminant analysis [27], decision trees [13] and Support Vector Machines (SVM) [35]. In this paper, we train our JIT cross-project models using the random forest algorithm, since compared to conventional modeling techniques (e.g., logistic regression and decision trees), random forest produces robust, highly accurate, stable models that are especially resilient to noisy data [9]. Furthermore, prior studies have shown that random forest tends to outperform other modeling techniques [10, 17].

Random Forest: Random forest is a classification (or regression) technique that builds a large number of decision trees at training time [3]. Each node in the decision tree is split using a random subset of all of the attributes. Performing this random split ensures that all of the trees have a low correlation between them [3].

First, the dataset is split into training and testing corpora. Typically, 90% of the dataset is allocated to the training corpus, which is used to build the forest. The remaining 10% of the dataset is allocated to the testing or Out Of Bag (OOB) corpus, which is used to test the prediction accuracy of the forest. Since there are many decision trees that may each report different outcomes, each sample in the OOB corpus is pushed down all of the trees in the forest and the final class of the sample is decided by aggregating the votes from all of the trees.

2.2 Just-In-Time Defect Prediction

While traditional defect prediction models use module metrics for predictor variables and module status (defect-prone or not) as a response variable, JIT defect prediction model uses change metrics (e.g., # modified files) and change status (i.e., defect-inducing or not).

Prior work suggests that JIT prediction is a more practical alternative to traditional defect prediction. For example, Mockus *et al.* [22] predict defect-inducing changes in a large-scale telecommunication system. Kim *et al.* [14] add change features, such as the terms in added and deleted deltas, modified file and directory names, change logs, source code, change metadata and complexity metrics to classify changes as being defect-inducing or not. Kamei *et al.* [12] also perform a large-scale study on the effectiveness of JIT defect prediction, reporting that the addition of a variety of factors extracted from commits and bug reports helps to effectively predict defect-inducing changes. In addition, the authors show that using their technique, careful inspection of 20% of the changes could prevent up to 35% of the defect-inducing changes from impacting users.

The prior work not only establishes that JIT defect prediction is a more practical alternative to traditional defect prediction, it is also viable, yielding actionable results. However, defect models must be trained on a large corpus of data in order to perform well [36]. Since new projects and legacy ones may not have enough historical data available, we set out to study JIT cross-project defect prediction.

2.3 Cross-Project Defect Prediction

Cross-project defect prediction is also a well-studied research area. Several studies have explored traditional defect prediction using cross-project models [20, 26, 33, 36]. For example, Zimmermann *et al.* [36] study cross-project defect prediction models using 28 datasets collected from 12 open source and industrial projects. They find that of the 622 cross-project combinations, only 21 produce acceptable results. They also identify the factors that influence the success of cross-project prediction, such as the number of observations (file count, binary count and component count).

Turhan *et al.* [33] investigate the applicability of cross-project prediction for building localized defect predictors using static code features. They report that the proposed

Table 1: Summary of project data. Parenthesized values show the percentage of defect-introducing changes.

Project name	Period	# of changes	Project name	Period	# of changes
Bugzilla (BUG)	08/1998 - 12/2006	4,620 (37%)	Perl (PER)	12/1987 - 06/2013	50,485 (24%)
Columba (COL)	11/2002 - 07/2006	4,455 (31%)	Eclipse Platform (PLA)	05/2001 - 12/2007	64,250 (15%)
Gimp (GIP)	01/1997 - 06/2013	32,875 (36%)	PostgreSQL (POS)	07/1996 - 05/2010	20,431 (25%)
Eclipse JDT (JDT)	05/2001 - 12/2007	35,386 (14%)	Ruby on Rails (RUB)	11/2004 - 06/2013	32,866 (19%)
Maven-2 (MAV)	09/2003 - 05/2012	5,399 (10%)	Rhino (RHI)	04/1999 - 02/2013	2,955 (44%)
Mozilla (MOZ)	01/2000 - 12/2006	98,275 (5%)	Median		32,866(24%)

cross-project prediction models actually outperform models built using within-project data.

Menzies *et al.* [20] comparatively evaluate local (within-project) vs. global (cross-project) lessons learned for defect prediction. They report that a strong prediction model can be built from projects that are included in the cluster that is nearest to the testing data. Furthermore, Nam *et al.* [26] use the transfer learning approach (TCA) to make feature distributions in training and testing projects similar. They also propose a novel transfer learning approach, TCA+, by extending TCA. They report that TCA+ significantly improves cross-project prediction performance in eight open source projects.

While prior studies have empirically evaluated cross-project prediction performance using traditional models, our study focuses on cross-project prediction using JIT models.

3. RESEARCH QUESTIONS

We suspect that the performance of JIT cross-project prediction models will improve if we select an appropriate training dataset [20, 33]. Hence, we set out to compare model performance when we apply three techniques for training dataset preprocessing. To structure our paper, we formulate each technique as a research question as listed below.

(RQ1) Are high performance within-project models also high performance cross-project models?
High performance within-project models have established a strong link between predictors and defect-proneness within one project. We suspect that properties of the relationship may still hold if the model is tested on another project.

(RQ2) Does similarity in the correlation between predictor and dependent variables indicate high performance cross-project models?
Defect prediction models assume that the distributions of the metrics in the training and testing datasets are similar [33]. Since the distribution of metrics can vary among projects, this assumption may be violated in a cross-project context. In such cases, we would expect that cross-project model performance would suffer. On the other hand, we expect that models trained using data from projects with similar metric distributions will have strong prediction performance.

(RQ3) Do ensemble techniques improve cross-project prediction performance?
Since ensemble classification techniques have recently proved useful in other areas of software engineering [15], we suspect that they may also improve JIT cross-project defect prediction. Ensemble techniques that leverage multiple datasets and/or methods could cover a large project characteristic space, and hence provide high performance for general prediction purposes, i.e., not only those of one project.

4. EXPERIMENTAL SETTING

4.1 Studied Systems

In order to address our three research questions, we conduct an empirical study using data from 11 open source projects, of which 6 projects (Bugzilla, Columba, Eclipse JDT, Mozilla, Eclipse Platform, PostgreSQL) are provided by Kamei *et al.* [12] and 5 well-known and long-lived projects (Gimp, Maven-2, Perl, Ruby on Rails, Rhino) needed to be collected. We study projects from various domains in order to combat potential bias in our results. Table 1 provides an overview of the studied datasets.

4.2 Change Measures

A previous study of JIT defect prediction uses 14 metrics from 5 categories derived from the Version Control System (VCS) of a project to predict defect-inducing changes [12]. We remove six of these metrics in the History and Experience categories because these metrics are project-specific, and hence cannot be measured from the software projects that do not have change histories (e.g., a new development project).

Identification of defect-inducing changes: To know whether or not a change introduces a defect, we used the SZZ algorithm [31]. This algorithm identifies when a bug was injected into the code and who injected it using a VCS.

Table 2 provides a brief description of each metric and the rationale behind it. We briefly describe each metric below.

Diffusion category: We expect that the diffusion dimension can be leveraged to determine the likelihood of a defect-inducing change. A total of four different factors makes up the diffusion dimension, as listed in Table 2.

Prior work has shown that a highly distributed change can be more complex and harder to understand [22]. For example, Mockus and Weiss [22] show that the number of changed subsystems is related to defect-proneness. Hassan [8] shows that change entropy is a more powerful predictors of the incidence of defects than the number of prior defects or changes. In our study, we normalize the change entropy by the maximum entropy $log_2 n$ to account for differences in the number of files n across changes, similar to Hassan [8].

For each change, we count the number of distinct names of modified: (1) subsystems (i.e., root directories) (NS), (2) directories (ND) and (3) changed files (NF). To illustrate, if a change modifies a file with the path: `org.eclipse.jdt.core/jdom/org/eclipse/jdt/core/dom/Node.java`, then the subsystem is `org.eclipse.jdt.core`, the directory is `org.eclipse.jdt.core/jdom/.../dom` and the file name is `org.`

Table 2: Summary of change measures [12]

Dim.	Name	Definition	Rationale	Related Work
Diffusion	NS	Number of modified subsystems	Changes modifying many subsystems are more likely to be defect-prone.	The defect probability of a change increases with the number of modified subsystems [22].
	ND	Number of modified directories	Changes that modify many directories are more likely to be defect-prone.	The higher the number of modified directories, the higher the chance that a change will induce a defect [22].
	NF	Number of modified files	Changes touching many files are more likely to be defect-prone.	The number of classes in a module is a good feature of post-release defects of a module [25]
	Entropy	Distribution of modified code across each file	Changes with high entropy are more likely to be defect-prone, because a developer will have to recall and track large numbers of scattered changes across each file.	Scattered changes are more likely to introduce defects [5, 8].
Size	LA	Lines of code added	The more lines of code added, the more likely a defect is introduced.	Relative code churn measures are good indicators of defect modules [23, 24].
	LD	Lines of code deleted	The more lines of code deleted, the higher the chance of a defect.	
	LT	Lines of code in a file before the change	The larger a file, the more likely a change might introduce a defect.	Larger modules contribute more defects [16].
Purpose	FIX	Whether or not the change is a defect fix	Fixing a defect means that an error was made in an earlier implementation, therefore it may indicate an area where errors are more likely.	Changes that fix defects are more likely to introduce defects than changes that implement new functionality [7][28].
History*	NDEV	The number of developers that changed the modified files	The larger the NDEV, the more likely a defect is introduced, because files revised by many developers often contain different design thoughts and coding styles.	Files previously touched by more developers contain more defects [19].
	AGE	The average time interval between the last and the current change	The **lower** the AGE (i.e., the more recent the last change), the more likely a defect will be introduced.	More recent changes contribute more defects than older changes [6].
	NUC	The number of unique changes to the modified files	The larger the NUC, the more likely a defect is introduced, because a developer will have to recall and track many previous changes.	The larger the spread of modified files, the higher the complexity [5, 8].
Experience*	EXP	Developer experience	More experienced developers are less likely to introduce a defect.	Programmer experience significantly reduces the likelihood of introducing a defect [22]. Developer experience is measured as the number of changes made by the developer before the current change.
	REXP	Recent developer experience	A developer that has often modified the files in recent months is less likely to introduce a defect, because she will be more familiar with the recent developments in the system.	
	SEXP	Developer experience on a subsystem	Developers that are familiar with the subsystems modified by a change are less likely to introduce a defect.	

* These metrics cannot be measured from the software projects that do not have change histories, and hence cannot be used in cross-project context.

`eclipse.jdt.core/jdom/org/eclipse/jdt/core/dom/Node.java`.

Size category: In addition to the diffusion of a change, prior work shows that the size of a change is a strong indicator of its defect-proneness [23, 24]. Hence, we use the size dimension to predict defect-inducing changes. We use three different LA, LD, and LT metrics to measure the size dimensions as shown in Table 2. These metrics can be extracted directly from a VCS.

Purpose category: A change that fixes a defect is more likely to introduce another defect [7, 28]. The intuition being that the defect-prone modules of the past tend to remain defect-prone in the future [6].

To determine whether or not a change fixes a defect, we scan VCS commit messages that accompany changes for keywords like "bug", "fix", "defect" or "patch", and for defect identification numbers. A similar approach to determine defect-fixing changes was used in other work [12, 14].

4.3 Data Preparation

Minimizing collinearity. To combat the threat of multicollinearity in our models [12, 22], we remove highly correlated metrics (Spearman $\rho > 0.8$). We manually remove the highly correlated factors, avoiding the use of automatic techniques, such as stepwise variable selection because they may remove fundamental metrics (e.g., NF), in favour of a non-fundamental ones (e.g., NS) if the metrics are highly correlated. Since the fundamentality of a metric is somewhat subjective, we discuss each metrics that we discarded below.

We found that NS and ND are highly correlated . To address this, we exclude ND and include NS in our prediction models. We also found LA and LD are highly correlated. Nagappan and Ball [24] reported that relative churn metrics perform better than absolute metrics when predicting defect density. Therefore, we adopt their normalization approach, i.e., LA and LD are divided by LT. We also normalized LT by dividing it by NF, since this metric is highly correlated with NF. In short, the NS, NF, Entropy, relative churn (i.e., (LA+LD)/LT), LT/NF and FIX metrics metrics survive our correlation analysis.

Handling class imbalance. Our datasets are imbalanced, i.e., the number of defect-inducing changes represents only a small proportion of all changes. This imbalance may cause the performance of the prediction models to degrade if it is not handled properly [11]. Taking this into account, we use a re-sampling approach for our training data. We reduce the number of majority class instances (i.e., non-defect-inducing changes in the training data) by deleting instances randomly such that the majority class drops to the same level as the minority class (i.e., defect-inducing changes). Note that re-sampling is only performed on the training data – the testing data is not modified.

4.4 Performance Measure

To evaluate model prediction performance, precision, recall and F-measure are often used [14, 26]. However, as Lessmann *et al.* point out [17], these criteria depend on the threshold that is used for classification. Choosing a different threshold may lead to different results.

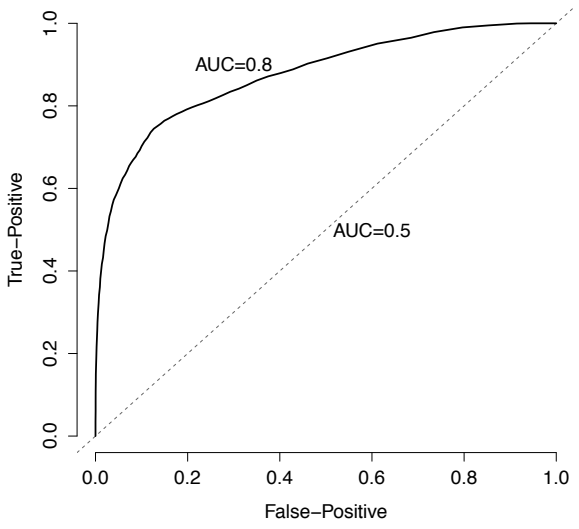

Figure 1: An example of ROC curve in the case of AUC=0.8 and AUC=0.5

To evaluate model prediction performance in a threshold-insensitive manner, we use the Area Under the Curve (AUC) of the Receiver Operating Characteristic (ROC) plot. Figure 1 shows an example ROC curve, which plots the false positive rate (i.e., the proportion of changes that are incorrectly classified as defect-inducing) on the x-axis and true positive rate (i.e., the proportion of defect-inducing changes that are classified as such) on the y-axis over all possible classification thresholds. The range of AUC is [0,1], where a larger AUC indicates better prediction performance. If the prediction accuracy is higher, the ROC curve becomes more convex in the upper left and the value of the AUC approaches 1. Any prediction model achieving an AUC above 0.5 is more effective than random predictions.

5. CASE STUDY

In this section, we present the results of our case study with respect to our three research questions.

(RQ1) Are high performance within-project models also high performance cross-project models?

Approach. We test all JIT cross-project model combinations available with our 11 datasets (i.e., 110 combinations $= 11 \times 10$). To address RQ1, we build prediction models using the historical data from one project for training and test the prediction performance using the historical data from each other project.

We validate whether or not datasets that have strong within-project prediction performance also perform well in a cross-project context. To measure the cross-project model performance, we test each within-project model using the data of all of the other projects. We use all of the data of each project to build the within-project model. We perform ten combinations of cross-project prediction (11 projects - 1 for training). We then select the median of the ten AUC values. This median value is referred to as the *cross-project AUC*.

To measure within-project performance, we select one project as the training dataset, perform tenfold cross-validation using data from the same project and then calculate the AUC value. The tenfold cross-validation process randomly divides one dataset into ten folds of equal sizes. The first nine folds are used to train the model, and the last fold is used to test it. This process is repeated ten times, using a different fold for testing each time. The prediction performance results of each fold are then aggregated. We refer to this aggregated value of within-project model performance as *within-project AUC*.

Finally, to evaluate RQ1, we compare within-project and cross-project AUC values.

Results. Table 3 shows the AUC values we obtain. Each row shows the projects used for testing and each column shows the projects used for training. Diagonal values (gray-colored cells) show the within-project AUC values. For example, the COL-COL cell is the AUC value of the tenfold cross-validation in the Columba project. Other cells show the cross-project prediction results. For example, the cell shown in boldface shows the performance of the prediction

Table 3: Summary of AUC values for within-project prediction and cross-project prediction

		Training project										
		BUG	COL	GIP	JDT	MAV	MOZ	PER	PLA	POS	RUB	RHI
Testing project	BUG	0.75	0.55	0.66	0.72	0.66	0.71	0.68	0.68	0.69	0.69	0.69
	COL	**0.56**	0.77	0.63	0.73	0.62	0.74	0.64	0.76	0.71	0.61	0.65
	GIP	0.47	0.47	0.79	0.69	0.63	0.58	0.68	0.60	0.66	0.62	0.69
	JDT	0.61	0.66	0.68	0.75	0.62	0.73	0.67	0.72	0.70	0.68	0.68
	MAV	0.38	0.63	0.76	0.72	0.83	0.76	0.75	0.79	0.72	0.73	0.75
	MOZ	0.69	0.64	0.74	0.74	0.69	0.80	0.73	0.74	0.77	0.74	0.75
	PER	0.57	0.49	0.69	0.67	0.65	0.63	0.75	0.60	0.66	0.69	0.72
	PLA	0.69	0.68	0.69	0.75	0.65	0.74	0.68	0.78	0.70	0.67	0.68
	POS	0.50	0.56	0.68	0.71	0.69	0.74	0.72	0.73	0.79	0.72	0.72
	RUB	0.51	0.60	0.63	0.65	0.62	0.65	0.70	0.64	0.66	0.74	0.68
	RHI	0.55	0.68	0.77	0.62	0.72	0.77	0.79	0.73	0.73	0.72	0.81

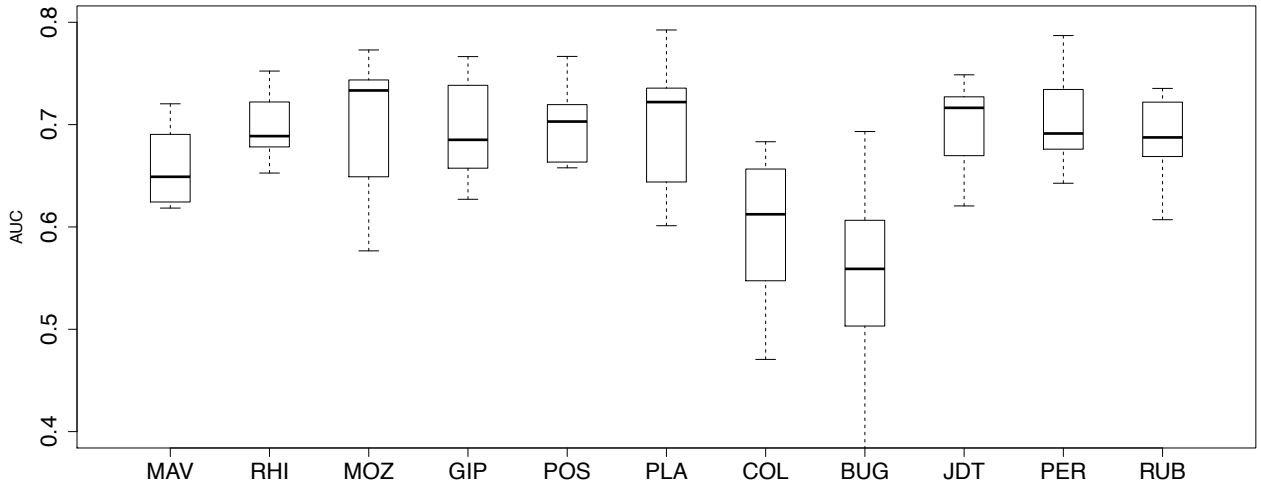

Figure 2: [RQ1] Within-project vs. cross-project model performance. Projects are sorted by within-project performance along the x-axis.

model learned using Bugzilla project data and tested using Columba project data.

Figure 2 groups the results of Table 3 in a boxplot. The projects are sorted along the X-axis by the AUC value of within-project prediction in descending order, and the Y-axis shows the 10 AUC values for cross-project performance. If there were truly a relationship between good within-project and cross-project prediction, one would expect that the boxes should also descend in value from left to right. Since no such pattern emerges, it seems that there is no relationship between good within-project predictors and good cross-project predictors. We validate our observation statistically using Spearman correlation tests. We calculate Spearman correlation between the rank of the AUC value of within-project prediction and the median of AUC value of cross-project prediction. The resulting value is $\rho = 0.036$ ($p = 0.9244$).

Strong within-project performance of a model does not necessarily indicate that it will perform well in a cross-project context.

(RQ2) Does similarity in the correlation between predictor and dependent variables indicate high performance cross-project models?

Approach. We validate whether or not we obtain better prediction performance when we use the models trained using a project that has similar characteristics with a testing project. Figure 3 provides an overview of our approach to calculate the similarity between two projects. We describe each step below:

1. We calculate Spearman correlation between a dependent variable and each predictor variable in the training dataset (Step 1 of Figure 3).

2. We select the three predictor variables ($q1$, $q2$ and $q3$) that have the highest Spearman correlation values (the gray shaded variables in Step 2 of Figure 3). We perform this step because we would like to focus on the metrics that have strong relationships with defect-inducing changes.

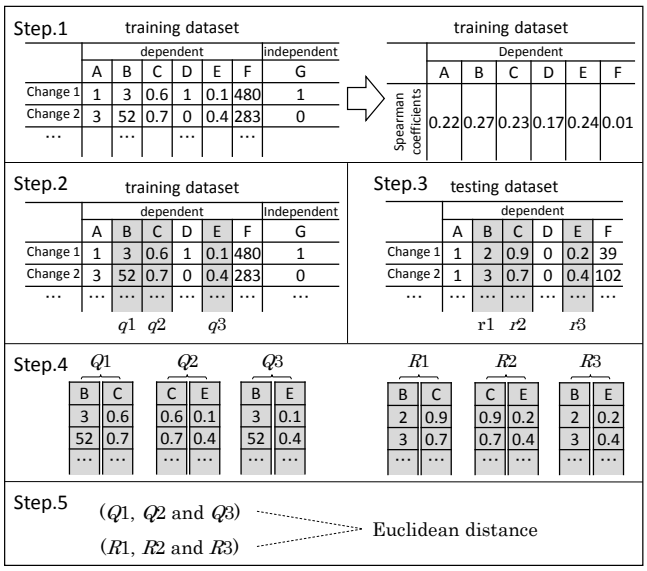

Figure 3: The five steps in the technique for calculating the similarity between two projects.

3. We then select the same three predictor variables ($r1$, $r2$ and $r3$) from testing dataset (the grey shaded variables in Step 3 of Figure 3).

4. We calculate the Spearman correlation between $q1$ and $q2$ ($Q1$), $q2$ and $q3$ ($Q2$), and $q3$ and $q1$ ($Q3$) to obtain a three-dimensional vector ($Q1$, $Q2$, $Q3$). We repeat these steps using the $r1$, $r2$ and $r3$ to obtain another vector ($R1$, $R2$, $R3$) for testing dataset.

5. Finally, we obtain our similarity measure by calculating the Euclidean distance between ($Q1$, $Q2$, $Q3$) and ($R1$, $R2$, $R3$).

In RQ2, we build a prediction model using the most similar project with a testing project based on our similarity metric.

In a prediction scenario, we will not know the value of the dependent variable, since it is what we aim to predict. Hence, our similarity metric does not rely on the dependent variable of the testing dataset.

Results. Figure 4 shows the results of RQ2. Base.RF is used as a baseline, which shows the median AUC values for all cross-project predictors, i.e., off-diagonal elements in Table 3.

Figure 4 shows that all of the models selected using our similarity metric have AUC values over 0.65. Furthermore, RQ2 models tend to outperform Base.RF in terms of median value. A one-tailed Mann-Whitney U test indicates that the difference between Base.RF and RQ2 is statistically significant ($\alpha = 0.05$). These results suggest that our similarity metric helps to identify stable models with strong cross-project prediction performance from a list of candidates.

To understand how well the similarity-based approach works, we check the relationship between similarity ranks and actual ranks. While the similarity ranks are measured by ordering projects using our similarity metric, the actual ranks are measured by ordering projects by the AUC of cross-project prediction. When we use our similarity metric for model selection, the actual top ranked project (i.e., the project to provides the best prediction model) is chosen for 3 of the 11 projects (Columba, Gimp and Platform), the second rank project is chosen for 1 project (Mozilla) and the third rank project is chosen for two projects (Bugzilla and Perl). This result suggests that our similarity metric approach helps to select high performance JIT models.

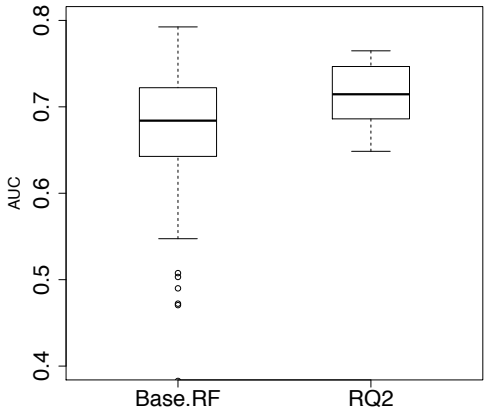

Figure 4: [RQ2] Effect of selecting training data by degree of similarity

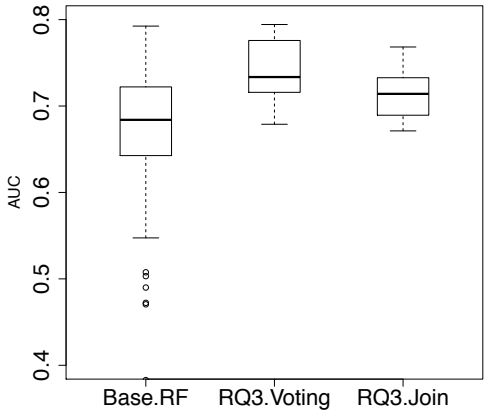

Figure 5: [RQ3] Effect of ensemble learning

Furthermore, we check the impact of the number of predictor variables that are used to calculate the similarity between two projects from 2 to 6 (3 was used in this RQ). The result shows that when we use 2 and 3 as the number of predictor variables to calculate the similarity, the median AUC values of RQ2 models are better than Base.RF. However, use of additional variables in the similarity calculation actually degrades RQ2 model performance. This result suggests that the step 2 of RQ2 models works well.

Similar predictor-dependent variable correlations tend to produce cross-project models that perform well in a cross-project context.

(RQ3) Do ensemble techniques improve cross-project prediction performance?

Since we possess several projects, we suspect that we using all datasets in tandem with each other may produce more powerful cross-project prediction models than ones built us-

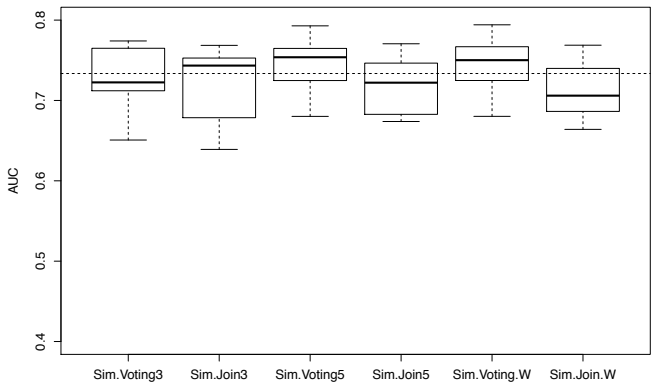

Figure 6: The result of hybrid approaches

ing only one project [21, 32]. We evaluate *voting* and *joining* ensemble approaches that leverage the entire dataset, and hence, divide RQ3 into two parts respectively (RQ3-1 and RQ3-2).

Approach. In RQ3-1, we build separate prediction models using each dataset. To calculate the likelihood of a change being defect-inducing, we push the change through each prediction model and then take the mean of the predicted probabilities.

We illustrate the voting method using an example in the case of Mozilla as the testing project below. First, we build 10 prediction models using each of the other datasets. Given a change from Mozilla project, we obtain 10 predicted probabilities from the 10 models. Finally, we calculate the mean of the 10 probabilities.

In RQ3-2, rather than using each dataset individually, we merge them together to make one dataset. Naturally, we exclude the testing dataset from the merge operation. We then build one prediction model using all of the data in the merged dataset.

Results. Figure 5 shows the results of RQ3-1 and RQ3-2. RQ3.Voting shows the result of the voting method in RQ3-1, while RQ3.Join shows the result of the joining method in RQ3-2.

The results indicate that both RQ3-1 and RQ3-2 outperform the baseline, including our similarity metric approach (RQ2) in terms of the median value. One tailed Mann-Whitney U tests confirm that the improvements are statistically significant ($\alpha = 0.05$). Furthermore, RQ3.Voting tends to perform better than the other cross-project prediction models, including RQ3.Join in terms of all boxplot statistics, i.e., minimum, 25th, 50th and 75th percentiles and maximum values.

Ensemble learning methods tend to produce JIT defect models that perform well in a cross-project context.

Summary

Although we do not find a relationship between strong within-project prediction performance and cross-project prediction performance (RQ1), our results suggest that the JIT prediction models built using projects with similar characteristics (RQ2) or using ensemble methods (RQ3) tend perform well in a cross-project context. The differences between models built using similar projects or ensemble methods indicate performance has improved to a statistically significant degree ($\alpha = 0.05$). Thus, we conclude that the answer to RQ1 is "no" and the answers to RQ2 and RQ3 are "yes".

The median values of RQ2 (Similarity) in Figure 4 and RQ3-1 (Voting) and RQ3-2 (Joining) in Figure 5 are 0.72, 0.73 and 0.71 respectively. We check the difference of the median values among three models using ANOVA and Tukey's HSD, which is a single-step multiple comparison procedure and statistical test [4]. The test results indicate that difference between the three result sets are not statistically significant. Hence, while we do not have evidence to indicate which of the three high performance approaches a practitioner should adopt, we do suggest that practitioners avoid the RQ1 approach.

6. DISCUSSION

6.1 Hybrid Approaches

Since we find that project similarity and ensemble approaches tend to improve the prediction performance of cross-project prediction models, we are interested in analyzing hybrid approaches that combine them. While project similarity provides an approach to select training projects from candidates, ensemble approaches describe how to leverage several datasets to build a more general model.

Approach. For each testing dataset, we use our similarity metric to select several training datasets, and then perform cross-project prediction by applying the voting and joining approaches to combine them. Using this approach, we must select the threshold n, i.e., the number of similar projects to use for training. We evaluate $n = 3$ and $n = 5$ threshold values.

As an alternative to using a threshold, we also evaluate the performance of a weighting approach, which randomly samples $(10 - (r - 1))/10 \times 100$ % of the changes (where r is the project rank based on our similarity metric) for each dataset. For example, 100% of changes are picked up from the most similar project, while 90% of changes are picked up from the second most similar project, and so on.

Results. Figure 6 shows the results of applying our hybrid approaches. The dashed line shows the median value of RQ3.Voting, i.e., the best median value among three high performance models (similarity, voting and joining). We find that either using (1) our similarity metric to select the top five similar projects (Sim.Voting5) or (2) the weighting approach (Sim.Voting.W) tend to provide more powerful prediction models than only using RQ3.Voting. We recommend the use of the weighted approach, since it does not depend on a threshold value.

6.2 What is the impact of model threshold selection for classification?

Throughout our case study, we used AUC to evaluate the performance across thresholds. Yet the AUC does not consider the ease of selecting a good threshold for the model, which one must do in practice. Therefore, in order to make sure our approaches improve the prediction performance in a more practical context, we evaluate the prediction performance using F-measure with a model threshold of 0.5, since 0.5 is a frequently adopted threshold value [7, 12].

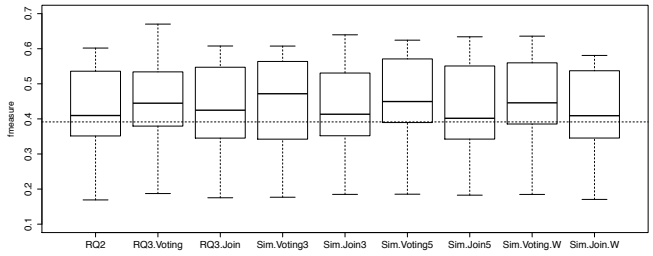

Figure 7: F-measure of all experiments

Results. Figure 7 shows the F-measure of our models built using our hybrid approaches. The dashed line shows the median value of the results of all cross-project predictors (Base.RF), i.e., non-diagonal elements in Table 3. The results show that the median value of F-measure of all approaches outperform the baseline. We, therefore, conclude that the hybrid approaches perform well both in terms of AUC and F-measure.

7. THREATS TO VALIDITY

In this section, we discuss the threats to the validity of our case study.

External validity. We only study 11 open source systems, and hence, our results may not generalize to all software systems. However, we study large, long-lived systems from various domains in order to combat potential bias in our results. Nonetheless, replication of our study using additional systems may prove fruitful.

We use random forest to evaluate the effect of the JIT prediction across projects, since this modeling technique is known to perform well for defect prediction. However, using other modeling techniques may produce different results.

Internal validity. Although we study eight metrics spanning three categories, there are likely other features of defect-inducing changes that we did not measure. For example, we suspect that the type of a change (e.g., refactoring [23, 29]) might influence the likelihood of introducing a defect. We plan to expand our metric set to include additional categories in future work.

We use defect datasets provided by prior work [12] that identify defect-inducing changes using the SZZ algorithm [31]. The SZZ algorithm is commonly used in defect prediction research [14, 23], yet has known limitations. For example, if a defect is not recorded in the VCS commit message or the keywords used defect identifiers differ from those used in the previous study (e.g., "Bug" or "Fix" [11]), such a change will not be tagged as defect-inducing. The use of an approach to recover missing links that improve the accuracy of the SZZ algorithm [34] may improve the accuracy of our results.

8. CONCLUSIONS

In this paper, we study approaches for constructing Just-In-Time (JIT) defect prediction models that identify source code changes that have a high risk of introducing a defect. Since one cannot produce JIT models if insufficient training data is available, e.g., a project does not archive change histories in a VCS repository, we empirically evaluated the use of the datasets collected from other projects (i.e., cross-project prediction). Through a case study on 11 open source projects, we make the following observations:

- Defect models with high within-project performance are rarely high performance cross-project models (RQ1).
- Prediction performance can be improved by selecting datasets for training that are highly similar to the testing dataset (RQ2).
- Several datasets can be used in tandem to produce more accurate models, especially when using the voting method (RQ3).
- Similarity and ensemble methods can be used in tandem with each other to yield even more accurate JIT cross-project models.

Future work. Our results suggest that the ensemble methods yield high performance JIT defect prediction models for cross-project prediction. For example, all models generated using the voting method proposed in RQ3-1 generate models with AUC values over 0.6. Hence, we plan to explore more powerful ensemble methods (e.g., clustering [20, 33]).

TCA is a state-of-the art transfer learning approach and makes feature distributions in training projects and testing projects similar. We will apply the TCA approach to JIT defect prediction models for cross-project prediction to make training and testing projects similar.

9. ACKNOWLEDGMENTS

This research was partially supported by JSPS KAKENHI Grant Numbers 24680003 and 25540026 and the Natural Sciences and Engineering Research Council of Canada (NSERC).

10. REFERENCES

[1] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[2] N. Bettenburg, M. Nagappan, and A. E. Hassan. Think locally, act globally: Improving defect and effort prediction models. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'12)*, pages 60–69, 2012.

[3] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[4] F. L. Coolidge. *Statistics: A Gentle Introduction*. SAGE Publications (3rd ed.), 2012.

[5] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'10)*, pages 31–41, 2010.

[6] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[7] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'10)*, volume 1, pages 495–504, 2010.

[8] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'09)*, pages 78–88, 2009.

[9] Y. Jiang, B. Cukic, and T. Menzies. Can data transformation help in the detection of fault-prone modules? In *Proc. Workshop on Defects in Large Software Systems (DEFECTS'08)*, pages 16–20, 2008.

[10] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort aware models. In *Proc. Int'l Conf. on Software Maintenance (ICSM'10)*, pages 1–10, 2010.

[11] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K.-i. Matsumoto. The effects of over and under sampling on fault-prone module detection. In *Proc. Int'l Symposium on Empirical Softw. Eng. and Measurement (ESEM'07)*, pages 196–204, 2007.

[12] 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 Trans. Softw. Eng.*, 39(6):757–773, 2013.

[13] T. M. Khoshgoftaar and E. B. Allen. Modeling software quality with classification trees. *Recent Advances in Reliability and Quality Engineering*, 2:247–270, 2001.

[14] S. Kim, E. J. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34(2):181–196, 2008.

[15] E. Kocaguneli, T. Menzies, and J. Keung. On the value of ensemble effort estimation. *IEEE Trans. Softw. Eng.*, 38(6):1403–1416, 2012.

[16] A. G. Koru, D. Zhang, K. El Emam, and H. Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Trans. Softw. Eng.*, 35(2):293–304, 2009.

[17] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Softw. Eng.*, 34(4):485–496, July 2008.

[18] P. L. Li, J. Herbsleb, M. Shaw, and B. Robinson. Experiences and results from initiating field defect prediction and product test prioritization efforts at ABB Inc. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'06)*, pages 413–422, 2006.

[19] S. Matsumoto, Y. Kamei, A. Monden, and K. Matsumoto. An analysis of developer metrics for fault prediction. In *Proc. Int'l Conf. on Predictive Models in Softw. Eng. (PROMISE'10)*, pages 18:1–18:9, 2010.

[20] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Trans. Softw. Eng.*, 39(6):822–834, 2013.

[21] A. T. Mısırlı, A. B. Bener, and B. Turhan. An industrial case study of classifier ensembles for locating software defects. *Software Quality Journal*, 19(3):515–536, 2011.

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

[23] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'08)*, pages 181–190, 2008.

[24] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'05)*, pages 284–292, 2005.

[25] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'06)*, pages 452–461, 2006.

[26] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'13)*, pages 382–391, 2013.

[27] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Trans. Softw. Eng.*, 22(12):886–894, 1996.

[28] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Trans. Softw. Eng.*, 31(6):511–526, 2005.

[29] J. Ratzinger, T. Sigmund, and H. C. Gall. On the relation of refactorings and software defect prediction. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'08)*, pages 35–38, 2008.

[30] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang. An industrial study on the risk of software changes. In *Proc. European Softw. Eng. Conf. and Symposium on the Foundations of Softw. Eng. (ESEC/FSE'12)*, pages 62:1–62:11, 2012.

[31] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'05)*, pages 1–5, 2005.

[32] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan. The impact of classifier configuration and classifier combination on bug localization. *IEEE Trans. Softw. Eng.*, 39(10):1427–1443, 2013.

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

[34] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proc. European Softw. Eng. Conf. and Symposium on the Foundations of Softw. Eng. (ESEC/FSE'11)*, pages 15–25, 2011.

[35] F. Xing, P. Guo, and M. R. Lyu. A novel method for early software quality prediction based on support vector machine. In *Proc. Int'l Symposium on Software Reliability Engineering (ISSRE'05)*, pages 10–pp, 2005.

[36] 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 *Proc. European Softw. Eng. Conf. and Symposium on the Foundations of Softw. Eng. (ESEC/FSE'09)*, pages 91–100, 2009.