Recalling the "Imprecision" of Cross-Project Defect Prediction

Foyzur Rahman Daryl Posnett Premkumar Devanbu

Department of Computer Science
University of California Davis, Davis, CA., USA
{mfrahman,dpposnett,ptdevanbu}@ucdavis.edu

ABSTRACT

There has been a great deal of interest in *defect prediction*: using prediction models trained on historical data to help focus quality-control resources in ongoing development. Since most new projects don't have historical data, there is interest in *cross-project prediction*: using data from one project to predict defects in another. Sadly, results in this area have largely been disheartening. Most experiments in cross-project defect prediction report poor performance, using the standard measures of *precision, recall and F-score*. We argue that these IR-based measures, while broadly applicable, are not as well suited for the quality-control settings in which defect prediction models are used. Specifically, these measures are taken at *specific threshold settings* (typically thresholds of the predicted probability of defectiveness returned by a logistic regression model). However, in practice, software quality control processes choose from a *range* of time-and-cost *vs* quality tradeoffs: how many files shall we test? how many shall we inspect? Thus, we argue that measures based on a variety of tradeoffs, *viz.*, 5%, 10% or 20% of files tested/inspected would be more suitable. We study cross-project defect prediction from this perspective. *We find that cross-project prediction performance is no worse than within-project performance, and substantially better than random prediction!*

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Code Inspections and Walk-Throughs*

General Terms

Experimentation; Measurement; Reliability; Verification

Keywords

Empirical Software Engineering, Fault Prediction, Inspection

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.

1. INTRODUCTION

Poor software quality is of vital concern; finding and fixing defects before release is a very high priority for development organizations. Unfortunately, given limited resources of budget, people and time, *only some* of the code can be fully tested or inspected before release. *Defect prediction models* are a statistical approach to help identify the most defect-prone components, and help stakeholders focus limited quality control resources on those bits of the system most likely to contain defects.

Typically, defect prediction models use a *training set* containing various measures (code metrics, network metrics, process metrics etc.) of the available software entities (files, modules, binaries etc.) as *predictors* and the history of defect proneness as the *response*. A prediction model is fit to this training set, and used to predict the future defect proneness of the entities from the currently available measured value of predictors. Such models have performed well in practice and are regularly used in the industry: for example, engineers at Google have built[1] a prediction model based on some earlier work by Kim *et al.* [13] and us [24]. Such models have generally been used in a within-project prediction setting, *viz.*, using data from earlier releases to predict defects in later releases within the *same* project.

Unfortunately, software is inherently a competitive and protean business, changing rapidly in response to changes in markets, hardware and software platforms. New projects start, and old ones get completely rewritten. This is a challenge for defect prediction models, which rely on historical defect data to predict future defect proneness. For many new projects we may not have enough historical data to train prediction models. So what can be done? For new projects, the obvious approach is to use models estimated from any training data available on older projects.

Researchers have already attempted the reuse of such models in cross-project settings. So far the results have been discouraging [5, 26, 31]. This is not surprising. One might reasonably expect that the distributions of the predictor metrics, conditioned on defect occurrence, in different projects are in fact different; thus models estimated in one project may not perform well on another project. This has been observed in a large number of studies.

We take a different approach to evaluate the efficacy of cross-project defect prediction. We argue that given the finite

[1] `http://google-engtools.blogspot.com/view/classic`

amount of resources available for testing and code inspection, one cannot possibly inspect or test all the entities that the model predicted as defective. So, we really want to evaluate the cross-project prediction performance, *when constrained within reasonable and practical resource limitations.*

This paper describes our work resource-constrained cross-project prediction:

- First, we collect a high-quality dataset of software metrics and defect occurrence. We build *within-release* prediction models, and test their performance at *within-project prediction*, and *cross-project prediction.* Our methodology is consistent with prior work in the area.
- When using the conventional resource-unconstrained measures reported in the literature, our results are well in-line with those previously reported; this indicates that our data are not inconsistent with prior work.
- Next, we measure model performance when applied in cross-project prediction settings under resource constraints. We find that, in fact, *cross-project defect performance is adequate, and comparable to within-project prediction*; we also find that this performance is quite stable across a range of possible cost settings.
- We evaluate the prediction model further to determine the factors that contribute to the cross-project cost-sensitive prediction performance.

2. BACKGROUND AND THEORY

Defect prediction models are learned from a training dataset which has both actual defect information and a set of predictive metrics that are believed to indicate defect proneness. Supervised learning techniques (such as logistic regression, decision trees, SVM, *etc.*) over the training data are used to induce models that can predict defect proneness from predictive metrics. These metrics typically measure product properties (*e.g.*, larger and more complex files are more likely to be defect prone), or process properties (heavily changed, and multiply authored, files are thought more likely to be defect prone). Prediction models have been a topic of considerable interest.

These approaches, however, are grounded in supervised learning techniques; thus, the quality, representativeness, and volume of the training data have a major influence on the usefulness and stability of model performance. Prior research found that the SE training data has significant quality issues [3, 4]; However, others report that prediction models can perform reasonably well albeit being trained on the noisy data [12]. But what if we don't have any data, noisy or otherwise? New projects are always emerging, and old ones are being rewritten. How should this be handled?

This is clearly an important consideration, and has attracted considerable attention from researchers.

To the best of our knowledge Briand *et al.* [5] first tried to port a model across different projects. They build a logistic regression model on the defect data of one open source project and try to use the learned prediction model on the defect data of another open source project which was developed by the same set of developers. Their findings indicate that even when the development team is similar, cross-project defect prediction is challenging and has lower prediction performance than their within project siblings. This study did consider cost-sensitive prediction; but they consider two very similar projects. Subsequently, many researchers have studied cross-project prediction performance to make it more useful and relevant. Turhan *et al.* [26] tried to improve prediction performance by selecting a similar set of training data using a nearest neighbor technique. While this approach yielded better performance than training the model with all data, the performance was still significantly lower than within-project models. Furthermore, nearest neighbor based filtering could be computationally expensive for large software projects. In their later paper, Turhan *et al.* [27] also found that adding mixed project data to an existing prediction model did not help much. This casts further doubt on the usefulness of the cross-project data.

Cross-project prediction even fails for projects drawn from the same domain. Zimmermann *et al.* [31] tried to port models between two web browsers (Internet Explorer and Firefox) and found that cross-project prediction was still not consistent: a model built on Firefox was useful for Explorer, but not *vice versa.*

Menzies *et al.* [19] found that prediction performance may not even be generalizable within a project and a given project's data may have many local regions which, when aggregated at a global level, may offer a completely different conclusion in terms of both quality control and effort estimation. Moreover, conclusions from local models are typically superior and more insightful than global models.

He *et al.* [10] sought the "empirical ceiling" of cross-project defect prediction performance. For each test set, they pick the best possible cross-project models among all available models built from various cross-project training sets using the test set oracle and found that such (unrealistic) models may even perform better than the standard within-project models. However, this is a post-facto approach that assumes perfect defect oracles on the test set, and thus not intended for practical prediction settings.

Existing research results on cross-project defect prediction are thus largely pessimistic; we take a different perspective. Most current research compares cross-project models with within-project models for the entire set of entities in the test project. While this approach works well in other settings (*e.g.* disease diagnosis, or information retrieval), real-world software development must be mindful of resource constraints. *Only some* of the files/entities predicted to be defective can realistically be subjected to additional quality-control measures. For example, during code inspection, a manager only looks at the top n% of the reported defect prone lines or files (where the n is resource-constained). Thus, the pivotal question is: what is the cross-project defect prediction performance at that critical top n% region? The pareto "like" distribution of software phenomena [14, 30] (defect occurrence, change-proneness, size, *etc.*) gives cause for optimism: since a few files account for most of the defects, if we find even some of these, we can still help focus quality control resources!

Our work was animated by the hope, that, from a cost-sensitive perspective, it's possible that cross-project defect prediction could still provide added value.

In the following section, we discuss the IR performance measures in use, and the cost-sensitive measures we propose to use instead.

3. PERFORMANCE MEASURES

A defect prediction model typically tags each file as either defective or non-defective. However, prediction models are not infallible. A model could fail to identify defective files (False Negatives, counted as FN)). Conversely, it could declare defect-free files as defective (False Positives, counted as FP). If working well, the model may correctly identify actually defective files, (True Positives, TP) and actually defect-free files, (True Negatives, TN). We can now define several commonly used performance measures.

3.1 Accuracy

Accuracy is the proportion of correct predictions, encompassing both predicting defective entities as defective and defect free entities as not-defective. It is defined as $\frac{TP+TN}{TP+FP+TN+FN}$. As is evident from the equation, accuracy is heavily influenced by class-balance: Typically, software projects have far more defect-free entities than defective entities. A useless model that declared all files to be defect-free will have high accuracy, almost equal to one that classified everyfile correctly. Accuracy is therefore not the best measure for comparing defect prediction models, in the usual high-imbalance defect datasets; this was noted by Ma *et al.* [16].

3.2 Precision

Precision of a model is the proportion of entities that are actually defective given that the model pronounces it defective. This estimates the expected return on investment if we are to spend quality-control resources on all the files indicated by the model as defective. If the precision is low, the resources would be mostly spent in vain. Precision is written as $\frac{TP}{TP+FP}$.

3.3 Recall

Recall of a model is the proportion of defective entities correctly identified by the model. It is defined as $\frac{TP}{TP+FN}$. This means, we want a very high recall to identify as many of the defects as possible.

A good model should achieve both high recall (finding most of bugs) and high precision (avoid too much wasted effort); however, it is well-known that maximizing precision/recall diminishes recall/precision. This brings us to another measure known as F-Measure.

3.4 F-Measure

F-Measure is simply the harmonic mean of precision and recall. This balances out the precision-recall tradeoff and gives one unified score to compare the models.

As is apparent from the above discussion, all of the so far discussed metrics depend on the values of TP, FP, TN and FN, which require a binary decision from the model. However, many of the classification techniques that are used to build models, do not yield binary decisions, rather they give the probability of an entity being defective. This immediately poses a challenge on how to discretize this continuous probability into a binary decision. We use a *cutoff* probability to define such discretization, where any entity with a defect proneness probability > cutoff would be considered by the model as defective, otherwise the model would consider the entity as defect free. Lessermann *et al.* [15] argue against using any metric in a defect prediction context that requires defining a threshold. Mende found [18] that reproducing SE defect prediction performance may be difficult due to the varying cutoffs used in the literature. For our purposes we baseline the probability threshold at 0.5, as used in the widely-cited and influential Zimmerann *et al.* [31] paper.

3.5 ROC

Receiver operating characteristic (ROC)is a non-parametric method of evaluating models, that is unaffected by class imbalance (proportion of true positives in the dataset) and also is (unlike the 3 measures above) independent of the cutoff. It represents the precision/recall pairing for all possible cutoff values between 0 and 1. ROC is a 2-D curve which plots the True Positive Rate TPR $= \frac{TP}{TP+FN}$ on the y-axis and False Positive Rate FPR $= \frac{FP}{FP+TN}$ on the x-axis. All such points pass through (0,0) and (1,1). The best possible model is an ROC curve close to $y = 1$: rises as steeply as possible, hits 1, and stays there. A random model will be close to the diagonal $y = x$. The area under the ROC curve is a measure of model performance. A perfect model will have an area of 1.0. A random prediction yields a ROC area of 0.5. We use AUC to signify the area under the ROC curve. AUC has a good property: regardless of the actual proportion of true positive, a random predictor always has the value 0.5! That, together with its independence of threshold setting, makes it a useful measure to compare within-*vs*-cross project prediction performance. Unfortunately, with few exceptions (e.g., [17]), prior research in cross-project prediction has *not* used AUC, which leads us to our first question.

> **Research Question 1:** How does within-project prediction performance compare with cross-project performance, when measured using AUC?

While the AUC is a very good metric for comparing different models or evaluating model performance, it ignores the cost-effectiveness of such models in a resource-constrained setting, such as when selecting code-entities for inspection. A model might very accurately predict defects, but if we are only capable of inspecting a fraction of that code, AUC may not be a good measure to evaluate a model. Arisholm *et al.* [1] propose a measure that considers cost effectiveness.

3.6 Cost Effectiveness

AUCEC is a cost-sensitive measure of defect prediction performance. It is defined as the "area under the cost-effectiveness curve". In this paper, we mostly focus on inspection or testing of a partial set of files. We want to maximize the utility of any such effort, where the utility is defined by the percentage of defects that we find.

Given a model that predicts defect-proneness, we can rank all the files according to the model's predicted defect-proneness score. Files predicted as more defect-prone would be ranked higher. Assuming that the defects are not completely randomly distributed throughout the system, the best possible model would rank the most defect-dense files higher (we can break ties by ranking smallest file higher). Guided by this approach, we should be able to inspect a *smaller* fraction of our code, yet finding a disproportionately *larger* fraction of defects. But this "fraction" that we want to inspect isn't fixed *a priori*. Depending on available resources, we may want to just inspect 5% of SLOC under deadline

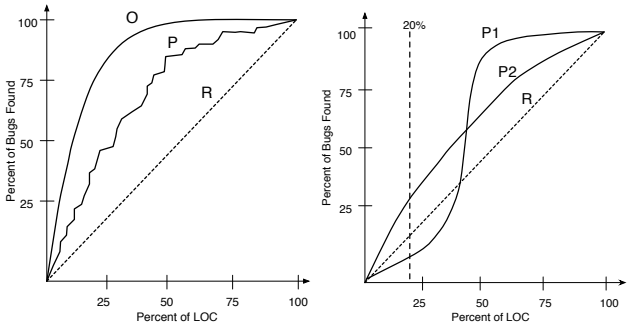

Figure 1: Cost Effectiveness Curve. On the left, O is the optimal, R is random, and P is a possible, *practical*, predictor model. On the right, we have two different models P1 and P2, with the same overall performance, but P2 is better when inspecting 20% of the lines or less. Figure quoted from our FSE 2011 paper [24]

.

pressure, 20% SLOC at other times. Therefore, we want a model which performs well for various operating percentage of SLOC.

A model that works well upon inspecting 80% of SLOC, may not work as well when inspecting only 20% of SLOC. Therefore, to summarize the efficacy of such models, we could just plot model performance in terms of percentage of bugs found after considering the top n% of SLOC, for different n. The left side plot in figure 1 (reproduced from [24]) depicts one such plot using three models, The optimal model possible, which would always order the files according to their defect density, is designated as O, a typical "good" prediction model is designated as P, and the random prediction is designated as R. Note, the random prediction model would just pick random files as defective, and therefore, in expectation, we will find x% of bugs after inspecting x% of SLOC. Similar to AUC, we can find the area under such cost-effectiveness curve to understand the model effectiveness at various percentage of SLOC. We can then use such area to compare different models at different percentages of SLOC.

The right side plot of Figure 1 shows the cost-effectiveness curve for two models P1, and P2. The dotted vertical line depicts our budget for inspection, which in this example is set to 20% of SLOC. Clearly, within this budget, P2 is a much better predictor than P1. However in the entire operating region, using 100% SLOC, we may find that P1 and P2 have identical cost-effectiveness. So, the performance of AUCEC also considers the area under the cost-effectiveness curve up to a particular threshold, tailored for a typical usage scenario of the model. This is the concept of AUCEC, defined by Arisholm *et al.* [1] to investigate the defects in telecommunications software.

Even if a model $\mathcal{M}$ has less PRECISION, and prescribes the inspection of too many files, as long as files predicted by $\mathcal{M}$ as *most defect-prone are the most defect-dense*, $\mathcal{M}$ is cost-effective, *and* would have a high measured value of AUCEC. A similar argument may apply for models with low RECALL. For this reason, we advocate measuring the cost-effectiveness of cross-project defect prediction using AUCEC.

Research Question 2: Does cross-project defect prediction work in terms of cost effectiveness?

Typically, AUCEC is evaluated at 10% and 20% of the SLOC in the system; however, in this study of performance of cross-project models, we examine AUCEC at a wide range of levels. At each level, we compare within-*vs*-cross performance. We have two motivations for this: first, this comparison gives some insight into the performance of the models at different levels, and how much they improve upon a trivial random model; secondly, there are some settings (*e.g.* safety-critical software) where it might be reasonable to inspect almost all the code, and this comparison gives an insight into the value of cross-project prediction in such settings.

Research Question 3: Does cross-project models behave differently at different regions of the AUCEC?

One issue that arises in cross-project prediction is the risk of *over-fitting*; one might reasonably expect that with increasing numbers of predictors, models can increasingly over-fit to the specific characteristics of the training set, and thus perform worse on the test data.

Research Question 4: Does cross project defect performance depend on model size (number of variables)?

4. EXPERIMENTAL METHODOLOGY

4.1 Data Collection

We collected defect data and predictive metrics for several projects. For predictors, we used a set of process metrics that are easily available even for new projects. Our choice of datasets and process metrics considers the following factors.

Metric	Description
Commits	Number of commits made to this file during this release
Active Dev	Number of developers who made changes during this release
Added	Added LOC during this release normalized by file size
Deleted	Deleted LOC during this release normalized by file size
Changed	Changed LOC during this release normalized by file size
Features	Number of new features in this file during this release
Improvements	Number of improvements in this file during this release
Log SLOC	Log source lines of code

Table 1: Metrics Used in the Models

- Prior research found that process metrics are more useful for defect prediction [21]
- Process metrics are easier to explain and act upon. So, *e.g.*, if more active developers are associated with higher

defect proneness of a file, a manager might restrict the number of developers; product metrics, *e.g.,* Halsted or McCabe, are harder to control.

The process metrics are listed in Table 1; all are easily obtained. We have found in prior studies [24] that even *one well-chosen variable* can give good prediction performance. Furthermore, most projects have a relatively small proportion of defective elements; using too many variables in such *class-imbalanced* settings risks over-fitting to the training set.

We extract all of the process related data from the GIT and JIRA repositories of the projects. We downloaded the GIT version control repositories from the Apache GIT website[2]. We used the corresponding JIRA[3] issue management system to identify all the defects that got fixed and the associated git commits that fixed those defects. The development process used in these projects requires that commits made in the GIT repo are explicitly linked to any specific JIRA issues that are addressed in that commit. The JIRA data has high linking rates, and thus generally is of good quality. We crawled the JIRA website and parsed all the issue information for the studied projects. We then extracted individual issue identifiers and the commits that fixed those issues.

4.2 Modeling Defects

We used Logistic regression for modeling defects. Logistic regression is an extension of the linear regression to binary classification, *i.e.* in our case whether a file is defect-prone or not. Rather than predicting a defect count or density, we predict the probability of defect occurrences, scaled as the log of the odds-ratio. Logistic regression is simple, widely applicable, and extensively used in the existing literature on defect prediction. All of our models are built in R[4] with standard generalized linear model and the binomial link function. We deliberately chose a simple, standard, widely-used approach so as to a) to the extent possible, replicate existing work, and b) ensure that our findings do not leverage any exotic, powerful techniques.

Since we were measuring prediction performance, rather than testing hypotheses, issues such as VIF, goodness of fit, variable significance etc were not such a concern. Thus selecting variables to use in our model was relatively straightforward: we simply used all the available variables. As we discuss below, subsequently we analyzed the models in detail to identify the most significant variables, and did indeed find that a very small subset of the variables accounted for most of the performance in both cross- and within-project settings.

4.3 Granularity of Analysis

We predict defects at file level: *viz.*, the entities that our models evaluate are files. We identify a *defective file* post-facto, *if the defect was fixed in that file.* We use defect fixing information of JIRA and GIT to associate a defect with a file. Unless a defect is fixed, we cannot identify the files that are culpable for that defect.

The temporal scope of the analysis is set to release level: if a file contains any defect within a release, we label it defective for that release. Like others [2, 29] we use a project specific release interval derived from the project's GIT repository. Each release cycle is a natural epoch in the process history of a project, and arguably, process metrics captured over such an epoch (*e.g.* number of different committers to a file during a release) will more fully capture the possible behaviors that might influence defect introduction. To identify different releases, we ran the `git tag` command to extract all the tags associated with different releases. Each release ends (or gets released to the customer) at the commit date associated with the GIT tag of that release. We consider the end date of the prior release as the start date of the development of the next release. If we cannot find any git tag prior to a release, we consider the start of project as the start of that release. All the releases that we considered are summarized in Table 2. To evaluate trained models, we only predicted defects during the development phase of each release, and checked if the file was eventually labeled as defective in that release. We did this to emulate a deployment scenario where quality control engineers would run the prediction model regularly to identify potentially defective files and take actions accordingly.

Our choice of file level analysis is motivated by recent findings on the risk of ecological inference by Posnett *et al.* [23]. Package or binary level models have a coarser scope than file level models; thus they may not be as good predictors from a cost-effectiveness perspective, because of ecological inference risk. Moreover, cost-effectiveness at package level would be too coarse as it is hard to prioritize resource for an entire set of packages.

4.4 Cross Vs. Within

We evaluate both within-project and cross-project model performance. When our training release and test release come from the same project, we call it a within-project prediction, otherwise we call it a cross-project prediction. Note, in the within-project setting, we test model performance in a pure *prediction* setting. For example, we use a release i to train up a model m_i; then, m_i is only evaluated for its ability to predict defects in release $i+1$ (Approach 1).

In the cross project setting, we test the prediction performance m_i against *every* release of every other project, regardless of calendar time. The projects we have chosen are quite different and independent, and we can leverage this independence to gain a larger sample of test sets to evaluate performance. We have also replicated our entire results using a second within-project prediction setting where we use release i to predict any release j, where j comes from the same project and $j > i$, i.e., j follows i chronologically (Approach 2). In other words, in Approach 1, we used $j = i + 1$ for within project prediction, while in Approach 2 we used any j released after i. In both settings, we used the same cross-project evaluation, i.e., using every release of every other projects. For brevity, we present the results only from first approach, i.e., j is restricted to $i+1$ in within-project setting. One might reasonably expect that this setting is ideal for within-project prediction, since it's likely that release $i + 1$ is quite close to release i. But all *our findings also hold for Approach 2*, which is surprising.

As the training and test sets are chronologically further apart in an within-project prediction setting, the prediction quality of the within-project model may degrade [7] So, our presented results (Approach 1) take a very conservative within-*vs*-cross comparison.

We evaluate cross-project performance using both the traditional IR F-measure, and also the cost-sensitive measures

[2] `http://git.apache.org`

[3] `https://issues.apache.org/jira`

[4] `http://cran.r-project.org`

Project	Description	Releases	# Releases	Avg # Files	Avg SLOC
Axis2	Web Services/SOAP/WSDL engine	1.5.3, 1.5.4, 1.6.0, 1.6.1	4	2757.25	296173.25
CXF	Services Framework	2.1, 2.2, 2.3.0, 2.4.1	4	3764.25	321919.75
Camel	Enterprise Integration Framework	1.5.0, 2.0.0, 2.5.0, 2.8.2	4	4214.50	220485.75
Cayenne	Object Relational Mapping for Java	3.0, 3.0.1	2	2763.50	198075.50
Derby	Relational Database	10.3, 10.4, 10.5, 10.6, 10.7, 10.8	6	2637.67	548439.17
Lucene	Text Search Engine Library	2.0, 2.2, 2.3.2, 2.4.1, 2.9.1, 3.0.3	6	1005.33	124987.17
OpenEJB	Enterprise Java Beans	3.0, 3.1.1, 3.1.2, 3.1.3	4	2615.50	191040.75
Wicket	Web Application Framework	1.3.0, 1.3.4, 1.3.7	3	1925.67	125702.00
XercesJ	Java XML Parser	2.7.1, 2.8.1, 2.9.1, 2.10.0, 2.11.0	5	789.40	132052.00

Table 2: Studied Projects and Release Information

discussed earlier. Before we proceed to validate and check the answers to our research questions, we wanted to ensure that our dataset yielded results consistent with earlier negative results on cross-project prediction:

Sanity Check: Ensure that with our dataset, the results as far as cross-project and within-project performance, *as measured by the traditional (non-cost-sensitive) approach* is consistent with prior results.

4.5 Evaluating Performance

As discussed in section 2, we use AUC, PRECISION, RECALL, F-MEASURE and AUCEC for comparing model performance. Of these measures, PRECISION, RECALL and F-MEASURE require a cutoff probability. As is found by Mende [18], the choice of such cutoff is arbitrary and makes the replication of existing prediction results difficult. Consequently, in this paper we choose three different cutoff values. In order to conduct the "sanity check" as framed above, we evaluate the models at 0.5 cutoff used in the influential Zimmermann *et al.* [32] paper, and also Shihab *et al.* [25];

To further explore the F-measure performance, we evaluate the model on two additional cutoff settings. The first, denoted TR_MAX, finds the cutoff value that maximizes the F-MEASURE within the training set. This is arguably a reasonable approach when seeking to maximize F-score; one might simply choose the threshold to maximize F-score on the data one has (the training data) in the hope that this also gives good performance on the test data. This approach is also feasible, in that it only uses data available within the training set. We call the PRECISION, RECALL and F-MEASURE at this cutoff $\text{PRECISION}_{tr\_max}$, RECALL_{tr_max} and F_{tr_max}.

Finally, and again for comparison, we computed a best possible (but infeasible) setting: we find the threshold that maximizes the F-MEASURE *on the test set* (both within and cross), and compare this optimal within-project F-MEASURE with this optimal cross-project F-MEASURE. This is clearly infeasible in practice, since the defect in the test set is not known ahead of time; however, it provides us an "outside envelope" optimal value to evaluate the actual f-measure performance.

5. RESULTS

We begin our discussion with our sanity check, to ensure that we are in fact replicating existing findings in terms of the traditional IR-based measures.

Sanity Check: *Do our models have comparable performance to that of the existing literature?* We investigate F-MEASURE at different cutoffs to evaluate our model performance, and compare the results with the existing literature on defect prediction. Figure 2 shows the F-MEASURE for the models built from all the metrics discussed in table 1. Following the suggestion of Menzies *et al.* [20] and the approach of other similar studies [10, 25, 32], we built the models using all the metrics and do not do a parameter selection. As we discussed earlier, computing the F-MEASURE requires an arbitrary cutoff. First, we compute the F-MEASURE at 50% threshold, as was done in prior research [25, 32]; *viz.*, a file is declared defective if the model finds it defective with a probability > 0.5. To probe further, we also compute the F-MEASURE on the test set at the threshold that maximizes the F-MEASURE on the training set [10]. This threshold might be reasonably chosen in practice: if a threshold works well on a training dataset, it might also work well on a new test dataset. Finally, we find an overly optimistic (and unrealistic) cutoff that maximizes the F-MEASURE on the test set. Note, the final approach using the test set to find the cutoff is not feasible in real prediction setting, as we would not have the defect data in the test set to determine the best possible cutoff. We present this as an "outside boundary" calculation to see how the best possible cross-project F-MEASURE compares to the best possible F-MEASURE in within-project. We present the corresponding F-MEASURE defined as F_{50}, F_{tr_max} and F_{te_max} in Figure 2. In general, within-project prediction performance dominates cross-project prediction performance. The disparity is greatest with F_{tr_max}, indicating that choosing the optimal threshold for the training release gives best performance for the other releases in the same project; likewise the (straw-man) F_{te_max} option of choosing the threshold for the test set boosts cross-project performance, but still within-project performance dominates. The F_{50} option gives similar results, also with within-project dominating. These results are strongly consistent with earlier reported such measures in cross and within-project defect prediction scenarios [10, 17, 25, 32].

We also ran *one-sided* Wilcoxon sign rank tests to compare the cross-project and within-project prediction performance. The alternative hypothesis was set to "Cross project performance as measured in F_{50}, F_{tr_max} and F_{te_max} is less than the within project performance". All p-values are adjusted to correct for false discovery using the Benjamini-Hochberg procedure. We present all the p-values in table 3. As is

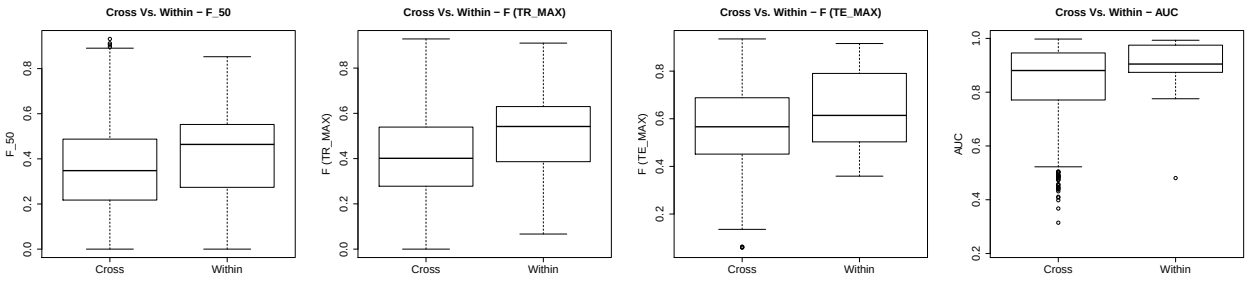

Figure 2: Performance comparison of the models for cross and within-project settings. Note that within-project measures are calculated in a prediction setting, predicting the next release using a model trained on the prior release.

AUC	AUCEC_10	AUCEC_20	AUCECF_10	AUCECF_20	F_50	F (TR_MAX)	F (TE_MAX)
0.020	0.492	0.231	0.061	0.067	0.035	0.004	0.125

Table 3: p-values for cross and within-project models with all variables.

apparent from the table, the F-MEASURE is always lower in the cross-project setting. Other than the straw-man F_{te_max}, which is exaggerated owing to the arbitrary cutoff chosen from the test set oracle, the p-values from the F-MEASURE at different cutoffs are all significant, indicating that the cross-project performance in terms of F-MEASURE is lower than the within-project performance. This corroborates the existing findings on the cross-project literature [17, 26, 31].

Sanity Check: Consistent with prior results, we find that the cross-project defect prediction, when measured in the traditional (non-cost-sensitive) approach, does not perform as well as the within-project defect prediction.

However, as discussed earlier, the F-MEASURE depends on arbitrary cutoffs; and so we turn to AUC as a more refined and comprehensive measure [15]. We note here, however, that with a few exceptions (*e.g.* [17]), existing cross-project defect prediction research does not report AUC. Still, we compare the within-*vs*-cross performance using AUC.

RQ1: *How does the cross-project models perform in terms of* AUC*?*

Figure 2 presents the AUC for cross and within-project prediction settings. Again, we keep all the predictors of table 1 in the models. Clearly, the cross-project AUC is significantly lower than within-project AUC; a two-sample Wilcoxon test of the alternative hypothesis: "cross-project AUC is lower than within-project AUC " provides statistical confirmation. (see table 3).

Next, we compare within-*vs*-cross prediction performance in a cost-sensitive setting, where we are concerned solely with the top n% of the files or SLOC, where the n is determined by the resource availability. The SLOC measure is suited for inspection tasks, where work load clearly depends on SLOC count; the file measure might be well-suited for coverage testing or design reviews, where the workload depends on an abstracted view of the code. So, we next evaluate the models' performance at two critical values, $n = 10$ and $n = 20$ [1].

RQ2: *Does cross-project defect prediction work in terms of cost-effectiveness?*

We present our findings in figure 3, comparing AUCEC at 10% and 20% SLOC and File. For SLOC based AUCEC, the cross-project defect prediction works surprisingly well and the boxplots look very similar. This visual impression is confirmed by a two-sample Wilcoxon sign-rank test, comparing the AUCEC scores at 10% and 20% of SLOC and Files: corrected p-values are shown in table 3 (AUCECF designates file based AUCEC). We fail to reject the null hypothesis "within-project AUCEC is bigger than cross-project AUCEC." This suggests that *cross*-project defect prediction is in general no less useful a way to focus code-inspection effort than *within*-project defect prediction. The results for file-based AUCEC are disappointing: even at 10% of files, the file-based within-project AUCEC prediction outperforms the cross-project counterpart. Further investigation of this (as we discuss later) suggests a possible explanation: our models tend to predict more defect-dense files. Even so, because of the skewed distribution of file size, the larger files tend to have a disproportionate number of defects, and this affects cross-project prediction performance. We repeated the experiment at 5% of SLOC and Files, and found similar results. This leads to the following surprising conclusion:

RQ2: In terms of AUCEC the cross-project defect prediction performs surprisingly well.

If we look carefully at table 3 it seems that as we increase the percentage of SLOC or files, the cross-project AUCEC becomes less competitive. This inspired us to find out the range of operating points where we can expect to see a feasible AUCEC performance from the cross-project prediction. To answer this question, we compare AUCEC at a wide range of cost factors.

RQ3: *Do cross-project models behave differently at different regions of* AUCEC*?*

We computed all within- and cross-project prediction performances at 20 cost settings, ranging from 5% of SLOC (or Files) to 100% of SLOC (or Files) at 5% increments. For each cross- and within-project training-prediction pairs, we get one sample at each cost setting. At each cost setting, we

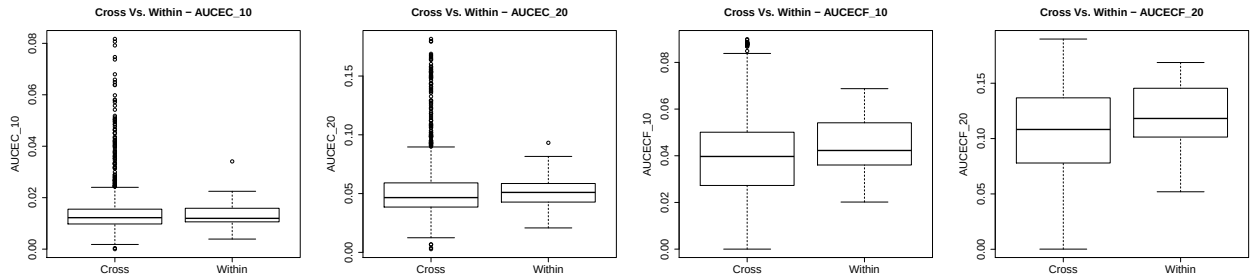

Figure 3: AUCEC **at 10% and 20% SLOC and File**

can calculate these summary statistics for the AUCEC values of all prediction pairs: the min, first quartile, median, second quartile and max. We obtain these from the R boxplot command, which ignores outliers before computing the min and max. We can then plot the variation of these summary statistics with the cost setting. For brevity, we only show the median plots here and discuss others in the text.

Figure 4 shows the smoothed line plot for both %SLOC and %File. In addition to showing the cross and within-project prediction models' AUCEC performance, we also for comparison show an *optimal* curve and a *random* baseline curve. The optimal performance is the theoretically maximum AUCEC attainable, and is calculated by ordering the files in the test set by decreasing defect density, breaking ties using file SLOC (smaller files are considered first). Certainly the optimal will still be lower than 1.0, as attaining 1.0 would require finding 100% of the defects at 0% SLOC. The optimal line is marked as "Optimal" which would be same for both cross and within project, as in both cases the evaluation is done on the same test set. We also plot the random prediction using the formula $\frac{(cut)^2}{2}$, where *cut* is the cutoff. This is the straw-man scenario where defects are distributed uniformly in the code, and we find n% of defects after inspecting n% of SLOC. In expectation, over a large number of trials, randomly choosing files to inspect will give us such a curve. The purpose of the random and optimal lines is to give the reader an idea on the effectiveness of the cross and within-project prediction.

The plot clearly indicates how cross-project performance hues very closely to the within-project performance. In the SLOC based AUCEC figure, when considering all the files (at 100% cutoff) the cross-project prediction has a median AUCEC within the range of $3-4\%$ of the within-project prediction and both do very well with an AUCEC area close to 80%. While the file-based AUCEC has understandably higher AUCEC for both cross-project and within-project prediction, they are still very close to each other. In both cases the models perform well over the random line. Moreover, when we operate within a small cutoff at SLOC level, such as 20% of SLOC, the line plots are essentially identical, and overlapping which means *we have nothing to lose, in terms of* AUCEC (!) if we port a within-project model to a cross-project setting. We also observed similar performance for other line plots using min, first quartile, third quartile and max. This suggests that the cross-project defect prediction works reasonably well for prioritizing resource allocation.

RQ2 & RQ3: In terms of AUCEC the cross-project defect prediction performs surprisingly well for a wide range of cost settings, and is as helpful as within-project prediction, for both SLOC-based and file-based cost factors.

We are using a number of variables for cross-project prediction; Does model size matter? Some prior research has considered this. In a within project setting, Wang *et al.* found that the models reach a performance ceiling only after four variables [28]. We try to identify the impact of many variables for cross-project prediction models. We hypothesize

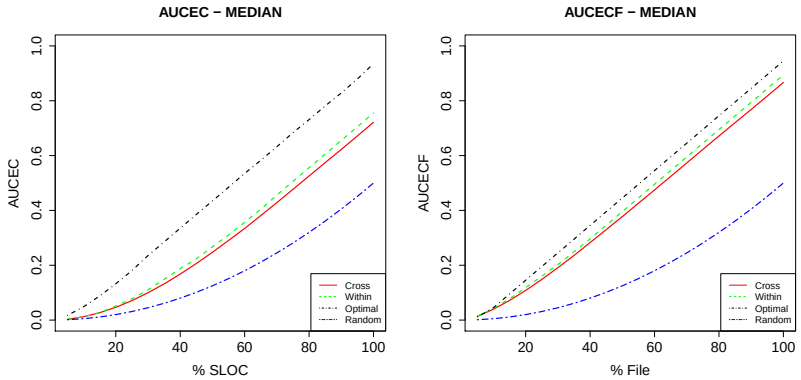

Figure 4: AUCEC **at different %SLOC and %File**

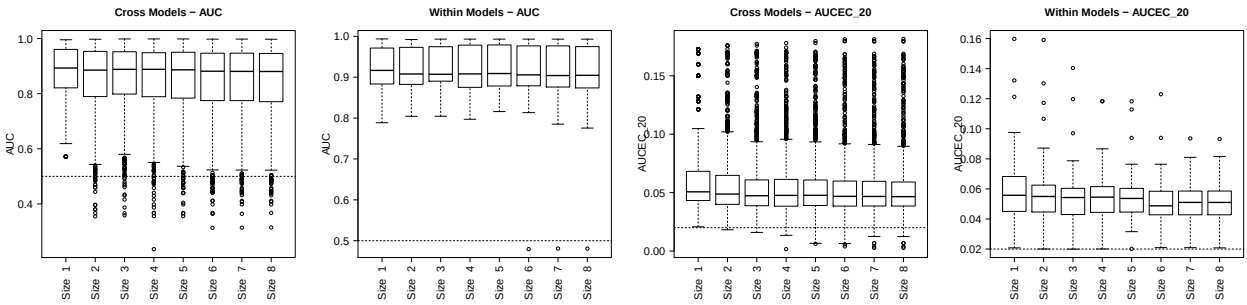

Figure 5: Comparing cross and within-project models of different sizes, optimized per training set.

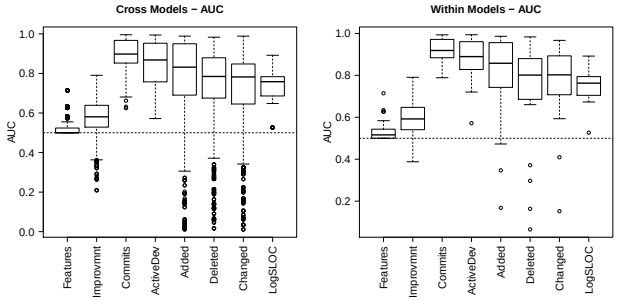

Figure 6: AUC **for cross and within-project single variable models.**

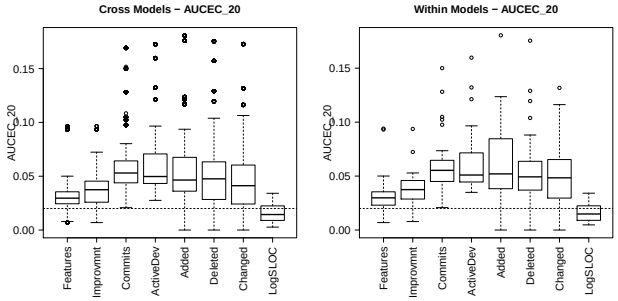

Figure 7: 20% SLOC based AUCEC **for cross and within-project single variable models.**

that as the number of variables increases, the models will tend to overfit more frequently, and therefore the portability of such models across projects would diminish.

RQ4: *Does the cross-project model performance depend on the model size?*

We start with selecting the best possible size k models, where $1 <= k <= n$ and n is the total number of metrics available (listed in table 1). We used the AIC (Akaike Information Criterion[5]) score to select the best size k model for each training set. Note, a training set may be used to predict the subsequent release of the same project (within-project setting) or to predict other projects' releases (cross-project setting). This gives us a total of $(n * r)$ trained models, having up to $1 \ldots n$ variables, where r is the number of distinct releases across all projects. We then test each of those models in cross and within-project setting. Note, we evaluate such models on all releases of other projects, while restricting the within-project evaluation to the immediately following release of the same project. Also note, at each size setting k, the actual set of variables chosen for best AIC may be different for different training data sets.

Figure 5 plots AUC and AUCEC at 20% SLOC for different model sizes. Two-sample Wilcoxon test for each possible model size k (with alternative hypothesis set to "Cross-project defect prediction has lower performance than the within-project prediction", and p-values corrected) rejects the null hypothesis for F-MEASURE for different cutoffs and for AUCEC at file level. However, we failed to reject the null hypothesis for %SLOC based AUCEC. All the p-values were high for 10% and 20% of SLOC, and hovered around 0.52 for 10% and 0.23 for 20%. This suggests that the *cross-project %SLOC based* AUCEC *is resilient to parameter selection for a reasonable cutoff choice.*

Figure 5 also shows that just a single variable model performs very well in both cross and within-project defect prediction for our dataset. Also, noticeable is the monotonic decrease of performance of the cross-project models as we add more variables. Though the within-project performance is also affected by increasing number of variables, the effect is less dramatic. For AUCEC at 20% SLOC, the model is more size invariant, which further corroborates the resilience of cross-project prediction performance in terms of cost-effectiveness.

Discussion To gain some understanding into the variables that were actually contributing to the good quality of the cross-project AUCEC performance, we built 8 single variable models per training dataset, one for each of our metrics. The corresponding 8 distinct "singleton" models' AUC is presented in figure 6, and the AUCEC in 7. Note the dashed random lines: they indicate the expected performance of a model that makes uniformly random predictions. In the case of AUC the expected random performance gives a value of 0.5; in the case of AUCEC at 20%, the expected random performance is 0.02. This is because when we inspect 20% of the lines under random ordering, we can expect that in the general case, we will find 20% of the defects. This gives us an area under the curve of 0.02 (using $\frac{1}{2}$ * 0.2 * 0.2). Horizontal lines in figures 6, 7 indicate the random performance.

The variation in singleton model performance indicates that the different variables have different explanatory power. Modulo multi-collinearity effects, these plots are suggestive of their potential contribution to the multi-variate model

[5] `http://en.wikipedia.org/wiki/Akaike_information_criterion`

performance. This variation *per se* is not that surpring, and has been observed in most modeling settings. However, the comparison of AUC and AUCEC plots, specially considering the random lines, reveals that the last variable, $Log(SLOC)$ is a special case.

While $Log(SLOC)$ gives good AUC performance, well above the random line, its AUCEC performance is **worse** than random guessing, in both within-project prediction and cross-project prediction. This bears further discussion. Using $Log(SLOC)$ for prediction essentially amounts to the belief that bigger files contain more defects, *viz.*., the defect density stays the same or increases as file size increases. The good performance on AUC suggests that bigger files do tend to have more defects than smaller files; however, the poor performance on AUCEC, when compared to the good performance for AUC suggests that larger files don't have higher defect density. This finding is consistent with [22].

However, this points to a potential issue in previous cross-project defect prediction approaches: there is a considerable body of evidence (*e.g.* [8, 11]) suggesting that size is *highly* correlated with most product metrics, including both the classical McCabe and Haslted metrics, and more modern OO metrics.

> *Our results suggest that using product metrics, which are usually correlated with size, might actually give worse cross-project performance for cost-sensitive cross-prediction models.*

6. THREATS TO VALIDITY

Using process data Other than $Log(SLOC)$, all of our models use process metrics. It's possible that process metrics are unavailable, and this a threat to our conclusions. We note, however, that the set of metrics we used are readily obtainable within current release, from version control and issue trackers. Thus we used current release variables (added, deleted, etc) and current issue tracker measures (Number of improvements and features) but avoided *post-facto* information such as number of defects (which is known to be highly effective [24]). We also avoided historical data such as past feature count, past improvement count, past number of developers, developer experience etc.

Project newness Another construct validity issue is that none of our projects are new, and thus our cross-project predictions are not tested on brand-new projects; we argue that this issue is mitigated by two factors a) the projects are quite different in application domain and b) we only use current-release data in any project both for training and test, so the data sets are "memory-less". We also note that our approach is consistent with prior experiments in cross-project prediction [10, 17, 26, 31].

Project Diversity Another issue is the diversity of our projects. All are Java langauge Apache projects; however, they do have very diverse sets of developers, arise from very diverse application domains. Thus, we believe our sample set is diverse enough to support a claim of generalizability. We also note that our data quality is very high, because these projects use a high-fidelity process to link GIT commits to closed bugs in the JIRA issue tracker. We did a data quality study and found that such structured and organized linking yields a very high quality data, and in our training data sets we observed a typical defect linking rate of more than 80%. This means, more than 80% of the resolved defects got linked to the files where the defects were fixed. This is a much higher linking rate than reported (typically under 50%) in the literature [4] using traditional heuristic based linking technique [6, 9]. Our study therefore suffers less from the threats to validity of false-positive and false-negative to identify the defective files.

However, all projects are indeed open-source projects developed under the Apache Software Foundation (ASF) process guidelines. It's certainly possible that our findings will not generalize to commercial projects. However, the processes used in ASF are not atypical of OSS projects, and our results may apply in other OSS projects. We hope that researchers with access to commercial data will replicate our study of the cost-effectiveness of cross-project prediction in commercial settings.

7. CONCLUSION

Prior results suggest that cross-project defect prediction is significantly worse than within-project prediction. We mounted a *cost-sensitive* analysis of the efficacy of cross-project defect prediction, comparing it with the within-project defect prediction. For 9 large ASF projects consisting of 38 releases, we found that when using the measures deployed in prior research, our findings were consistent with reported results: cross-project performance is significantly worse. However, we found that cost-sensitive cross-project prediction is a) as good as within-project prediction, and b) substantially better than what we would expect under a random model. Thus, our major finding is this:

> In terms of AUCEC, cross-project defect prediction performs surprisingly well and may have a comparable performance to that of the within-project models.

References

[1] E. Arisholm, L. C. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *ISSRE*, pages 215–224. IEEE Computer Society, 2007.

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

[3] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: Bugs and bug-fix commits. In *FSE*, pages 97–106. ACM, 2010.

[4] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th FSE*, pages 121–130. ACM, 2009.

[5] L. C. Briand, W. L. Melo, and J. Wüst. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE TSE*, 28(7):706–720, 2002.

[6] D. Cubranić and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 408–418, Portland, Oregon, 2003. IEEE Press.

[7] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein. Tracking concept drift of software projects using defect prediction quality. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 51–60, Washington, DC, USA, 2009. IEEE Computer Society.

[8] K. El Emam, S. Benlarbi, N. Goel, and S. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE TSE*, 27(7):630–650, 2001.

[9] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03*, page 23, Washington, DC, USA, 2003. IEEE Computer Society.

[10] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang. An investigation on the feasibility of cross-project defect prediction. *Autom. Softw. Eng.*, 19(2):167–199, 2012.

[11] T. Khoshgoftaar and J. Munson. Predicting software development errors using software complexity metrics. *Selected Areas in Communications, IEEE Journal on*, 8(2):253–261, 1990.

[12] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *ICSE'2011*, pages 481–490. IEEE, 2011.

[13] S. Kim, T. Zimmermann, E. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th ICSE*, pages 489–498. IEEE Computer Society, 2007.

[14] A. G. Koru and H. Liu. Identifying and characterizing change-prone classes in two large-scale open-source products. *Journal of Systems and Software*, 80(1):63–73, 2007.

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

[16] Y. Ma and B. Cukic. Adequate and precise evaluation of quality models in software engineering studies. In *Proceedings of the 29th ICSE Workshops*, ICSEW '07, pages 68–, Washington, DC, USA, 2007. IEEE Computer Society.

[17] Y. Ma, G. Luo, X. Zeng, and A. Chen. Transfer learning for cross-company software defect prediction. *Information and Software Technology*, 54(3):248–256, 2012.

[18] T. Mende. Replication of defect prediction studies: problems, pitfalls and recommendations. In T. Menzies and G. Koru, editors, *PROMISE*, page 5. ACM, 2010.

[19] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok. Local vs. global models for effort estimation and defect prediction. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 343–351. IEEE, 2011.

[20] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE TSE*, 33(1):2–13, 2007.

[21] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE*, pages 181–190. ACM, 2008.

[22] T. Ostrand and E. Weyuker. The distribution of faults in a large industrial software system. In *International Symposium on Software Testing and Analysis: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis: Roma, Italy*. Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,, 2002.

[23] D. Posnett, V. Filkov, and P. Devanbu. Ecological inference in empirical software engineering. In *ASE'2011*, pages 362–371. IEEE, 2011.

[24] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. Bugcache for inspections: hit or miss? In *Proceedings of the 19th ACM SIGSOFT FSE*, pages 322–331. ACM, 2011.

[25] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan. Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In G. Succi, M. Morisio, and N. Nagappan, editors, *ESEM*. ACM, 2010.

[26] 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 Softw. Engg.*, 14(5):540–578, Oct. 2009.

[27] B. Turhan, A. T. Misirli, and A. B. Bener. Empirical evaluation of mixed-project defect prediction models. In *EUROMICRO-SEAA*, pages 396–403. IEEE, 2011.

[28] H. Wang, T. M. Khoshgoftaar, and N. Seliya. How many software metrics should be selected for defect prediction? In R. C. Murray and P. M. McCarthy, editors, *FLAIRS Conference*. AAAI Press, 2011.

[29] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *ESE*, 13(5):539–559, 2008.

[30] H. Zhang. On the distribution of software faults. *IEEE TSE*, 34(2):301–302, March-April 2008.

[31] 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 H. van Vliet and V. Issarny, editors, *ESEC/SIGSOFT FSE*, pages 91–100. ACM, 2009.

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