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

Studying Just-In-Time Defect Prediction using Cross-Project Models

Yasutaka Kamei · Takafumi Fukushima · Shane McIntosh · Kazuhiro Yamashita · Naoyasu Ubayashi · Ahmed E. Hassan

Author pre-print copy. The final publication is available at Springer via:
`http://dx.doi.org/10.1007/s10664-015-9400-x`

Abstract Unlike traditional defect prediction models that identify defect-prone modules, Just-In-Time (JIT) defect prediction models identify defect-inducing changes. As such, JIT defect models can provide earlier feedback for developers, while design decisions are still fresh in their minds. Unfortunately, similar to traditional defect models, JIT models require a large amount of training data, which is not available when projects are in initial development phases. To address this limitation in traditional defect prediction, prior work has proposed cross-project models, i.e., models learned from other 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 models in a cross-project context. Through an empirical study on 11 open source projects, we find that while JIT models rarely perform well in a cross-project context, their performance tends to improve when using approaches that: (1) select models trained using other projects that are similar to the testing project, (2) combine the data of several other projects to produce a larger pool of training data, and (3) combine the models of several other projects to produce an ensemble model. Our findings empirically confirm that JIT models learned using other projects are a viable solution for projects with limited historical

Yasutaka Kamei · Takafumi Fukushima · Kazuhiro Yamashita · Naoyasu Ubayashi
Principles of Software Languages Group (POSL)
Kyushu University, Japan
E-mail: kamei@ait.kyushu-u.ac.jp, ubayashi@ait.kyushu-u.ac.jp
E-mail: f.taka@posl.ait.kyushu-u.ac.jp, yamashita@posl.ait.kyushu-u.ac.jp

Shane McIntosh
Department of Electrical and Computer Engineering
McGill University, Canada
E-mail: shane.mcintosh@mcgill.ca

Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University, Canada
E-mail: ahmed@cs.queensu.ca

data. However, JIT models tend to perform best in a cross-project context when the data used to learn them are carefully selected.

Keywords Empirical study · Defect prediction · Just-In-Time prediction

1 Introduction

Software Quality Assurance (SQA) activities, such as code inspection and unit testing are standard practices for improving the quality of a software system prior to its official release. However, software teams have limited testing resources, and must wisely allocate them to minimize the risk of incurring *post-release defects*. For this reason, a plethora of software engineering research is focused on prioritizing SQA activities (Li *et al.*, 2006; Shihab, 2012). 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 (Basili *et al.*, 1996; Li *et al.*, 2006). Using these techniques, practitioners can allocate limited SQA resources to the most defect-prone modules.

However, recent work shows that traditional defect prediction models often make recommendations at a granularity that is too coarse to be applied in practice (Kamei *et al.*, 2010, 2013; Shihab *et al.*, 2012). For example, since the largest files or packages are often the most defect-prone (Koru *et al.*, 2009), 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 when the code was initially produced may be difficult 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 (Kim *et al.*, 2008).

To address these limitations 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 (Kamei *et al.*, 2013; Kim *et al.*, 2008; Mockus and Weiss, 2000; Shihab *et al.*, 2012; Śliwerski *et al.*, 2005). The advantages of change-level predictions are that: (1) the predictions are made at a fine granularity, since changes often impact only a small region 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. Change-level defect prediction has been successfully adopted by industrial software teams at Avaya (Mockus and Weiss, 2000), BlackBerry (Shihab *et al.*, 2012), and Cisco (Tan *et al.*, 2015). We refer to change-level defect prediction as “Just-In-Time (JIT) defect prediction” (Kamei *et al.*, 2013).

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 (Zimmermann *et al.*, 2009). 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 the limited availability of training data, prior work has proposed *cross-project defect prediction models*, i.e., models trained using historical data from other projects (Turhan *et al.*, 2009).

While studies have shown that cross-project defect prediction models can perform well at the file-level (Bettenburg *et al.*, 2012; Menzies *et al.*, 2013), cross-project JIT models remain largely unexplored. We, therefore, set out to empirically study the performance of JIT models in a cross-project context using data from 11 open source projects. We find that the within-project performance of a JIT model does not indicate how well it will perform in a cross-project context (Section 4). Hence, we set out to study three approaches to optimize the performance of JIT models in a cross-project context. We structure our study along the following three research questions:

(RQ1) Do JIT models selected using project similarity perform well in a cross-project context? (Model selection)
Defect prediction models assume that the distributions of the metrics in the training and testing datasets are similar (Turhan *et al.*, 2009). 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 the performance of cross-project models would suffer. On the other hand, we expect that models trained using data from similar projects will have strong cross-project performance.

(RQ2) Do JIT models built using a pool of data from several projects perform well in a cross-project context? (Data merging)
A model that was fit using data from only one project may be *overfit*, i.e., too closely related to the training data to apply to other datasets. Conversely, sampling from a more diverse pool of changes from several other projects may provide a more robust model fit that will apply better in a cross-project context. Hence, we want to investigate whether the cross-project performance of JIT models improve when we train them using changes from a variety of other projects.

(RQ3) Do ensembles of JIT models built from several projects perform well in a cross-project context? (Ensembles of models)
Since ensemble classification techniques have recently proven useful in other areas of software engineering (Kocaguneli *et al.*, 2012), we suspect that they may also improve the cross-project performance of JIT models. Ensemble techniques that leverage multiple datasets cover a large variety of project characteristics, and hence may provide a more general JIT model for cross-project prediction, i.e., not only those of one project.

Through an empirical study on 11 open source projects, we find that the most similar projects yield JIT models that are among the 3 top-performing cross-project models for 6 of the 11 studied systems (RQ1). Although, while similarity helps to select the top-performing models, these models tend to under-perform with respect to within-project performance. On the other hand, combining data from (RQ2) and models trained using (RQ3) several other projects tends to yield JIT models that have strong cross-project performance, which is indistinguishable from within-project performance. However, when we use similarity to filter away dissimilar project data, it rarely improves model performance. This suggests that additional training data is a more important factor for cross-project JIT models than project similarity is.

This paper is an extended version of our earlier work (Fukushima *et al.*, 2014). We extend our previous work by:

- Studying domain-aware similarity techniques (RQ1) to combat limitations in our threshold-dependent, domain-agnostic similarity approach.
- Studying context-aware rank transformation as a means of using data from several projects simultaneously (RQ2).
- Grounding the cross-project performance of JIT models by normalizing them by the performance of the corresponding within-project JIT model.

1.1 Paper Organization

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 describes the setting of our empirical study. Section 4 describes a preliminary study of the relationship between the within-project and cross-project performance of JIT models, while Section 5 presents the results of our empirical study with respect to our three research questions. Section 6 discusses the broader implications of our findings. 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 JIT and cross-project defect prediction.

2.1 Just-In-Time Defect Prediction

A traditional defect model classifies each module as either defective or not using module metrics (e.g., SLOC and McCabe's Cyclomatic complexity) as predictor variables. On the other hand, JIT models use change metrics (e.g., # modified files) to explain the status of a change (i.e., defect-inducing or not).

Prior work suggests that JIT prediction is a more practical alternative to traditional defect prediction. For example, Mockus and Weiss (2000) predict defect-inducing changes in a large-scale telecommunication system. Kim *et al.* (2008) 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.* (2013) 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, but also that it is viable, yielding actionable results. However, defect models must be trained using a large corpus of data in order to perform well (Zimmermann *et al.*, 2009). Since new projects and legacy ones may not have enough historical data available to train effective JIT models, we set out to study JIT models in a cross-project context.

2.1.1 Building JIT Models

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 (Hall *et al.*, 2012), such as linear discriminant analysis, decision trees, Naive Bayes and Support Vector Machines (SVM).

Random forest. In this paper, we train our JIT 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 (Jiang *et al.*, 2008). Furthermore, our prior studies have shown that random forest tends to outperform other modeling techniques for defect prediction (Kamei *et al.*, 2010).

Random forest is a classification (or regression) technique that builds a large number of decision trees at training time (Breiman, 2001). 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 (Breiman, 2001).

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 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 (Menzies *et al.*, 2013; Minku and Yao, 2014; Nam *et al.*, 2013; Turhan *et al.*, 2009; Zhang *et al.*, 2014; Zimmermann *et al.*, 2009).

Briand *et al.* (2002) train a defect prediction model using data from one Java system, and test it using data from another Java system, reporting lower prediction performance for the cross-project context than the within-project one. On the other hand, Turhan *et al.* (2009) find that cross-project prediction models can actually outperform models built using within-project data. However, Turhan *et al.* (2011) also find that adding mixed project data to an existing prediction model yields only minor improvements to prediction performance.

Zimmermann *et al.* (2009) 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.

Rahman *et al.* (2012) evaluate the prediction performance of cross-project models by taking into account the cost of software quality assurance effort. They show that using such a perspective, the performance of cross-project models is comparable to that of within-project models. He *et al.* (2012) show that cross-project models outperform within-project models if it is possible to pick the best cross-project models among all available models to predict testing projects.

Menzies *et al.* (2011, 2013) 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.* (2013) 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.

Recent work has shown that cross-project models can achieve performance similar to that of within-project models. Zhang *et al.* (2014) proposes a context-aware rank transformation method to preprocess predictors and address the variations in their distributions. Using 1,398 open source projects, they produce a “universal” defect prediction model that achieves performance that rivals within-project models. Minku and Yao (2014) investigate how to make best use of cross-project data in the domain of software effort estimation. Through use of a proposed framework to map metrics from one context to another, their cross-project effort estimation models achieve performance similar to within-project ones.

Similar to prior work, we find that cross-project models that use a combination of ensemble and similarity techniques can outperform within-project models. Furthermore, while prior studies have empirically evaluated cross-project prediction performance using traditional models, our study focuses on cross-project prediction using JIT models.

3 Experimental Setting

3.1 Studied Systems

In order to address our 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.* (2013) 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.

3.2 Change Measures

Our 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 (Kamei *et al.*, 2013). Table 2 provides a brief description of each metric and the rationale behind using it in JIT models. We remove 6 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). We briefly describe each surviving metric below.

3.2.1 Identify Defect-Inducing Changes

To know whether or not a change introduces a defect, we used the SZZ algorithm (Śliwerski *et al.*, 2005). This algorithm identifies when a bug was injected into the code and who injected it using a VCS. We discuss the noise introduced by the heuristic nature of the SZZ algorithm in Section 7.2.

3.2.2 Diffusion

We expect that the diffusion dimension can be leveraged to determine the likelihood of a change being defect-inducing. We use four diffusion metrics in our JIT models, as listed in Table 2.

Prior work has shown that a highly distributed change can be more complex and harder to understand (Hassan, 2009; Mockus and Weiss, 2000). For example, Mockus and Weiss (2000) have shown that the number of changed subsystems is related to defect-proneness. Hassan (2009) has shown that change entropy is a more powerful predictors of the incidence of defects than the number of prior defects or changes. In our study, similar to Hassan (2009), we normalize the change entropy by the maximum entropy $log_2 n$ to account for differences in the number of files n across changes.

For each change, we count the number of distinct names of modified: (1) subsystems (NS, i.e., root directories), (2) directories (ND) and (3) 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/org/eclipse/jdt/core/dom` and the file name is `org.eclipse.jdt.core/jdom/org/eclipse/jdt/core/dom/Node.java`.

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

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

Table 2 Summary of change measures (Kamei *et al.*, 2013).

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 (Mockus and Weiss, 2000).
	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 (Mockus and Weiss, 2000).
	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 (Nagappan *et al.*, 2006)
	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 (D'Ambros *et al.*, 2010; Hassan, 2009).
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 (Moser *et al.*, 2008; Nagappan and Ball, 2005).
	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 (Koru *et al.*, 2009).
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 (Guo *et al.*, 2010)(Purushothaman and Perry, 2005).
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 (Matsumoto *et al.*, 2010).
	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 (Graves *et al.*, 2000).
	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 (D'Ambros *et al.*, 2010; Hassan, 2009).
Experience*	EXP	Developer experience	More experienced developers are less likely to introduce a defect.	Programmer experience significantly reduces the likelihood of introducing a defect (Mockus and Weiss, 2000). 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.**

3.2.3 Size

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 (Moser *et al.*, 2008; Nagappan and Ball, 2005). Hence, we use the size dimension to identify defect-inducing changes. We use the lines added (LA), lines deleted (LD), and lines total (LT) metrics to measure change size as shown in Table 2. We normalize LT by dividing it by NF (relative LT), similarly to Kamei *et al.* (2013). These metrics can be extracted directly from a VCS.

Table 3 The median of Spearman correlation values among dataset.

	NF	Entropy	Relative churn	Relative LT	Fix
NS	0.21	0.11	0.03	-0.05	-0.04
NF	-	0.72	0.17	-0.02	-0.14
Entropy	-	-	0.40	0.16	0.01
Relative churn	-	-	-	0.18	0.08
Relative LT	-	-	-	-	0.22

3.2.4 Purpose

A change that fixes a defect is more likely to introduce another defect (Guo *et al.*, 2010; Purushothaman and Perry, 2005). The intuition being that the defect-prone modules of the past tend to remain defect-prone in the future (Graves *et al.*, 2000).

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 (Kamei *et al.*, 2013; Kim *et al.*, 2008).

3.3 Data Preparation

3.3.1 Minimizing Collinearity

To combat the threat of multicollinearity in our models, 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 below each metrics that we discarded.

We found that NS and ND are highly correlated ($\rho = 0.84$). To address this, we exclude ND and include NS in our prediction models. We also found that LA and LD are highly correlated ($\rho = 0.89$). Nagappan and Ball (2005) 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. In short, the NS, NF, Entropy, relative churn (i.e., (LA+LD)/LT), relative LT (= LT/NF) and FIX metrics survive our correlation analysis (Table 3). Table 4 and 5 provide descriptive statistics of the six studied metrics.

3.3.2 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 (Kamei *et al.*, 2007). 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

Table 4 Descriptive statistics of the studied metrics (1/2)

		NS	NF	Entropy	Relative churn	Relative LT	Fix
BUG	Minimum	1.000	1.000	0.000	0.000	0.000	–
	1st Quartile	1.000	1.000	0.000	0.006	210.000	–
	Median	1.000	1.000	0.000	0.017	455.000	–
	Mean	1.170	2.288	0.229	0.103	591.400	0.860
	3rd Quatile	1.000	2.000	0.551	0.057	799.200	–
	Maximum	4.000	63.000	1.000	21.000	2751.000	–
COL	Minimum	1.000	1.000	0.000	0.000	0.000	–
	1st Quartile	1.000	1.000	0.000	0.007	38.000	–
	Median	1.000	2.000	0.000	0.093	77.000	–
	Mean	1.034	6.195	0.277	0.430	114.200	0.328
	3rd Quatile	1.000	4.000	0.667	0.384	150.000	–
	Maximum	6.000	1297.000	1.000	8.667	1371.000	–
GIP	Minimum	0.000	0.000	0.000	0.000	0.000	–
	1st Quartile	1.000	2.000	0.021	0.001	787.000	–
	Median	2.000	2.000	0.625	0.007	2524.000	–
	Mean	1.873	6.737	0.513	0.276	5385.000	0.165
	3rd Quatile	2.000	5.000	0.863	0.035	7422.000	–
	Maximum	39.000	2730.000	1.000	2877.000	74172.000	–
JDT	Minimum	1.000	1.000	0.000	0.000	0.000	–
	1st Quartile	1.000	1.000	0.000	0.011	105.000	–
	Median	1.000	1.000	0.000	0.039	238.100	–
	Mean	1.011	3.874	0.269	0.167	437.700	0.305
	3rd Quatile	1.000	2.000	0.670	0.125	496.000	–
	Maximum	4.000	1645.000	1.000	264.000	7140.000	–
MAV	Minimum	0.000	0.000	0.000	0.000	0.000	–
	1st Quartile	1.000	1.000	0.000	0.013	51.000	–
	Median	1.000	1.000	0.000	0.052	156.300	–
	Mean	1.691	4.386	0.314	0.399	313.700	0.150
	3rd Quatile	1.000	3.000	0.753	0.175	376.900	–
	Maximum	32.000	732.000	1.000	295.767	3994.000	–
MOZ	Minimum	1.000	1.000	0.000	0.000	0.000	–
	1st Quartile	1.000	1.000	0.000	0.004	170.000	–
	Median	1.000	1.000	0.000	0.016	521.000	–
	Mean	1.199	3.705	0.307	0.136	970.600	0.640
	3rd Quatile	1.000	3.000	0.722	0.064	1269.000	–
	Maximum	30.000	2817.000	1.000	170.733	38980.000	–

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.

3.4 Evaluating Model Performance

To evaluate model prediction performance, precision, recall and F-measure are often used (Kim *et al.*, 2008; Nam *et al.*, 2013). However, as Lessmann *et al.* (2008) point out, these criteria depend on the threshold that is used for classification. Choosing a different threshold may lead to different results.

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.,

Table 5 Descriptive statistics of the studied metrics (2/2)

		NS	NF	Entropy	Relative churn	Relative LT	Fix
PER	Minimum	0.000	0.000	0.000	0.000	0.000	–
	1st Quartile	1.000	1.000	0.000	0.001	318.000	–
	Median	1.000	1.000	0.000	0.005	1196.000	–
	Mean	1.851	3.350	0.271	0.079	2772.000	0.201
	3rd Quatile	2.000	2.000	0.669	0.023	3379.000	–
	Maximum	183.000	974.000	1.000	196.438	92171.000	–
PLA	Minimum	1.000	1.000	0.000	0.000	0.000	–
	1st Quartile	1.000	1.000	0.000	0.007	62.000	–
	Median	1.000	1.000	0.000	0.040	169.000	–
	Mean	1.058	3.763	0.274	0.230	354.700	0.400
	3rd Quatile	1.000	2.000	0.678	0.152	410.000	–
	Maximum	18.000	1569.000	1.000	289.000	8744.000	–
POS	Minimum	1.000	1.000	0.000	0.000	0.000	–
	1st Quartile	1.000	1.000	0.000	0.009	254.000	–
	Median	1.000	1.000	0.000	0.026	567.000	–
	Mean	1.301	4.461	0.282	0.101	853.300	0.437
	3rd Quatile	1.000	3.000	0.689	0.075	1136.200	–
	Maximum	11.000	990.000	1.000	62.567	11326.000	–
RUB	Minimum	0.000	0.000	0.000	0.000	0.000	–
	1st Quartile	1.000	1.000	0.000	0.004	60.000	–
	Median	1.000	1.000	0.000	0.016	231.000	–
	Mean	1.126	2.778	0.369	0.358	486.700	0.192
	3rd Quatile	1.000	3.000	0.813	0.060	626.000	–
	Maximum	10.000	547.000	1.000	1397.385	13060.000	–
RHI	Minimum	1.000	1.000	0.000	0.000	0.000	–
	1st Quartile	1.000	1.000	0.000	0.005	217.800	–
	Median	1.000	2.000	0.047	0.019	764.000	–
	Mean	1.305	3.575	0.402	0.266	1134.300	0.431
	3rd Quatile	2.000	3.000	0.872	0.064	1749.100	–
	Maximum	12.000	434.000	1.000	431.012	6565.000	–

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 guessing.

4 Preliminary Study of Within-Project Performance

Models that perform well on data within the project have established a strong link between predictors and defect-proneness within that project. We suspect that properties of the relationship may still hold if the model is tested on another project.

4.1 Approach

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

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 values. 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*.

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). Finally, we compare within-project and cross-project AUC values.

4.2 Results

Table 6 shows the AUC values that we obtained. Each row shows the projects that we used for testing and each column shows the projects that we used for training. Diag-

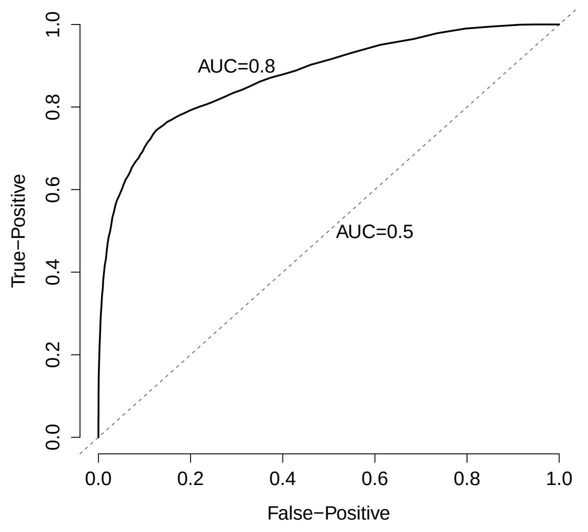


Fig. 1 An example of ROC curve in the case of AUC=0.8 and AUC=0.5.

Table 6 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

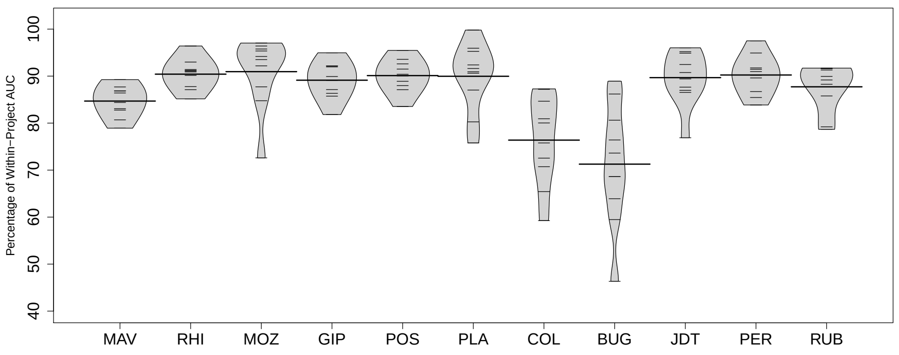


Fig. 2 Cross-project performance of models trained on each project. Projects are sorted by within-project performance along the x-axis. Y-axis shows the cross-project performance (non-gray cells in Table 6) normalized by the performance of the within-project model.

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

Figure 2 shows the cross-project performance (non-gray cells in Table 6) normalized by the performance of the within-project model using beanplots (Kampstra, 2008). Beanplots are boxplots in which the vertical curves summarize the distribution of the dataset. The solid horizontal lines indicate the median value.

The beanplots are sorted in descending order along the X-axis according to the AUC value of within-project prediction. If there were truly a relationship between good within-project and cross-project prediction, one would expect that the beanplots should also descend in value from left to right. Since no such pattern emerges, it seems that there is no relationship between the within-project and cross-project performance of a JIT model. We validate our observation statistically using Spearman correlation tests. We calculate the Spearman correlation between the rank of the AUC value of within-project prediction and the median of the AUC values of cross-project prediction. The resulting value is $\rho = 0.036$ ($p = 0.924$).

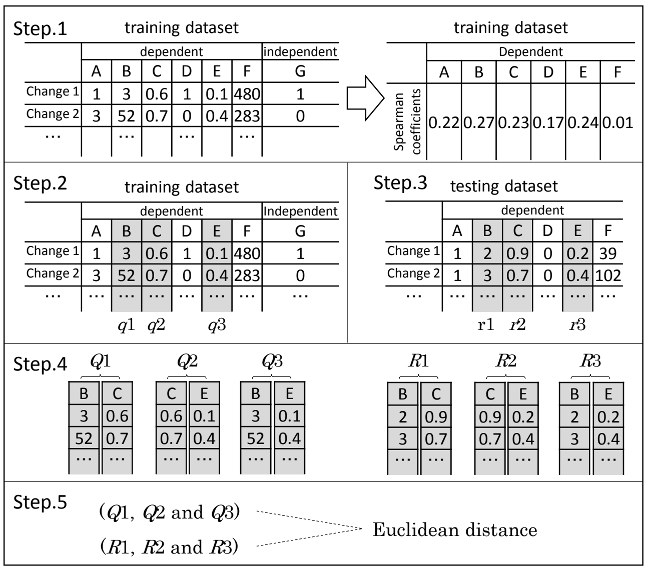
The within-project performance of a JIT model is not a strong indicator of its performance in a cross-project context.

5 Empirical Study Results

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

(RQ1) Do JIT models selected using project similarity perform well in a cross-project context?

We explore domain-agnostic (RQ1-1) and domain-aware (RQ1-2) types of similarity between the studied projects. Domain-agnostic similarity is measured using predictor metric values, whereas domain-aware similarity is measured using project details. We discuss our results with respect to each style of similarity below.


Step.1 training dataset

	dependent						independent
	A	B	C	D	E	F	G
Change 1	1	3	0.6	1	0.1	480	1
Change 2	3	52	0.7	0	0.4	283	0
…		…		…		…	

training dataset

	Dependent					
	A	B	C	D	E	F
Spearman coefficients	0.22	0.27	0.23	0.17	0.24	0.01

Step.2 training dataset

	dependent						Independent
	A	B	C	D	E	F	G
Change 1	1	3	0.6	1	0.1	480	1
Change 2	3	52	0.7	0	0.4	283	0
…	…	…	…	…	…	…	…
		$q1$	$q2$		$q3$		

Step.3 testing dataset

	dependent					
	A	B	C	D	E	F
Change 1	1	2	0.9	0	0.2	39
Change 2	1	3	0.7	0	0.4	102
…	…	…	…	…	…	…
		r1	$r2$		$r3$	

Step.4

$Q1$

B	C
3	0.6
52	0.7
…	…

$Q2$

C	E
0.6	0.1
0.7	0.4
…	…

$Q3$

B	E
3	0.1
52	0.4
…	…

$R1$

B	C
2	0.9
3	0.7
…	…

$R2$

C	E
0.9	0.2
0.7	0.4
…	…

$R3$

B	E
2	0.2
3	0.4
…	…

Step.5

($Q1$, $Q2$ and $Q3$)
($R1$, $R2$ and $R3$)
Euclidean distance

Fig. 3 The five steps in the technique for calculating the domain-agnostic similarity between two projects.

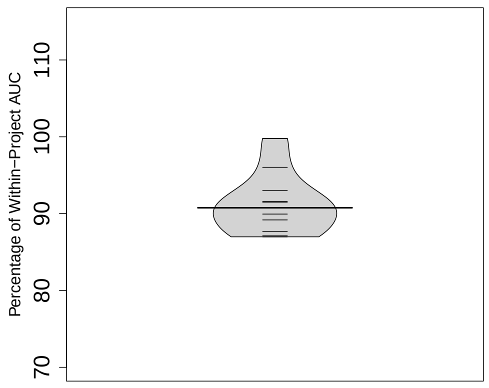


Fig. 4 [RQ1-1] Effect of selecting training data by degree of domain-agnostic similarity.

(RQ1-1) Approach

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

1. We calculate the 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.
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 RQ1-1, we select the prediction model of the most similar project with a testing project. 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.

(RQ1-1) Results

Figure 4 shows the percentage of within-project performance that the model for the most similar project achieves in a cross-project context. We achieve a minimum of 88% of the within-project performance by selecting similar cross-project models. These results suggest that our domain-agnostic similarity metric helps to identify stable JIT models with strong cross-project prediction performance from a list of candidates.

To further analyze how well our domain-agnostic similarity approach is working, we check the relationship between similarity ranks and actual ranks. While the similarity ranks are measured by ordering projects using our domain-agnostic similarity metric, the actual ranks are measured by ordering projects according to the AUC of cross-project prediction (normalized by within-project AUC). When we use our domain-agnostic similarity metric for model selection, the actual top-ranked project (i.e., the cross-project model that performs the best for this project) is chosen for 3 of the 11 studied projects (Columba, Gimp and Platform), the second ranked project is chosen for 1 studied project (Mozilla) and the third ranked project is chosen for 2 studied projects (Bugzilla and Perl). Altogether, 8 projects perform better than the sixth (median) rank. This result suggests that our domain-agnostic similarity metric helps to select top-performing JIT models for cross-project prediction.

On the other hand, while our domain-agnostic results are promising, there are many thresholds that must be carefully selected. A threshold analysis reveals that the models selected by our domain-agnostic similarity perform best when a small number of predictor variables (4 or fewer) are used. When we use too many variables (i.e., more than 4) in the domain-agnostic similarity calculation, the models identified as highly similar tend to perform worse.

While our domain-agnostic similarity metric selects JIT models that tend to perform better than the median cross-project performance, the metric is sensitive to threshold values.

(RQ1-2) Approach

In addition to the threshold-sensitivity of our domain-agnostic similarity metric, one would also require access to the data of all other projects in order to calculate it. This may not be practical in a cross-company context due to privacy concerns of the companies involved. To avoid such limitations, we set out to study privacy-preserving means of selecting similar projects.

Previous work has also explored the use of similarity to select models that will likely perform well in a cross-project context. For example, Zimmermann *et al.* (2009) propose the following project-based metrics (i.e., context factors) to calculate similarity:

Company (Mozilla Corp/Eclipse/Apache foundation/Others): the organization responsible for developing a system.

Intended audience (End user/Developer): whether a system is built for interaction with end users (e.g., Mozilla) or built for development professionals (e.g., Ruby on Rails).

User interface (Graphical/Toolkit/Non-interactive): The type of system. For example, Gimp has a GUI, while Maven-2 is a toolkit and Perl is non-interactive.

Product uses database (Yes/No): whether or not a system persists data using a database.

Programming language (Java/JavaScript/C/C++/Perl/Ruby): programming language used in the system development. We identify the programming language using Linguist.[1] Similar to other work (McIntosh *et al.*, 2014), if there are more than 10% of files that are written in a programing language, then that programming languages is considered used in the system development.

We refer to this style of similarity as "domain-aware", since it focuses on characteristics of software projects, rather than the data that is produced. In RQ1-2, we study whether these domain-aware characteristics can also help to select JIT models that will perform well in a cross-project context.

Due to differences in our experimental settings, we could not adopt all of the domain-aware metrics of Zimmermann *et al.* (2009). For example, we do not use "open source or not" and "global development or not", since we only study open source projects with globally distributed development teams.

We calculate the Euclidean distance between each project using the domain-aware project characteristics described above. A categorical variable (e.g., company) is transformed into dummy variables (e.g. if the variable has n categories, it is transformed into $n-1$ dummy variables). For example, the company column is transformed into Mozilla (Yes:1 or No:0), Eclipse (Yes:1 or No:0) and Apache (Yes:1 or No:0). If a project is an Eclipse subproject like JDT, each column is 0, 1, 0. If a project is in others line, then each column is 0, 0, 0. Similar to RQ1-1, we use our domain-aware similarity score to select the model with the project that is most similar to the testing project.

(RQ1-2) Results

Figure 5 shows the result of the impact of metrics for calculating project similarity. For comparison, we also show the beanplot of the domain-agnostic similarity metric from Figure 4.

The results indicate that, although the beanplot of domain-aware similarity covers a broader range than the domain-agnostic one, its median values is higher. Similar to our domain-agnostic approach, we analyze the relationship between similarity ranks and actual ranks. The actual top-ranked project (i.e., the project whose model performs the best) is chosen for 3 of the 11 studied projects (Maven-2, Platform and Rhino), the second ranked project is chosen for 2 studied projects (Columba and Eclipse JDT) and the third ranked project is chosen for 1 studied project (Ruby on Rails). Altogether, the model selected by our domain-aware similarity metric is in the top 3 ranks according to actual model performance in 6 of the 11 studied projects, and above the sixth (median) rank in 7 of the 11 studied projects. This result suggests that

[1] `https://github.com/github/linguist`

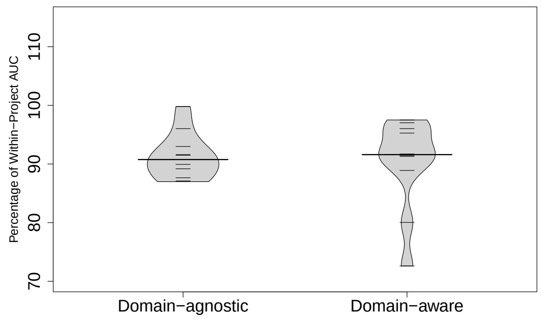


Fig. 5 The impact of metrics for calculating domain-aware similarity.

the domain-aware metrics also help to select JIT models that tend to perform well in a cross-project context.

Domain-aware similarity measures are preferred over domain-agnostic ones, since the domain-agnostic similarity is highly sensitive to threshold selection, while domain-aware similarity is not. Furthermore, we find that models selected using either similarity technique have similar cross-project performance.

(RQ2) Do JIT models built using a pool of data from several projects perform well in a cross-project context?

A model that was fit using data from only one project may be *overfit*, i.e., too specialized for the project from which the model was fit that it would not apply to other projects. It would be useful to use the data of several projects from the entire set of diverse projects to build a JIT model, since such diversity would likely improve the robustness of the model (Turhan *et al.*, 2009; Zhang *et al.*, 2014).

We evaluate three approaches that leverage the entire pool of training datasets (RQ2-1, RQ2-2 and RQ2-3). First, we simply merge the datasets of all of the other projects into a single pool of data, and use it to train a single model (RQ2-1). Next, we train a model using a dataset assembled by drawing more instances from similar projects (RQ2-2). Finally, we transform the data of each project using a rank transformation as proposed by Zhang *et al.* (2014) for cross-project models (RQ2-3).

(RQ2-1) Approach

In RQ2-1, we set aside one project for testing and merge the datasets of the other projects together to make one large training dataset. Next, we train one model using the merged training dataset. Finally, we test the model using the dataset we left out of the merge operation.

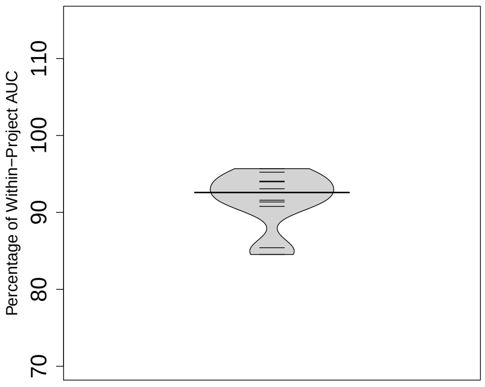


Fig. 6 [RQ2-1] The result of a dataset merging approach.

(RQ2-1) Results

Figure 6 shows the results of our simple merging technique. The models trained using these large datasets yields strong cross-project performance ranging between 85%-96% of the within-project AUC. These results suggest that even a simple approach to combine the data of multiple projects yields JIT models that perform well in a cross-project context.

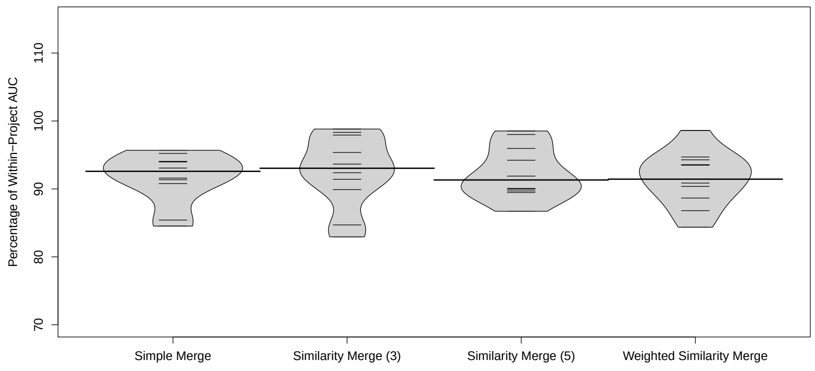
(RQ2-2) Approach

While JIT models trained using a combined pool of data from the other studied projects yields models that perform nearly as well as within-project ones, our result from RQ1 suggests that datasets from similar projects may prove more useful than others. Therefore, we set out to evaluate two approaches to apply the similarity concept to our merging of training datasets:

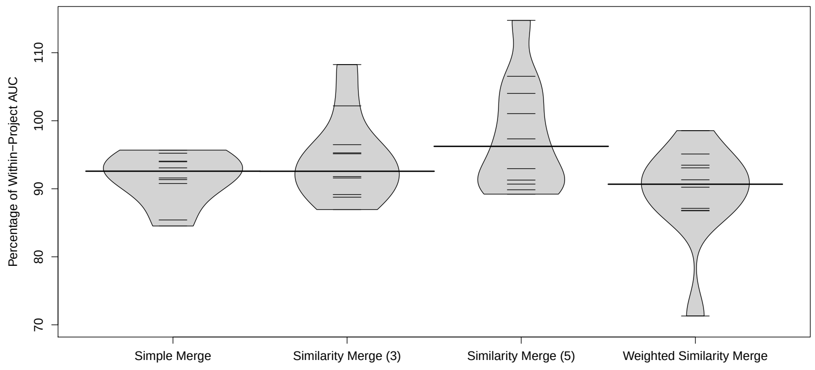
1. For each testing dataset, we use our metrics from RQ1 to select similar datasets to be merged into a larger training dataset. We select the top *n* most similar projects for the training dataset.
2. As a threshold-independent alternative to the above approach, we randomly sample $\frac{(10-(r-1))}{10} \times 100$ % of the changes from each training dataset, where *r* is the project rank based on our similarity metric. 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.

(RQ2-2) Results

Figure 7 shows the results of applying our similarity-inspired approaches to merging training datasets. Similarity Merge (3) and Similarity Merge (5) show the results of



(a) Domain-agnostic



(b) Domain-aware

Fig. 7 [RQ2-2] The results of our dataset merging approaches that leverage project similarity.

using a threshold of 3 and 5 projects respectively, while Weighted Similarity Merge shows the result of a weighing approach. To reduce clutter, we only show the Similarity Merge with threshold values of 3 and 5. However, we provide online access to the figure showing all of the possible threshold values, i.e., between 2 and 10.[2]

When focusing on domain-agnostic similarity, Figure 7a shows that Similarity Merge (3) slightly outperforms the other RQ2 models (i.e., Simple Merge, Similarity Merge (3), Similarity Merge (5) and Weighted Similarity Merge) in terms of the median value. However, the range of the beanplot in Similarity Merge (3) is slightly broader than the other three approaches. We check the difference of the median values among the four models using Tukey's HSD, which is a single-step multiple comparison procedure and statistical test (Coolidge, 2012). The test results indicate that the difference between the four result sets are not statistically significant ($p = 0.941$).

[2] `http://posl.ait.kyushu-u.ac.jp/Disclosure/emse_jit.html`

Similarly, when we focus on domain-aware similarity, Figure 7b shows that Similarity Merge (5) is the strongest performer. Indeed, the Similarity Merge (5) models even outperform the within-project models of Columba, Mozilla, PostgreSQL and Ruby on Rails ($>$ 100% in Figure 7b). On the other hand, the performance of the Weighted Similarity Merge models appear lower due to its poor performance on the Rhino project where it only achieved 71% of the within-project AUC. Table 6 shows that the Rhino project is our second strongest within-project performer, with an AUC of 0.81. Hence, the poor performance of our Weighted Similarity Merge model is likely inflated due to Rhino's high within-project performance.

Our findings suggests that a simple merging approach as was used in RQ2-1 would likely suffice for future work. The benefit of training models using all of the projects tend to outweigh the benefits of narrowing the training dataset down to a smaller set of similar projects.

JIT models that are trained by merging multiple training datasets tend to perform well in a cross-project context. However, JIT models trained using the merged data of similar projects rarely outperform models trained by blindly merging all of the available training datasets.

(RQ2-3) Approach

RQ2-1 and RQ2-2 have shown that larger pools of training data tend to produce more accurate cross-project prediction models. This complements recent findings that "universal" defect prediction models may be an attainable goal (Zhang *et al.*, 2014). However, in order to build a versatile "universal" defect model, Zhang *et al.* (2014) propose context-aware rank transformation of predictor variables. Hence, we suspect that applying such transformations to our pool of training datasets may improve the performance of JIT cross-project models.

We briefly explain how we build a universal defect prediction model (details in Zhang *et al.* (2014)). The main idea of the universal model is to cluster projects based on the similarity of the distributions of each predictor and to derive rank transformations using quantiles of predictors for a cluster.

Figure 8 shows the steps that we followed to build a universal model. Each project is classified into one group based on context factors that have similar distributions of software metrics. Then, for each metric, we compare the distribution of the metric between groups. If there are few differences (i.e., similar projects) between two groups using statistical tests (i.e., Mann-Whitney U test, Bonferroni correction and Cohen's standards), we merge them into one cluster. This step is conducted for each metric. Therefore, two groups may be in the same cluster according to one metric, but different clusters according to another. Then, in each cluster, we transform the raw metrics value to the $k \times 10\%$ quantiles to make the distributions fit on the same scale across projects. For example, given a metric with values 11, 22, 33, 44, 55, 66, 77, 88 and 99 in one cluster, a new raw value of 25 would be transformed to the 3rd quantile, since $25 \geq 22$ (i.e., the 2^{nd} quantile), $25 < 33$ (i.e., the 3^{rd} quantile).

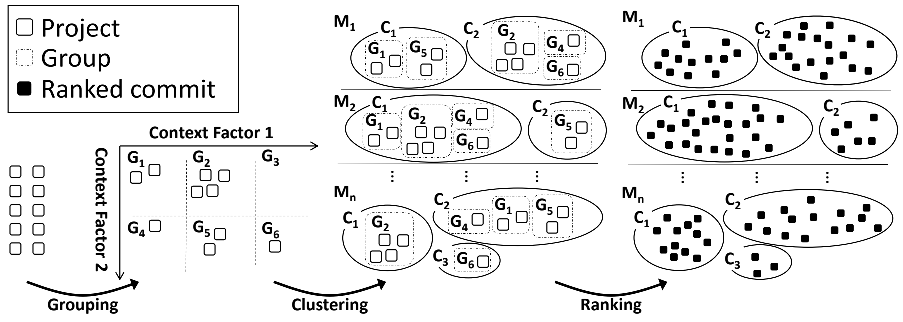


Fig. 8 Steps to train a universal defect prediction model.

After the above steps, the metrics are transformed such that they range between 1 and 10. Finally, we set aside the data of one project for testing, and use the remaining data to train a "universal" prediction model.

We adopt the same context factors as Zhang *et al.* (2013, 2014). We briefly outline these factors.

Programming language (Java/JavaScript/C/C++/Perl/Ruby): programming language used in the system development. We identify the programming language using Linguist.[3] We choose the most frequently used programming language in the system (i.e., the programming language with the largest number of files).

Issue Tracking (True/False): whether or not a project uses an issue tracking system.

Total Lines of Code (least, less, more, most): total lines of code of source code in the system. Based on the first, second and third quartiles, we separate the set of projects into four groups (i.e., least, less, more and most).

Total Number of Files (least, less, more, most): total number of files in the system. Based on the first, second and third quartiles, we separate the set of projects into four groups (i.e., least, less, more and most).

Total Number of Commits (least, less, more, most): total number of commits in the system. Based on the first, second and third quartiles, we separate the set of projects into four groups (i.e., least, less, more and most).

Total Number of Developers (least, less, more, most): total number of unique developers in the system. Based on the first, second and third quartiles, we separate the set of projects into four groups (i.e., least, less, more and most).

We exclude the issue tracking context factor, since all of the studied projects use an issue tracking system.

(RQ2-3) Results

Figure 9 shows the performance of universal JIT defect prediction models. The results show that the universal model achieves 56%-84% of the AUC of the within-project models. This is well below the results that we observed in the prior sections.

[3] `https://github.com/github/linguist`

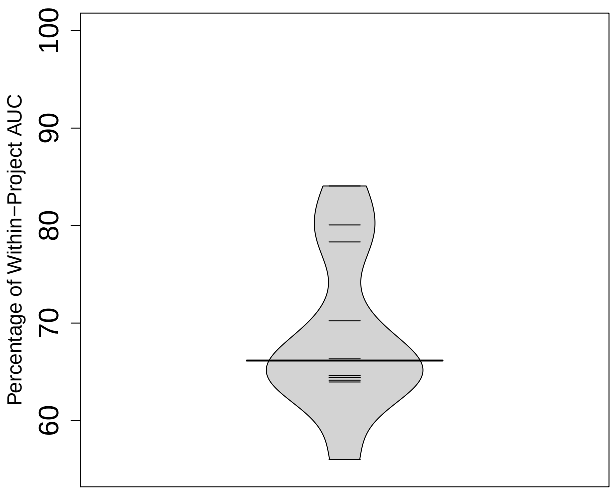


Fig. 9 [RQ2-3] The results of "universal" defect prediction (Zhang *et al.*, 2014) in a JIT cross-project context.

Our sample of 11 projects is similar in size to prior work on cross-project prediction, which focuses on samples of 8-12 projects (Nam *et al.*, 2013; Zimmermann *et al.*, 2009). However, this sample size may not be large enough to build a reliable universal defect prediction model. Indeed, Zhang *et al.* (2014) used 1,398 projects (937 SourceForge projects and 461 GoogleCode projects) initially collected by Mockus (2009) to build a universal defect prediction model at the file-level. Future work is needed to evaluate the performance of such universal modeling in the context of JIT prediction using a large sample of projects.

Since a "universal" defect prediction model likely needs to be trained on a large sample of projects, context-aware rank transformations do not tend to perform well in our context of JIT cross-project prediction in a sample of 11 projects.

(RQ3) Do ensembles of JIT models built from several projects perform well in a cross-project context?

In RQ2, we leveraged our collection of training datasets by training a single model using a pool of their collective data. Alternatively, in RQ3, we combine our training projects by training a model on each project individually (Mısırlı *et al.*, 2011; Thomas *et al.*, 2013). Then, each change in the testing project is passed through each of the

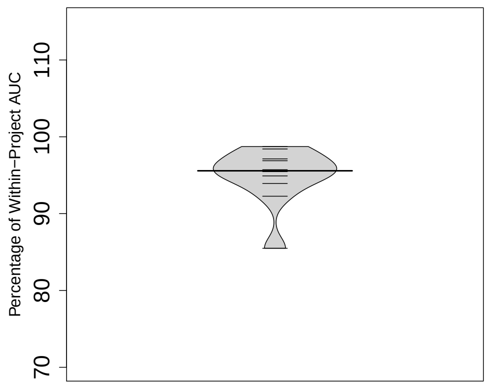


Fig. 10 [RQ3-1] The results of our simple voting approach.

project-specific models. Thus, for each change in the testing project, we receive several "votes", one from each of the project-specific models. We evaluate an approach that treats the votes of each of the project-specific models in training datasets equally (RQ3-1), and an approach that gives more weight to the votes of models trained using similar projects (RQ3-2).

(RQ3-1) Approach

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

We illustrate the voting method using an example in the case of Mozilla below. First, we select the 10 models of the other (non-Mozilla) projects. Given a change from Mozilla project, we obtain 10 predicted probabilities from the 10 models. Finally, we calculate the mean of the 10 probabilities.

(RQ3-1) Results

Figure 10 shows that the simple voting approach that we propose performs well, achieving 85%-99% of the within-project AUC values. Indeed, in Mozilla, the voting approach performs almost as well as the within-project model (99%).

Even simple ensemble techniques that combine the predictions of JIT models trained on each training dataset tend to perform well in a cross-project context.

(RQ3-2) Approach

Similar to RQ2-1 and RQ2-2, we suspect that applying similarity heuristics to our voting approaches may improve the performance of our ensemble models. We again evaluate two approaches to apply similarity to our JIT models:

1. For each testing dataset, we use our similarity metrics to select n training datasets, and then build prediction models for each selected training dataset. Then, we push a change through each prediction model and then take the mean of the predicted probabilities.
2. We evaluate a threshold-independent approach that provides more weight to the votes of the models of similar projects. Similar to RQ2, we use $\frac{10-(r-1)}{10} \times 100$ % to calculate the weight of a project's vote, where r is the project rank based on our similarity metric. For example, the vote of the most similar project is given full (100%) weight, while the vote of the second most similar project is given a weight of 90%, and so on.

(RQ3-2) Results

Figure 11 shows the results of applying our similarity-driven voting approaches. Again, to reduce clutter, we only show the Similarity Voting with threshold values of 3 and 5, and provide online access to the figure showing all possible threshold values between 2 and 10.[4]

We find that models trained using either of our similarity metrics do not tend to outperform the simple voting approach of RQ3-1. Tukey's HSD test results indicate that the difference between the result sets are not statistically significant in either of the cases ($p_{agnostic} = 0.765, p_{aware} = 0.661$). Hence, we the simple voting approach will likely suffice for future work.

Ensemble voting techniques also tends to yield JIT models perform well in a cross-project context. Similar to RQ2, dampening the impact of dissimilar projects does not improve the performance of our ensembles of JIT models.

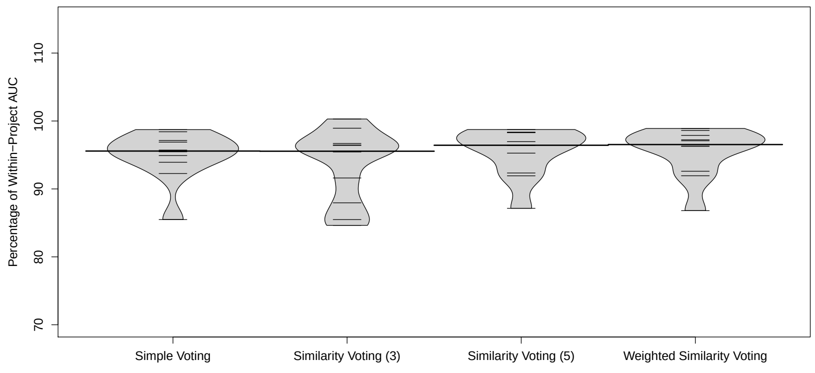
6 Discussion

6.1 Summary of results

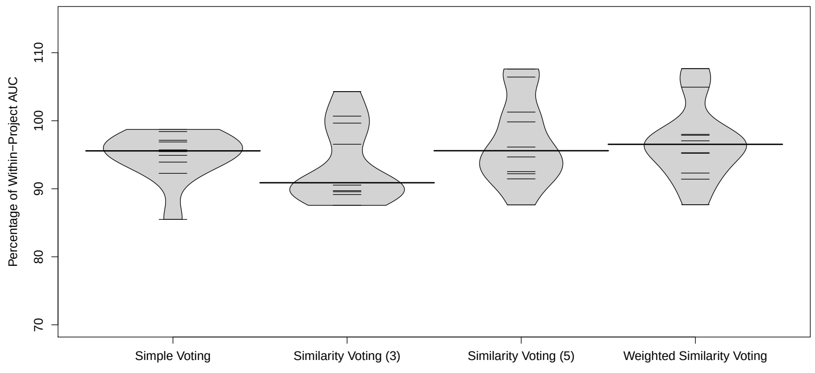
Below, we use statistical tests to provide yes/no answers for our research questions.

(Section 4) The within-project performance of a JIT model is not a strong indicator of its performance in a cross-project context.
We calculate the Spearman correlation between the rank of the AUC value of within-project prediction and the median of the AUC values of cross-project prediction. The value of Spearman correlation is $\rho = 0.036$ ($p = 0.924$).

[4] `http://posl.ait.kyushu-u.ac.jp/Disclosure/emse_jit.html`



(a) Domain-agnostic



(b) Domain-aware

Fig. 11 The result of similarity-driven voting approaches.

(RQ1) Do JIT models selected using project similarity perform well in a cross-project context?
The answer is no. Within-project JIT models significantly outperform domain-aware similarity techniques ($p = 0.005$).[5] Prediction performance was not improved by selecting datasets for training that are highly similar to the testing dataset.

(RQ2) Do JIT models built using a pool of data from several projects perform well in a cross-project context?
We find no evidence of a difference in the performance of within- and cross-

[5] We choose domain-aware similarity techniques, similarity merge (5) using domain-aware similarity and weighted similarity voting using domain-aware similarity, which show the best median value in each RQ, and within-project JIT models as ideal models. We check the difference of the median values among the four models using Tukey's HSD. If we find that there is not statistically significant difference between within- and one cross-project JIT models, we find no evidence of a difference in the performance of within- and cross-project models in those cases (i.e., the cross-project model perform well).

project models in RQ2. We find no statistically significant difference in the performance of within-project JIT models and similarity merge (5) using domain-aware similarity ($p = 0.967$).[5] Several datasets can be used in tandem to produce more accurate cross-project JIT models by sampling from a larger pool of training data.

(RQ3) Do ensembles of JIT models built from several projects perform well in a cross-project context?
We find no evidence of a difference in the performance of within- and cross-project models in RQ3. We find no statistically significant difference in the performance of within-project JIT models and weighted similarity voting using domain-aware similarity ($p = 0.825$).[5] Combining the predictions of several models could contribute to building more accurate cross-project JIT models.

6.2 Practical Guidelines

We propose the following guidelines to assist in future work:

Guideline 1: Future work should not use the within-project performance of a JIT model as a indicator of its performance in a cross-project context.
The value of Spearman correlation is $\rho = 0.036$ (Please see Section 6.1). In addition, even if we build the cross-project JIT models using the top project (i.e., MAV) and the bottom project (i.e., RUB) in Figure 2, we obtain the models of similar prediction performance.

Guideline 2: Future work should not use similarity to filter away dissimilar project data or models.
Our domain-agnostic and domain-aware similarity select JIT models that tend to perform better than the median cross-project performance. However, within-project JIT models outperform similarity-selected cross-project models to a statistically significant degree.

Guideline 3: Future work should consider data and models from other projects in tandem with one another to produce more accurate cross-project JIT models.
In practical settings, a simple merge approach (RQ2-1) might be the best choice that we evaluated in our experiments. We illustrate the advantages of the simple merge approach using the following example: Alice is a manager and Bob is a developer of a new system. Alice wants to use JIT models to promote risk awareness, but the system has not accrued sufficient historical data to train such models. She decides to use a cross-project approach. If Alice adopts the simple merge approach, she only needs to build one prediction model. On the other hand, if she adopts the voting method, she needs to build several prediction models (i.e., one model for every selected dataset). Thus, the simple merge approach may require less effort in the building phrase.

Table 7 Summary of precision 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	0.62	0.38	0.53	0.51	0.47	0.67	0.58	0.55	0.61	0.56	0.55
	COL	0.38	0.53	0.45	0.38	0.41	0.55	0.46	0.48	0.42	0.41	0.42
	GIP	0.43	0.33	0.58	0.50	0.43	0.47	0.44	0.42	0.47	0.48	0.46
	JDT	0.23	0.18	0.24	0.28	0.20	0.30	0.24	0.24	0.27	0.24	0.23
	MAV	0.13	0.13	0.23	0.18	0.26	0.26	0.20	0.21	0.19	0.19	0.19
	MOZ	0.10	0.06	0.10	0.09	0.08	0.14	0.10	0.09	0.12	0.10	0.10
	PER	0.48	0.21	0.42	0.41	0.36	0.47	0.44	0.30	0.42	0.45	0.41
	PLA	0.26	0.20	0.26	0.23	0.21	0.32	0.25	0.30	0.26	0.24	0.23
	POS	0.32	0.27	0.36	0.38	0.31	0.50	0.39	0.36	0.51	0.42	0.36
	RUB	0.27	0.23	0.34	0.30	0.27	0.40	0.33	0.30	0.36	0.34	0.32
	RHI	0.59	0.55	0.69	0.54	0.57	0.73	0.66	0.60	0.69	0.64	0.70

Table 8 Summary of recall 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	0.63	0.64	0.54	0.72	0.67	0.39	0.51	0.52	0.35	0.50	0.58
	COL	0.48	0.66	0.50	0.91	0.36	0.53	0.53	0.74	0.76	0.61	0.59
	GIP	0.10	0.44	0.73	0.62	0.60	0.38	0.80	0.86	0.56	0.49	0.76
	JDT	0.48	0.69	0.60	0.64	0.60	0.53	0.60	0.71	0.56	0.61	0.66
	MAV	0.19	0.45	0.61	0.70	0.74	0.45	0.70	0.73	0.59	0.72	0.75
	MOZ	0.67	0.71	0.72	0.82	0.78	0.67	0.75	0.82	0.70	0.72	0.78
	PER	0.17	0.44	0.50	0.43	0.59	0.25	0.63	0.72	0.42	0.38	0.62
	PLA	0.62	0.66	0.58	0.80	0.57	0.50	0.58	0.68	0.56	0.62	0.64
	POS	0.40	0.65	0.65	0.74	0.79	0.57	0.72	0.81	0.65	0.66	0.75
	RUB	0.22	0.57	0.37	0.46	0.58	0.32	0.59	0.56	0.44	0.63	0.54
	RHI	0.32	0.65	0.54	0.50	0.81	0.49	0.72	0.77	0.50	0.58	0.70

6.3 Additional Analysis

Do other performance metrics provide different results?

To avoid the impact of the threshold that is used for classification, we used AUC. However, there are several studies that use threshold-dependent performance metrics (e.g., precision, recall and F-measure) (Kamei *et al.*, 2013; Kim *et al.*, 2008). To better understand the prediction performance we obtained, we also show the results using threshold-dependent metrics.

The JIT models that we train using Random forest produce a risk probability for each change, i.e., a value between 0 and 1. We use a threshold value of 0.5, which means that if a change has a risk probability greater than 0.5, the change is classified as defect-inducing, otherwise it is classified as clean. Table 7, 8 and 9 show precision, recall and F-measure of our JIT models, similar to the AUC of Table 6.

In addition to AUC, within-project JIT models also outperform cross-project JIT models in terms of f-measure, precision and recall. While the median of the f-measures of cross-project models (0.419) is higher than that of random guessing

Table 9 Summary of F-measure 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	0.63	0.48	0.54	0.59	0.55	0.49	0.54	0.54	0.45	0.53	0.56
	COL	0.43	0.59	0.48	0.54	0.38	0.54	0.49	0.58	0.54	0.49	0.49
	GIP	0.16	0.38	0.65	0.55	0.50	0.42	0.57	0.57	0.51	0.48	0.57
	JDT	0.31	0.29	0.34	0.39	0.29	0.38	0.34	0.36	0.36	0.34	0.34
	MAV	0.15	0.20	0.33	0.28	0.38	0.33	0.31	0.33	0.29	0.29	0.30
	MOZ	0.17	0.12	0.17	0.16	0.14	0.23	0.17	0.16	0.20	0.18	0.17
	PER	0.25	0.28	0.45	0.42	0.45	0.33	0.52	0.42	0.42	0.41	0.49
	PLA	0.37	0.31	0.36	0.36	0.31	0.39	0.35	0.42	0.35	0.35	0.34
	POS	0.35	0.38	0.46	0.50	0.45	0.53	0.51	0.49	0.57	0.51	0.49
	RUB	0.24	0.33	0.35	0.36	0.37	0.36	0.43	0.39	0.39	0.44	0.40
	RHI	0.42	0.60	0.60	0.52	0.67	0.59	0.69	0.67	0.58	0.61	0.70

(0.324), the median of the f-measures of within-project models is still higher than cross-project models (0.521). Similar observations hold for precision and recall.

Does a sampling approach provide different results?

Similar to Kamei *et al.* (2007); Menzies *et al.* (2008), we use a re-sampling approach for our training data to deal with the imbalance of defect-inducing and clean classes in our dataset. On the other hand, Turhan (2012) points out that such sampling approaches make the distributions of the training and testing sets incongruent. Therefore, we reevaluate our experiment from Section 4 without re-sampling the training dataset. Table 10, Table 11, 12 and 13 show AUC, precision, recall and F-measure of our JIT models.

We find that there are negligible differences between the models that are trained with and without re-sampling the training data in terms of AUC. The median AUC values for cross-project prediction are 0.71 with re-sampling and 0.69 without.

On the other hand, re-sampling tends to provides better model performance in terms of f-measure and recall. We find that cross-project f-measure values are 0.42 with re-sampling and 0.25 without. Recall values show a similar trend to the f-measure values. We also found that re-sampling tends to decrease the prediction performance in terms of precision. These results are consistent with the results of our previous work (Kamei *et al.*, 2007).

7 Threats to Validity

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

7.1 Construct Validity

We estimate within-project prediction performance using tenfold cross validation. However, tenfold cross validation may not be representative of the actual cross-version model performance. In reality, training data is always younger than the testing

Table 10 Summary of AUC values for models using the training data without re-sampling.

		Training project										
		BUG	COL	GIP	JDT	MAV	MOZ	PER	PLA	POS	RUB	RHI
Testing project	BUG	0.76	0.70	0.70	0.73	0.66	0.71	0.70	0.73	0.70	0.70	0.68
	COL	0.60	0.77	0.62	0.75	0.63	0.72	0.67	0.73	0.75	0.66	0.63
	GIP	0.46	0.67	0.80	0.72	0.70	0.71	0.68	0.71	0.72	0.66	0.67
	JDT	0.63	0.72	0.68	0.75	0.63	0.73	0.70	0.75	0.72	0.70	0.67
	MAV	0.42	0.75	0.76	0.75	0.82	0.76	0.71	0.78	0.75	0.73	0.76
	MOZ	0.72	0.76	0.75	0.79	0.70	0.80	0.75	0.80	0.78	0.75	0.74
	PER	0.59	0.64	0.70	0.70	0.69	0.69	0.76	0.70	0.68	0.69	0.71
	PLA	0.71	0.73	0.69	0.77	0.64	0.76	0.70	0.78	0.72	0.69	0.68
	POS	0.52	0.76	0.73	0.75	0.71	0.75	0.71	0.75	0.79	0.74	0.72
	RUB	0.55	0.67	0.66	0.70	0.62	0.65	0.71	0.66	0.69	0.74	0.68
	RHI	0.61	0.73	0.74	0.73	0.74	0.77	0.78	0.74	0.73	0.71	0.81

Table 11 Summary of precision values for models using the training data without re-sampling.

		Training project										
		BUG	COL	GIP	JDT	MAV	MOZ	PER	PLA	POS	RUB	RHI
Testing project	BUG	0.70	0.66	0.68	0.74	0.50	0.45	0.69	0.71	0.63	0.63	0.58
	COL	0.49	0.67	0.59	0.72	0.33	0.57	0.58	0.65	0.65	0.60	0.42
	GIP	0.48	0.52	0.66	0.67	0.64	0.88	0.57	0.79	0.58	0.54	0.48
	JDT	0.30	0.37	0.48	0.57	0.38	0.76	0.37	0.54	0.38	0.37	0.24
	MAV	0.18	0.23	0.52	0.46	0.68	0.75	0.29	0.53	0.35	0.29	0.23
	MOZ	0.14	0.16	0.24	0.26	0.26	0.57	0.17	0.28	0.20	0.19	0.11
	PER	0.60	0.51	0.50	0.55	0.57	0.75	0.64	0.57	0.51	0.50	0.46
	PLA	0.36	0.38	0.50	0.53	0.52	0.77	0.37	0.62	0.40	0.37	0.25
	POS	0.43	0.58	0.68	0.66	0.72	0.90	0.56	0.69	0.68	0.69	0.41
	RUB	0.33	0.46	0.49	0.60	0.48	0.83	0.50	0.57	0.52	0.56	0.36
	RHI	0.70	0.79	0.77	0.79	0.82	0.86	0.79	0.83	0.82	0.86	0.74

data. When using tenfold cross validation, folds are constructed randomly, which may produce folds that use older changes for training than testing. While tenfold cross-validation is a popular performance estimation technique (Kim *et al.*, 2008; Moser *et al.*, 2008), other performance estimation techniques (e.g., using training and testing data based on releases) may yield different results.

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 (Moser *et al.*, 2008; Ratzinger *et al.*, 2008)) might influence the likelihood of introducing a defect. We plan to expand our metric set to include additional categories in future work.

7.2 Internal Validity

We use defect datasets provided by prior work (Kamei *et al.*, 2013) that identify defect-inducing changes using the SZZ algorithm (Śliwerski *et al.*, 2005). The SZZ algorithm is commonly used in defect prediction research (Kim *et al.*, 2008; Moser *et al.*, 2008), yet has known limitations. For example, if a defect is not recorded in the

Table 12 Summary of recall values for models using the training data without re-sampling.

		Training project										
		BUG	COL	GIP	JDT	MAV	MOZ	PER	PLA	POS	RUB	RHI
Testing project	BUG	0.43	0.14	0.11	0.13	0.03	0.00	0.21	0.13	0.13	0.13	0.46
	COL	0.30	0.40	0.05	0.11	0.00	0.00	0.23	0.09	0.32	0.25	0.38
	GIP	0.07	0.11	0.55	0.14	0.25	0.01	0.35	0.11	0.40	0.16	0.65
	JDT	0.31	0.25	0.06	0.08	0.01	0.00	0.20	0.08	0.26	0.22	0.50
	MAV	0.11	0.12	0.22	0.15	0.22	0.03	0.23	0.21	0.38	0.27	0.66
	MOZ	0.57	0.34	0.27	0.30	0.13	0.06	0.47	0.30	0.46	0.39	0.69
	PER	0.12	0.06	0.24	0.09	0.12	0.01	0.25	0.06	0.25	0.09	0.47
	PLA	0.42	0.24	0.08	0.13	0.01	0.01	0.22	0.16	0.25	0.22	0.47
	POS	0.30	0.24	0.29	0.20	0.18	0.02	0.30	0.21	0.37	0.28	0.66
	RUB	0.13	0.12	0.04	0.04	0.03	0.00	0.12	0.03	0.15	0.17	0.41
	RHI	0.23	0.17	0.12	0.11	0.07	0.00	0.29	0.08	0.24	0.21	0.64

Table 13 Summary of F-measure values for models using the training data without re-sampling.

		Training project										
		BUG	COL	GIP	JDT	MAV	MOZ	PER	PLA	POS	RUB	RHI
Testing project	BUG	0.53	0.23	0.19	0.22	0.06	0.01	0.32	0.22	0.21	0.22	0.52
	COL	0.37	0.50	0.09	0.19	0.00	0.01	0.33	0.16	0.43	0.35	0.40
	GIP	0.12	0.19	0.60	0.24	0.36	0.02	0.43	0.20	0.48	0.25	0.56
	JDT	0.30	0.30	0.11	0.13	0.01	0.01	0.26	0.14	0.31	0.28	0.33
	MAV	0.14	0.16	0.31	0.22	0.33	0.05	0.25	0.30	0.36	0.28	0.34
	MOZ	0.22	0.22	0.25	0.28	0.17	0.11	0.25	0.29	0.28	0.25	0.19
	PER	0.19	0.11	0.32	0.16	0.20	0.02	0.36	0.11	0.34	0.15	0.46
	PLA	0.39	0.29	0.13	0.21	0.03	0.01	0.27	0.25	0.31	0.27	0.33
	POS	0.35	0.34	0.41	0.31	0.29	0.03	0.39	0.32	0.48	0.40	0.51
	RUB	0.19	0.19	0.07	0.07	0.05	0.00	0.19	0.06	0.24	0.26	0.38
	RHI	0.35	0.27	0.20	0.19	0.13	0.01	0.42	0.15	0.37	0.34	0.68

VCS commit message or the keywords used defect identifiers differ from those used in the previous study (e.g., “Bug” or “Fix” (Kamei *et al.*, 2007)), 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 (Wu *et al.*, 2011) may improve the accuracy of our results.

7.3 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.

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 datasets collected from other projects (i.e., cross-project prediction). We evaluate the use of conventional data mining and software engineering context to produce JIT models that perform well in a cross-project context. Through an empirical study on 11 open source projects, we make the following observations:

- The within-project performance of a JIT model is not a strong indicator of its performance in a cross-project context (Section 4).
- Although using similarity to select JIT models from a collection of choices tends to identify the best-performing option for a cross-project context, the performance of these models is significantly lower than within-project model performance (RQ1).
- Several datasets can be used in tandem to produce more accurate cross-project JIT models by sampling from a larger pool of training data (RQ2) or combining the predictions of several models (RQ3). The performance of these models is statistically indistinguishable from within-project JIT model performance.
- However, using project similarity to filter away dissimilar project data (RQ2) or models (RQ3) does not tend to improve the cross-project performance of JIT models that use all available training data.

9 Acknowledgments

This research was partially supported by JSPS KAKENHI Grant Numbers 15H05306 and 24680003 and the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

Basili VR, Briand LC, Melo WL (1996) A validation of object-oriented design metrics as quality indicators. IEEE Trans Softw Eng 22(10):751–761

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

Breiman L (2001) Random forests. Machine learning 45(1):5–32

Briand LC, Melo WL, Wüst J (2002) Assessing the applicability of fault-proneness models across object-oriented software projects. IEEE Trans Softw Eng 28(7):706–720

Coolidge FL (2012) Statistics: A Gentle Introduction. SAGE Publications (3rd ed.)

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

Fukushima T, Kamei Y, McIntosh S, Yamashita K, Ubayashi N (2014) An empirical study of just-in-time defect prediction using cross-project models. In: Proc. Int'l Working Conf. on Mining Software Repositories (MSR'14), pp 172–181

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

Guo PJ, Zimmermann T, Nagappan N, Murphy B (2010) Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In: Proc. Int'l Conf. on Softw. Eng. (ICSE'10), vol 1, pp 495–504

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

Hassan AE (2009) Predicting faults using the complexity of code changes. In: Proc. Int'l Conf. on Softw. Eng. (ICSE'09), pp 78–88

He Z, Shu F, Yang Y, Li M, Wang Q (2012) An investigation on the feasibility of cross-project defect prediction. Automated Software Engg 19(2):167–199

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

Kamei Y, Monden A, Matsumoto S, Kakimoto T, Matsumoto Ki (2007) 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), pp 196–204

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

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

Kampstra P (2008) Beanplot: A boxplot alternative for visual comparison of distributions. Journal of Statistical Software, Code Snippets 28(1):1–9

Kim S, Whitehead EJ, Zhang Y (2008) Classifying software changes: Clean or buggy? IEEE Trans Softw Eng 34(2):181–196

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

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

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

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

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

McIntosh S, Nagappan M, Adams B, Mockus A, Hassan AE (2014) A large-scale empirical study of the relationship between build technology and build maintenance. Empirical Software Engineering DOI 10.1.1/jpb001, URL `http://link.springer.com/article/10.1007%2Fs10664-014-9324-x`

Menzies T, Turhan B, Bener A, Gay G, Cukic B, Jiang Y (2008) Implications of ceiling effects in defect predictors. In: Proc. Int'l Conf. on Predictive Models in Softw. Eng. (PROMISE'10), pp 47–54

Menzies T, Butcher A, Marcus A, Zimmermann T, Cok D (2011) Local vs. global models for effort estimation and defect prediction. In: Proc. Int'l Conf. on Automated Software Engineering (ASE'11), pp 343–351

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

Minku LL, Yao X (2014) How to make best use of cross-company data in software effort estimation? In: Proc. Int'l Conf. on Software Engineering (ICSE'14), pp 446–456

Mısırlı AT, Bener AB, Turhan B (2011) An industrial case study of classifier ensembles for locating software defects. Software Quality Journal 19(3):515–536

Mockus A (2009) Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In: Proc. Int'l Working Conf. on Mining Software Repositories (MSR'09), pp 11–20

Mockus A, Weiss DM (2000) Predicting risk of software changes. Bell Labs Technical Journal 5(2):169–180

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

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

Nagappan N, Ball T, Zeller A (2006) Mining metrics to predict component failures. In: Proc. Int'l Conf. on Softw. Eng. (ICSE'06), pp 452–461

Nam J, Pan SJ, Kim S (2013) Transfer defect learning. In: Proc. Int'l Conf. on Softw. Eng. (ICSE'13), pp 382–391

Purushothaman R, Perry DE (2005) Toward understanding the rhetoric of small source code changes. IEEE Trans Softw Eng 31(6):511–526

Rahman F, Posnett D, Devanbu P (2012) Recalling the "imprecision" of cross-project defect prediction. In: Proc. Int'l Symposium on the Foundations of Softw. Eng. (FSE'12), pp 61:1–61:11

Ratzinger J, Sigmund T, Gall HC (2008) On the relation of refactorings and software defect prediction. In: Proc. Int'l Working Conf. on Mining Software Repositories (MSR'08), pp 35–38

Shihab E (2012) An exploration of challenges limiting pragmatic software defect prediction. PhD thesis, Queen's University

Shihab E, Hassan AE, Adams B, Jiang ZM (2012) An industrial study on the risk of software changes. In: Proc. Int'l Symposium on the Foundations of Softw. Eng. (FSE'12), pp 62:1–62:11

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

Tan M, Tan L, Dara S, Mayuex C (2015) Online defect prediction for imbalanced data. In: Proc. Int'l Conf. on Softw. Eng. (ICSE'13 SEIP), p (To appear)

Thomas SW, Nagappan M, Blostein D, Hassan AE (2013) The impact of classifier configuration and classifier combination on bug localization. IEEE Trans Softw Eng 39(10):1427–1443

Turhan B (2012) On the dataset shift problem in software engineering prediction models. Empirical Softw Engg 17(1-2):62–74

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

Turhan B, Tosun A, Bener A (2011) Empirical evaluation of mixed-project defect prediction models. In: Proc. EUROMICRO Conf. on Software Engineering and Advanced Applications (SEAA'11), pp 396–403

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

Zhang F, Mockus A, Zou Y, Khomh F, Hassan AE (2013) How does context affect the distribution of software maintainability metrics? In: Proc. Int'l Conf. on Software Maintenance (ICSM'13), pp 350–359

Zhang F, Mockus A, Keivanloo I, Zou Y (2014) Towards building a universal defect prediction model. In: Proc. Int'l Working Conf. on Mining Software Repositories (MSR'14), pp 182–191

Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B (2009) Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proc. European Softw. Eng. Conf. and Symposium on the Foundations of Softw. Eng. (ESEC/FSE'09), pp 91–100