Revisiting Common Bug Prediction Findings Using Effort-Aware Models

Yasutaka Kamei†, Shinsuke Matsumoto††, Akito Monden‡,
Ken-ichi Matsumoto‡, Bram Adams† and Ahmed E. Hassan†
†*Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University,*
††*Graduate School of Engineering, Kobe University,*
‡*Graduate School of Information Science, Nara Institute of Science and Technology,*
†{*kamei, bram, ahmed*}*@cs.queensu.ca,* ††*shinsuke@cs.kobe-u.ac.jp*
‡{*akito-m, matumoto*}*@is.naist.jp*

***Abstract*—Bug prediction models are often used to help allocate software quality assurance efforts (e.g. testing and code reviews). Mende and Koschke have recently proposed bug prediction models that are effort-aware. These models factor in the effort needed to review or test code when evaluating the effectiveness of prediction models, leading to more realistic performance evaluations. In this paper, we revisit two common findings in the bug prediction literature: 1) Process metrics (e.g., change history) outperform product metrics (e.g., LOC), 2) Package-level predictions outperform file-level predictions. Through a case study on three projects from the Eclipse Foundation, we find that the first finding holds when effort is considered, while the second finding does not hold. These findings validate the practical significance of prior findings in the bug prediction literature and encourage their adoption in practice.**

I. Introduction

Software quality assurance activities (e.g., source code inspection and unit testing) are becoming increasingly important as software systems are being widely used in our society. Software faults (or bugs) in released products have expensive consequences for a company and affect its reputation. Since a company has only limited resources (e.g., developers and cost) for software quality assurance activities, these activities have to be performed as efficiently as possible.

To prioritize quality assurance efforts, fault prediction techniques are often used to prioritize modules based on their probability of having a fault or the number of expected faults [1]–[3]. With these models, practitioners can allocate limited testing or reviewing efforts to the most fault-prone modules.

However, as pointed out by Mende and Koschke [4], traditional prediction models [5]–[10] typically ignore the effort needed to fix bugs, i.e., they do not distinguish between a predicted bug in a small module and a predicted bug in a large module. Clearly, both bugs require a different amount of effort to inspect and fix, yet both are considered equal when measuring the effectiveness of prediction models.

Mende and Koschke [11] proposed effort-aware models that include the notion of effort. In their study, they use lines of code as a proxy for effort. An experimental result using publicly-available data sets shows that the prediction performance of effort-aware models improved from a cost-effectiveness point, compared to no effort-aware models (i.e., traditional prediction models).

Since effort-aware prediction models offer a totally new interpretation and the practical adoption-oriented view of bug prediction results, it is necessary to reconsider some of the major findings in the fault prediction literature (e.g., [12]–[15]) by taking into account effort. In particular, we are interested in addressing the following two research questions:

RQ1 *Are process metrics still more effective than product metrics in effort-aware models?*
It is a known fact that process metrics are more efficient fault predictors than product metrics [12]–[14]. Since, Mende and Koschke [11] only used product metrics, in their effort-aware models, it is not clear whether process metrics still outperform product metrics when considering effort.

RQ2 *Are package-level predictions still more effective than file-level predictions?*
Traditionally, package-level bug prediction models have been shown to have higher precision and recall than file-level bug prediction models [15], [16]. We examine whether or not effort-aware models at the package-level are still as effective.

This paper provides the following contributions:

- We show that process metrics still outperform product metrics using effort-aware bug prediction models.
- We show that package-level predictions are not more effective than file-level predictions. This finding holds even when considering Martin's package design metrics [17].
- We show that the effectiveness of package-level predictions can improve if we perform our predictions at the file-level then lift it to the package-level instead of collecting all metrics at the package-level. However this new model still does not outperform file-level predictions when considering the quality assurance efforts.

In what follow, Section II introduces related work. Section III provides the design of our experiment, and Section IV gives the results. Section V discusses the performance differences

26th IEEE International Conference on Software Maintenance in Timişoara, Romania
978-1-4244-8628-1/10/$26.00 ©2010 IEEE

between package-level and file-level predictions. Section VI presents the threats to validity and Section VII summarizes the paper.

II. Related Work

In this section, we discuss related work in fault prediction models and effort-aware models.

A. Fault prediction models

Many fault prediction models have been proposed in literature [8]–[10], [18]–[20]. Zimmermann and Nagappan [10] have introduced the use of network analysis on dependency graphs for fault module prediction. They conducted an experiment using industrial datasets. The result of their experiment showed that network measures could identify 60% of the files developers considered as critical. Mizuno and Kikuno [8] applied a generic text discriminator (i.e., as Spam filter) to predict faults. They showed that their approach could classify 78% of the actual faulty files as fault-prone. Kim et al. [18] proposed a new bug prediction technique that works at the granularity of an individual file level change. Their classifier is trained using features (e.g., terms in the added delta source code and terms in the change log) extracted from a version archive such as CVS. They showed that their approach could classify the files as buggy or not, with a 78 percent accuracy. Ratzinger et al. [19] introduced non-refactoring and refactoring related features to predict faults with high performance. Our study evaluates the effect of well-known fault prediction models at the package-level instead of at the file-level.

There are also several studies on evaluation methods for prediction models [1], [4], [21], [22]. Lessmann et al. [21] proposed a framework for comparative software fault classification (i.e., fault-prone or not). They considered three sources for bias, such as, relying on accuracy indicators that are conceptually inappropriate for software fault prediction and cross-study comparisons. The AUC (Area Under the receiver operating characteristics Curve) was recommended as the primary accuracy indicator for comparative studies. More recently, Mende and Koschke [4] introduced a method for comparative fault density prediction based on the cost of quality assurance activities. In our study, the experiments for our two research questions are evaluated based on Mende and Koschke's method.

Some studies evaluated the performance of fault prediction models for package-level modules. Nagappan et al. [23] predicted the likelihood of post-release faults at the package-level using a regression model and principal component analysis. Schröter et al. [15] also predicted the number of faults at the package-level. In contrast to those studies, we evaluate the effect of prediction models on the costs of software quality assurance activities. Also, while these studies used a single technique for the prediction of package, such as lifting the file-level metrics up to the package-level, our study uses three techniques, in order to study the impact of these techniques on our findings.

B. Effort-Aware Models

Many fault prediction studies evaluate the performance of prediction models that classify a module into fault-prone or not fault-prone [5], [7], [8], [10]. Test managers and quality managers identify fault-prone modules using a prediction model and allocate more test efforts to the modules that are detected to fault-prone.

However, the prediction model would not be effective because the costs of quality assurance activities are largely ignored. Conventional studies evaluate the prediction performance based on the assumption that the effort of test/inspection is the same across modules. The assumption is rarely true in many cases. Arisholm et al. [24] pointed out that the effort of testing or reviewing a module is roughly likely to be proportional to the size.

This paper evaluates the performance of prediction models in terms of effort as inspired by Mende and Koschke's study [11]. Their model considers the effort required to review a module (i.e., file or package) and predicts the relative risk $R_{dd}(x)$ of a module:

$$R_{dd}(x) = \frac{\#errors(x)}{E(x)}, \tag{1}$$

where $\#errors(x)$ is the number of errors in a module and $E(x)$ is defined as the effort required to test or inspect a module x. In this study, we use the lines of code as a measure of effort, similar to Mende and Koschke [11].

We use the P_{opt} evaluation metrics [4] to evaluate the prediction performance of models. P_{opt} is defined as the area Δ_{opt} between the LOC-based cumulative lift charts of the optimal model and the prediction model (Figure 1). In the optimal model, all modules are ordered by the decreasing actual fault density. While in the predicted model, all modules are ordered by decreasing predicted fault density. As shown in the following equation, a larger P_{opt} value means a smaller difference between the optimal and predicted model.

$$P_{opt} = 1 - \Delta_{opt} \tag{2}$$

However, the minimum value of P_{opt} depends on the number of bugs in our dataset. In this study, we used a normalized value based on the following equation:

$$Norm(P_{opt}) = \frac{P_{opt} - min(P_{opt})}{max(P_{opt}) - min(P_{opt})}, \tag{3}$$

where $max(P_{opt})$ and $min(P_{opt})$ are calculated on the LOC-based cumulative lift chart in which all modules are ordered by fault density.

III. Experiment Setting

In this section, we describe our experiment setting. We describe the modeling techniques and the data sources. We explain the recovery of bugs from software repositories.

TABLE I
STATISTICS SUMMARY OF STUDIED PROJECTS

	ver.	# of files (*.java)	# of packages	Source lines of code (KSLOC)	# of faults	Fault density (Faults/KSLOC)
Platform	3.0	4,629	376	611	3,030	4.96
	3.1	5,462	439	738	3,191	4.32
	3.2	6,512	552	883	2,571	2.91
JDT	3.0	2,961	173	471	1,343	2.85
	3.1	3,328	185	574	1,403	2.44
	3.2	3,908	234	673	638	0.95
PDE	3.0	723	56	90	156	1.73
	3.1	873	64	114	73	0.64
	3.2	1,105	76	145	294	2.03

A. Used Modeling Techniques

We use three well-known modeling techniques; regression model [25], regression tree [26] and random forest [27]. We used the statistical computing and graphics toolkit R [28] and its `MASS`, `rpart` and `randomForest` libraries to build the three models.

B. Used systems

The target of our study is three subprojects within the Eclipse software system, one of the best-known open development platforms. We collected module (i.e., file and package) datasets from three versions (v.3.0, v.3.1 and v.3.2) in each subproject (Platform, JDT and PDE) from the Eclipse CVS repository. We consider a Java file as file-level and a *package* as package-level. For example, in case of /org/eclipse/jdt/core/ElementChangedEvent.java, the file is ElementChangedEvent.java and the package is org.eclipse.jdt.core. Table I summarizes the statistics of the used datasets.

C. Recovery of bugs

To obtain the number of bugs in source code files, we implemented the SZZ algorithm [29]. This algorithm identifies when a bug was injected into the code and who injected it by linking a version archive (such as CVS) to a bug tracking system (such as Bugzilla). The SZZ algorithm basically consists of three steps. First step, identifying the commit that fixes a bug, SZZ searches for keywords such as "Fixed" or "Bug" in the CVS comments. We used "bug", "fix", "defect" and "patch" as keywords and identified the commit that had the keyword and a digit number (e.g., bug 12345) as a bug-fix commit. The second step confirms whether that commit is really a bug-fix commit using information from Bugzilla. We link the digit number of the CVS comments to the bug number of Bugzilla. The commit is more likely to be a bug-fix commit if the author of the commit has been assigned to the identified bug report. Other, heuristics to confirm whether a commit is really a bug-fix commit are discussed where [29]. The third step identifies when the bug is introduced, we use the CVS *diff* and *annotate* command (Figure 2). We locate fixed lines (e.g., line #3) of the bug-fix commit (rev. C) and original lines (e.g., line #3) of the previous commit (rev. B) using the *diff* command. We identify the most recent revision (rev. A) in which the original lines were changed using the *annotate* command. We consider the commit of the identified revision as the bug-introducing commit.

When we identify the dates of the introduction and fixing of a bug using the SZZ algorithm, we count that file A has one bug in v3.1 but file B has no bug in v3.1 because the bug of file B is already fixed before v3.1 (Figure 3). We used the Eclipse CVS repository and Eclipse Bugzilla reports provided by the MSR 2008 Mining Challenge [30].

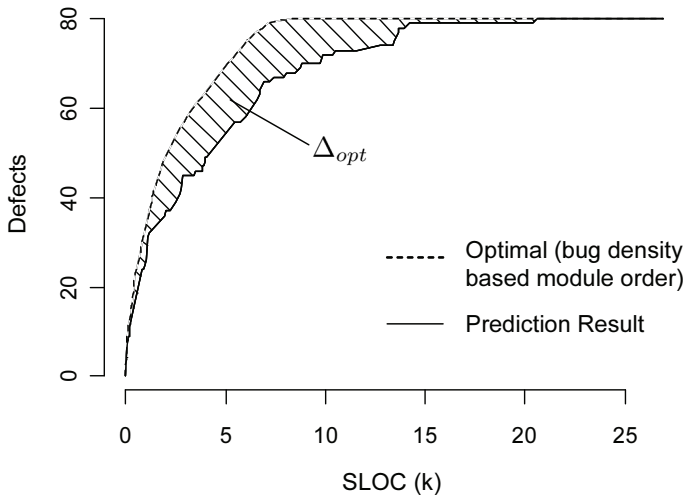

Fig. 1. Example of LOC-based Cumulative Lift Chart.

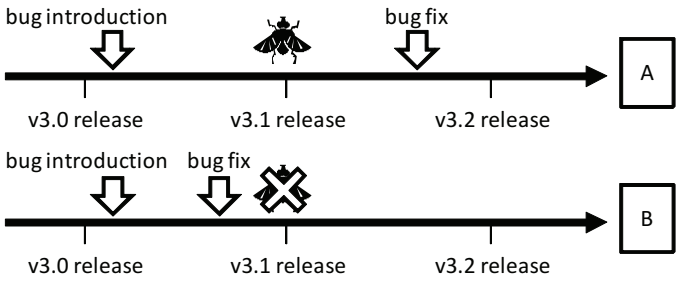

Fig. 3. Example of Counting the Number of Bugs.

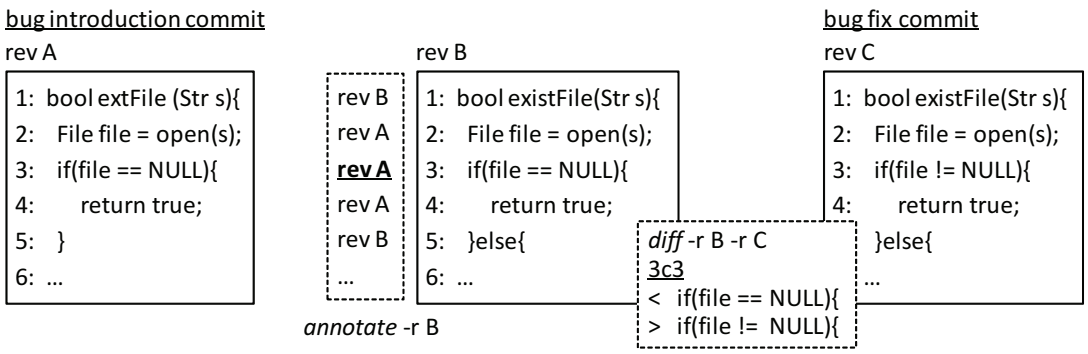

Fig. 2. Example of bug-fix and bug-introduction commits.

IV. Experimental Results

The goal of our experiment is to study the effect of effort-aware models on prior findings in the bug prediction literature. We now present the results of our study with respect to our two research questions.

RQ1: Are process metrics still more effective than product metrics in the effort-aware models?

Overview. In the experiment for RQ1, we compare models based on product metrics to models based on process metrics at the file-level. We apply these metrics to three well-known modeling techniques (regression model, regression tree, and random forest) and evaluate the prediction performance. We also study the performance of the product and process metrics when combined together.

Motivation. Several studies continue to show that process metrics are more effective than product metrics in predicting faulty modules [12]–[14]. Since, Mende and Koschke [11] only used product metrics, in their effort-aware models, it is not clear whether process metrics still outperform product metrics when considering effort.

Used Metrics. For our file-level analysis, we measure product metrics and process metrics (Table II). The product metrics measure the static structure of source code such as source lines of codes and McCabe's cyclomatic complexity. The product metrics are measured using the Eclipse Metrics plug-in [31].

For process metrics, we used the metrics proposed by Moser et al. [14]. The process metrics measure the change history of source code, such as the number of revisions and the number of times a file has been refactored. We wrote a script that calculates the process metrics from the change history (i.e., the Eclipse CVS repository).

Since many product and process metrics have a strong correlation with SLOC, we normalized metrics with a correlation coefficient that is higher than 0.4 by dividing by SLOC [32]. The normalized metrics were MLOC, NBD, PAR, VG, NOF, NOM, WMC, Codechurn, LOCAdded, LOCDeleted and Revisions.

Model Building Approach. In order to address our research questions, we build fault prediction models that are based on product and process metrics. We use three modeling techniques (regression model, regression tree and random forest) to build the prediction models.

TABLE II
Measured Metrics at the File-level

	Metrics name	Definition
Product metrics	SLOC	Source Lines of Codes
	MLOC	LOC executable
	PAR	Number of parameters
	NOF	Number of attributes
	NOM	Number of methods
	NORM	Number of overridden methods
	NSC	Number of children
	NSF	Number of static attributes
	NSM	Number of static methods
	NBD	Nested block depth
	VG	Cyclomatic complexity
	DIT	Depth of Inheritance Tree
	LCOM	Lack of Cohesion of Methods
	WMC	Number of Weighted Methods per Class
	SIX	Specialization Index (NORM+DIT)/NOM
Process metrics	Codechurn	Sum of (added lines of code - deleted lines of code)
	LOCAdded	Sum over all revisions of the lines of code added to a file
	LOCDeleted	Sum over all revisions of the lines of code deleted from a file
	Revisions	Number of revisions of a file
	Age	Age of a file in weeks
	BugFixes	Number of times a file was involved in a bug-fix transaction
	Refactorings	Number of times a file has been refactored

Results of our Experiment. We performed two types of evaluation analyses to study the performance of our experiments. The two analyses are: cross validation and cross-release prediction of post-release failures. For the cross-validation analysis, we randomly divided our module (i.e., files or packages) dataset into two sets with equal sizes. One of the datasets was used as training and the other was used as test. This division was repeated 20 times based on the experimental result by Kirsopp and Shepperd [33]. For the cross-release analysis, a training dataset was built from a past release of a project, and a test dataset was built from the following release. The cross-release evaluation leads to an evaluation in a more practical setting.

Cross-validation analysis

Table III shows the experimental results of our cross validation analysis at the file-level. The values in each row are the average value of P_{opt} for 20 iterations. A "*" symbol next to a value indicates that it is the best performance among the combinations of the three metric types (product metrics, process metrics and both) and three modeling techniques (LM - linear model, RT - regression tree and RF - Random Forest). The value next to the "LOC" in the header of a table indicates the P_{opt} of a LOC based file order, which is the simplest classifier that orders files just by their decreasing size (LOC). In other words, if one were to just review files by picking the biggest files to review first then proceeding to smaller files. The value beside the 'LOC" is used as the baseline for the models.

We find that regardless of modeling techniques (LM, RT and RF), using process metrics is better than using product metrics. In the case of RF, the P_{opt} range of process metrics is from 0.80 to 0.94, while that of product metrics is from 0.56 to 0.75. Among the models, the improvement of P_{opt} by using process metrics is 0.07 at minimum (in PDE v3.0, RF) and 0.35 at maximum (in Platform v3.2, RF). The prediction performance of a combined model of product metrics and process metrics shows no difference from using only process metrics. We also note that using product metrics is better than LOC based file order. In short, process metrics outperform product metrics and combing both types of metrics does not lead to an improved performance.

Cross-release analysis

Table VII shows the experimental results for the cross-release analysis at the file-level. The values in each row are the value of P_{opt} for just a single iteration unlike the cross validation study. Similar to the experimental results of the cross validation study, process metrics show a better prediction performance than product metrics across all datasets. The improvement of P_{opt} using process metrics is 0.01 at minimum (in PDE v3.0, RT) and 0.43 at maximum (in PDE v3.0, LM). For product metrics, LM and RT are worse in some cases than the baseline (LOC based file) order, but RF is always better than LOC based file order. For process metrics, RF gives the best performance among three techniques for all datasets except PDE v.3.0 $\rightarrow$ v.3.1. Again, we find that process metrics outperform product metrics and combing both types of metrics does not lead to an improved performance.

Figure 4 shows the LOC-based cumulative lift charts of the random forest in Platform v.3.1 $\rightarrow$ v.3.2 of Table IV. We order all files by decreasing fault density. The x-axis shows the cumulative SLOC and the y-axis shows the cumulative number of bugs. The dashed line plots the cumulative lift chart for files ordered by decreasing actual (i.e., optimal) fault density, and solid line and dot-line plot the cumulative lit chart for files ordered by decreasing the predicted fault density of random forest using process metrics and product metrics. This result indicates that when we conduct a test on only 20% (of the lines) of all files based on the predicted fault density, we could detect 29% of all faults using product metrics and 74% of all faults using process metrics. That is, using our bug predictions, we only need to spend 20% of the efforts that it would take to test all files to detect up to 74% of all faults. We find that process metrics outperform product metrics by a factor of 2.6(=74/29) when considering effort.

The performance of our product metrics are consistent with results reported earlier by Mende and Koschke's [11]. They showed that the result of random forest using only product metrics was 35% in Eclipse v.3.0.

> Process metrics outperform product metrics as predictors of fault density when taking test effort into account.

RQ2: Are package-level predictions more effective than file-level predictions?

Overview. In the experiment for RQ2, we compare file-level predictions to package-level predictions. We compare the prediction performance of one file-level prediction against three types of package-level predictions. We use three modeling techniques (i.e., LM, RT and RF) to build these three package-level predictions.

Motivation. Traditionally, package-level bug prediction models have been shown to have higher precision and recall than file-level bug prediction models [15], [16]. We clarify whether or not effort-aware models at the package-level module are still more effective.

Used Metrics. For package-level predictions, in addition to product metrics and process metrics, we use the metrics suite proposed by Martin [17]. Table V shows the Martin metrics used in our study. Martin's package design metrics indicate instability (Ca, Ce, I) and abstractness (NA, NC, A) of packages, and imbalance (D) of the instability and the abstractness. In this paper, we consider that a highly instable package is likely to have many bugs because when classes that

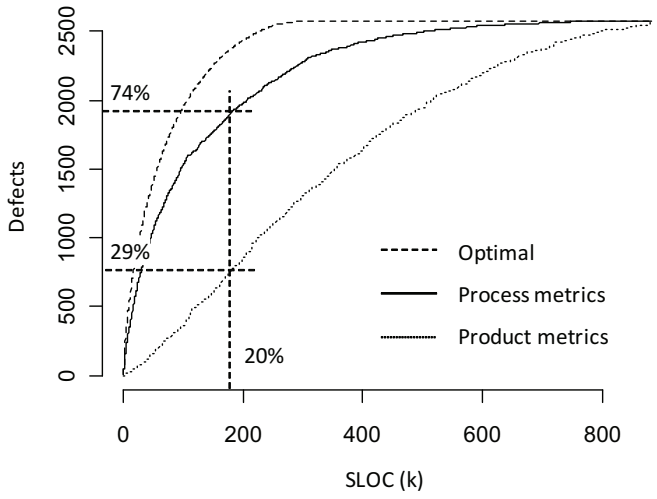

Fig. 4. File-level: LOC-based Cumulative Lift Chart for Product Metrics v.s. Process Metrics (RF, Platform v3.1$\rightarrow$v3.2).

TABLE III
FILE-LEVEL: EXPERIMENTAL RESULT FOR PRODUCT METRICS V.S. PROCESS METRICS (CROSS-VALIDATION)

(a) Platform

P_{opt}	v.3.0 (LOC : 0.49)			v.3.1 (LOC : 0.46)			v.3.2 (LOC : 0.43)		
	LM	RT	RF	LM	RT	RF	LM	RT	RF
Product	0.59	0.52	0.56	0.58	0.54	0.58	0.60	0.54	0.59
Process	0.81	0.80	0.87	0.82	0.83	0.89	0.90	0.89	0.94*
Prod. + Proc.	0.83	0.80	0.88*	0.85	0.83	0.90*	0.90	0.88	0.93

*: Best Performance LM: Linear Model RT: Regression Tree RF: Random Forest

(b) JDT

P_{opt}	v.3.0 (LOC : 0.50)			v.3.1 (LOC : 0.54)			v.3.2 (LOC : 0.47)		
	LM	RT	RF	LM	RT	RF	LM	RT	RF
Product	0.55	0.50	0.57	0.54	0.56	0.61	0.52	0.56	0.60
Process	0.80	0.80	0.86*	0.74	0.76	0.80	0.75	0.78	0.83
Prod. + Proc.	0.76	0.73	0.82	0.79	0.78	0.87*	0.74	0.76	0.85*

(c) PDE

P_{opt}	v.3.0 (LOC : 0.53)			v.3.1 (LOC : 0.60)			v.3.2 (LOC : 0.51)		
	LM	RT	RF	LM	RT	RF	LM	RT	RF
Product	0.61	0.58	0.75	0.62	0.67	0.66	0.50	0.49	0.61
Process	0.78	0.78	0.82	0.78	0.77	0.83*	0.78	0.83	0.84*
Prod. + Proc.	0.75	0.75	0.84*	0.82	0.74	0.78	0.76	0.73	0.81

TABLE IV
FILE-LEVEL: EXPERIMENTAL RESULT FOR PRODUCT METRICS V.S. PROCESS METRICS (CROSS-RELEASE)

(a) Platform

P_{opt}	v.3.0 → v.3.1 (LOC : 0.45)			v.3.1 → v.3.2 (LOC : 0.43)			v.3.1 → v.3.2 (LOC : 0.43)		
	LM	RT	RF	LM	RT	RF	LM	RT	RF
Product	0.59	0.53	0.61	0.53	0.49	0.57	0.58	0.54	0.64
Process	0.81	0.82	0.87*	0.82	0.86	0.91*	0.85	0.86	0.92*
Prod. + Proc.	0.82	0.77	0.87*	0.83	0.82	0.91*	0.87	0.84	0.92*

(b) JDT

P_{opt}	v.3.0 → v.3.1 (LOC : 0.54)			v.3.1 → v.3.2 (LOC : 0.47)			v.3.1 → v.3.2 (LOC : 0.47)		
	LM	RT	RF	LM	RT	RF	LM	RT	RF
Product	0.48	0.55	0.58	0.52	0.52	0.54	0.52	0.53	0.62
Process	0.76	0.75	0.80*	0.76	0.76	0.82*	0.80	0.77	0.84*
Prod. + Proc.	0.72	0.71	0.79	0.73	0.74	0.81	0.75	0.78	0.83

(c) PDE

P_{opt}	v.3.0 → v.3.1 (LOC : 0.59)			v.3.1 → v.3.2 (LOC : 0.50)			v.3.1 → v.3.2 (LOC : 0.50)		
	LM	RT	RF	LM	RT	RF	LM	RT	RF
Product	0.41	0.65	0.74	0.50	0.53	0.58	0.49	0.51	0.58
Process	0.84*	0.66	0.83	0.71	0.67	0.76*	0.72	0.56	0.73
Prod. + Proc.	0.82	0.65	0.83	0.70	0.68	0.73	0.74*	0.55	0.68

the package depends on are modified then the package needs to be modified too increasing the chances that a bug might be introduced. We also consider that a low-abstractness package is likely to have many bugs because many program logics could be implemented in the package. Additionally, we expect that a highly imbalanced package is likely to have many bugs because the design quality of the package might be worse.

Model Building Approach. When building a file-level prediction model, product metrics and process metrics can be used as is. However, since a package consists of multiple modules (i.e., files), we either need to lift the file-level metrics up to the package-level or use special package-level metrics (e.g., Martin's metrics). Figure 5 shows an overview of the construction process of package-level prediction model. B0 File(Prod. + Proc.) shows building a file-level prediction model. In this paper, we use the following building methods for the prediction of packages in the experiment.

B1 Lifting the file-level metrics up to the package-level
First, this building method measures metrics at the file-level and calculates representative value (e.g., maximum and median) in each package. For example, if there are three files in package with a cyclomatic complexity of 8, 4 and 2, then the representative value for the package is 8 in the case of median. Next, this building method predicts the fault density of a package. This method has been used in other studies [23]. We use the mean for this study.

B2 Metrics for package-level modules
This building method measures the metrics that can

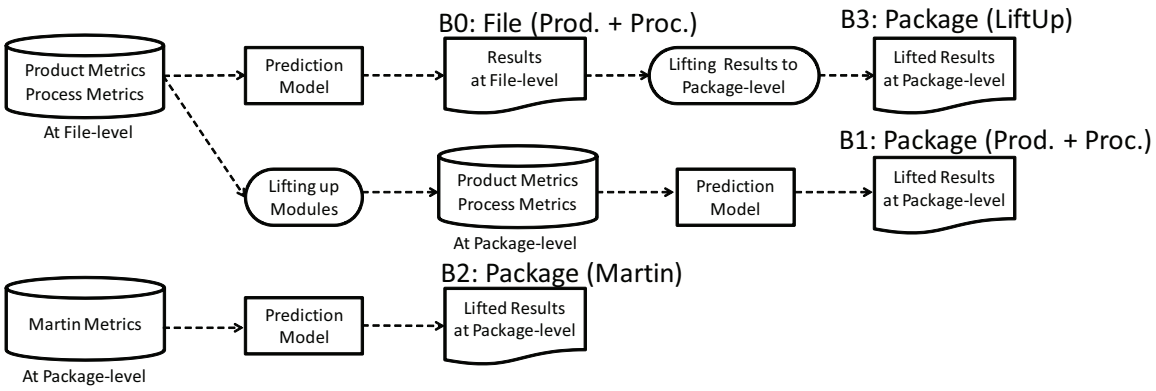

Fig. 5. Overview of the construction process of prediction models.

TABLE V
MEASURED METRICS AT THE PACKAGE-LEVEL

Name	Definition
Ca	Number of classes outside this package that depend on classes within this package
Ce	Number of classes inside this package that depend on classes outside this package. We ignore dependencies to standard libraries such as java.* and javax.*.
I	Instability $I = Ce/(Ca + Ce)$
NA	Number of abstract classes
NC	Number of concrete classes
A	Abstractness $A = NA/(NA + NC)$
D	Distance from the Main Sequence $\|A + I - 1\|$ Main Sequence: balance between A and I that is, 1

be directly collected at the package-level. We use Martin' package-level metrics and predict the fault density at the package-level. To our knowledge, no study has reported the effects of Martin metrics on fault density prediction.

B3 Lifting file-level prediction results to the package-level

Instead of lifting metrics, we lift prediction results. First, this building method measures metrics at the file-level and predicts the fault density of each file. Next, this method lifts the file-level fault density predictions to the package-level while taking into account the SLOC of a file since we lift density values not basic bug counts. To our knowledge, this paper represents the first experiment to ever consider this method of building package-level predictions.

Results of our experiment. The presentation of our results follows the same style as done in RQ1.

Cross-validation analysis

Table VI shows the experimental results using a cross validation analysis. The values in each row are the average value of P_{opt} for 20 iterations. A "*" symbol next to a value indicates that it is the best performance in a particular dataset (i.e., project). Package(LiftUp) is the result of lifting file-level combined process and product predictions (File(Prod. + Proc.)) to the package-level.

There is little difference between File(Prod. + Proc.) and Package(LiftUp). For example, in the case of RF, the P_{opt} range of File(Prod. + Proc.) is from 0.78 to 0.93, while that of Package(LiftUp) is from 0.81 to 0.93.

We find that the prediction performance of Package(LiftUp) using random forests is the best for all 3 datasets across three releases. On the other hand, the prediction performance of Package(Martin) is the worst for all three datasets across the three releases.

Cross-release analysis

Table VII shows the experimental results on a cross-release study. The values in each row are the value of P_{opt} for 1 iteration unlike the cross validation study. The result shows a somewhat similar tendency as the cross validation study. There is little difference between File(Prod. + Proc.) and Package(LiftUp).

The result show that Package(LiftUp) is better than the Package(Prod. + Proc.) and the Package(Martin). In the case of RF, the P_{opt} range of Package(LiftUp) is from 0.72 to 0.93, while those of Package(Prod. + Proc.) and Package(Martin) are from 0.51 to 0.89 and from 0.48 and 0.71.

Figures 6 and 7 show the LOC-based cumulative lift charts of the best performance of File(Prod. + Proc.) and Package(LiftUp) from Table VII. We order all files/packages by decreasing fault density. The x-axis shows the cumulative SLOC and the y-axis shows the cumulative number of bugs. The dashed line plots the cumulative lift chart for files ordered by decreasing actual (i.e., optimal) fault density, and solid line plots the cumulative lift chart for files ordered by decreasing the predicted fault density of random forest. Note that the cumulative lift charts of the optimal fault density is different between file-level and package-level because the distribution of faults inside the dataset is different at file-level versus the package-level.

Figures 6 and 7 indicate that when we conduct a test on 20% (of the lines) of all files/packages based on the predicted fault density, we could detect 74% and 62% of the faults at the file-level and package-level respectively. That is, we only need to spend 20% of the efforts that it would take to test all

TABLE VI
EXPERIMENTAL RESULT FOR FILE-LEVEL V.S. PACKAGE-LEVEL (CROSS-VALIDATION)

(a) Platform

P_{opt}	v.3.0			v.3.1			v.3.2		
	LM	RT	RF	LM	RT	RF	LM	RT	RF
File (Prod. + Proc.)	0.83	0.80	0.88	0.85	0.83	0.90	0.90	0.88	0.93*
Package (Prod. + Proc.)	0.82	0.79	0.84	0.82	0.83	0.88	0.89	0.85	0.91
Package (Martin)	0.65	0.56	0.57	0.55	0.60	0.60	0.54	0.59	0.52
Package (LiftUp)	0.86	0.87	0.90*	0.88	0.89	0.92*	0.90	0.92	0.93*

(b) JDT

P_{opt}	v.3.0			v.3.1			v.3.2		
	LM	RT	RF	LM	RT	RF	LM	RT	RF
File (Prod. + Proc.)	0.76	0.73	0.82*	0.79	0.78	0.87*	0.74	0.76	0.85
Package (Prod. + Proc.)	0.56	0.67	0.74	0.72	0.70	0.72	0.75	0.75	0.83
Package (Martin)	0.49	0.39	0.46	0.66	0.63	0.60	0.54	0.57	0.50
Package (LiftUp)	0.76	0.76	0.82*	0.79	0.83	0.87*	0.75	0.82	0.87*

(c) PDE

P_{opt}	v.3.0			v.3.1			v.3.2		
	LM	RT	RF	LM	RT	RF	LM	RT	RF
File (Prod. + Proc.)	0.75	0.75	0.84	0.82	0.74	0.78	0.76	0.73	0.81
Package (Prod. + Proc.)	0.72	0.72	0.74	0.83	0.83	0.78	0.63	0.73	0.71
Package (Martin)	0.46	0.50	0.56	0.47	0.54	0.26	0.60	0.58	0.53
Package (LiftUp)	0.76	0.78	0.88*	0.84*	0.75	0.81	0.76	0.82	0.85*

TABLE VII
EXPERIMENTAL RESULT FOR FILE-LEVEL V.S. PACKAGE-LEVEL (CROSS-RELEASE)

(a) Platform

P_{opt}	v.3.0 → v.3.1			v.3.0 → v.3.2			v.3.1 → v.3.2		
	LM	RT	RF	LM	RT	RF	LM	RT	RF
File (Prod. + Proc.)	0.82	0.77	0.87	0.83	0.82	0.91	0.87	0.84	0.92
Package (Prod. + Proc.)	0.81	0.72	0.85	0.79	0.82	0.84	0.87	0.83	0.89
Package (Martin)	0.63	0.62	0.71	0.66	0.56	0.63	0.61	0.56	0.63
Package (LiftUp)	0.86	0.87	0.90*	0.81	0.90	0.92*	0.86	0.92	0.93*

(b) JDT

P_{opt}	v.3.0 → v.3.1			v.3.0 → v.3.2			v.3.1 → v.3.2		
	LM	RT	RF	LM	RT	RF	LM	RT	RF
File (Prod. + Proc.)	0.72	0.71	0.79*	0.73	0.74	0.81*	0.75	0.78	0.83
Package (Prod. + Proc.)	0.50	0.52	0.51	0.54	0.72	0.71	0.78	0.74	0.74
Package (Martin)	0.42	0.38	0.58	0.48	0.37	0.52	0.53	0.52	0.61
Package (LiftUp)	0.65	0.69	0.72	0.72	0.79	0.80	0.81	0.84	0.85*

(c) PDE

P_{opt}	v.3.0 → v.3.1			v.3.0 → v.3.2			v.3.1 → v.3.2		
	LM	RT	RF	LM	RT	RF	LM	RT	RF
File (Prod. + Proc.)	0.82	0.65	0.83	0.70	0.68	0.73	0.74*	0.55	0.68
Package (Prod. + Proc.)	0.87	0.88	0.85	0.65	0.58	0.61	0.64	0.59	0.63
Package (Martin)	0.69	0.75	0.70	0.59	0.61	0.57	0.54	0.55	0.48
Package (LiftUp)	0.89*	0.57	0.89*	0.71	0.77	0.79*	0.74*	0.64	0.73

files to detect up to 74% and 62% of all faults using file-level and package-level model. In short, the file-level model is 20% better than the package-level model.

There is little difference between P_{opt} of the file-level and package-level when taking test effort into account. However, file-level predictions are more effective than package-level prediction, since only 20% of test effort is needed to detect up to 74% of all faults. Also, we find that lifting up prediction results is better than building package-level models using lifted metrics.

V. SUMMARY AND ANALYSIS

In this paper, we evaluated the prediction performance of effort-aware models at the file-level and package-level using three data sets collected from the Eclipse Platform. Our results indicated that, at the file-level, process metrics outperform product metrics as predictors of fault density in effort-aware models, and random forest is the best prediction model among the three models.

To illustrate the impact of product metrics and process metrics, we use *IncNodePurity* in the output of the R `randomForest` library [34]. IncNodePurity shows the mean decrease in node impurity. That is, a higher IncNodePurity means that a variable plays a more important role in a

built prediction model. Figure 8 shows IncNodePurity, sorted decreasingly from top to bottom, of the metrics as assigned by the random forest (Prod. + Proc.) for Platform v.3.1 → v.3.2. The top five metrics are all process metrics (Revisions, BugFixes, Age, LOCDeleted and Codechurn). By using random forest and process metrics, test managers and quality managers could allocate inspection/test effort to find faulty modules more effectively.

For package-level prediction, the prediction performance of Martin metrics is the worst for all three datasets. To illustrate this, we build a random forest using the 7 Martin metrics and the 22 lifted-up product metrics and process metrics in Platform v.3.1 → v.3.2 and calculated the IncNodePurity. None of the Martin metrics show up in the top five metrics. The highest three positions in the 29 metrics are positions 10(Ce), 17(D) and 21(I).

There is little difference between P_{opt} at the file and package-level. However, we find that if we test 20% of all modules based on the predicted fault density, we would detect 74% of faults using file-level models and 62% of faults using package-level models. Such performance shows that fault density prediction model at the file-level is more effective than that at package-level.

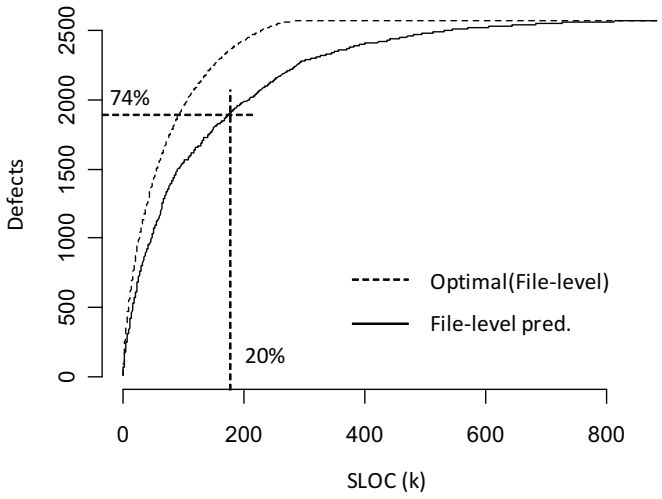

Fig. 6. LOC-based Cumulative Lift Chart at the File-level (RF, Platform v3.1→v3.2).

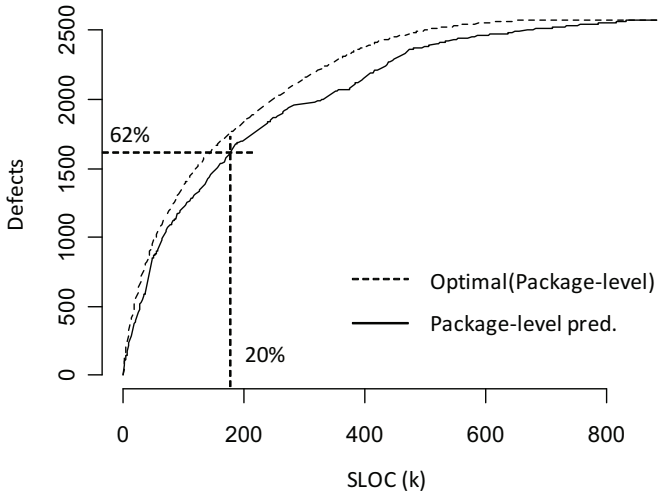

Fig. 7. LOC-based Cumulative Lift Chart at the Package-level (RF, Platform v3.1→v3.2).

VI. Threats to Validity

In this section, we discuss the threats to the validity of our work. We use a dataset collected from one foundation. In Tables III and IV, and Tables VI and VII, we can see that the tendency of evaluation value P_{opt} is similar in three subprojects and three versions. We need to analyze other open source and closed source systems to generalize our findings.

This study uses the lines of code as a measure of effort, similar to Mende and Koschke [11], because Arisholm et al. [24] pointed out that the effort is likely to be proportional to the size. Replicated studies using other measures (e.g., McCabe cyclomatic complex) of effort will be useful to assess the generalizability of our findings.

We use linear regression, regression tree, and random forest techniques to evaluate the effect of the effort-aware models, since these modeling techniques are well-known for bug prediction. However, using other modeling techniques may produce different results. We also use the mean value to build package-level models using lifted metrics. Using other representative values (e.g., maximum and median) may lead to different results.

This study obtains the number of bugs in source code files using the SZZ algorithm. The algorithm is commonly used in fault prediction research [8] and [14], but has the limitation that faults not recorded in CVS log comments cannot be collected. Further research is required to improve the accuracy of faults collection from repositories.

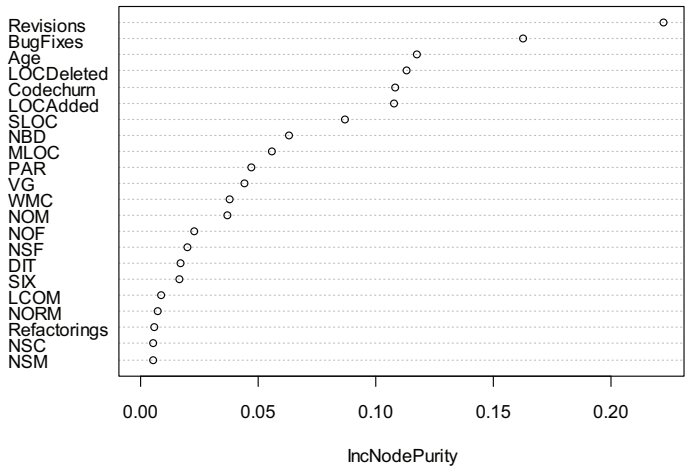

Fig. 8. IncNodePurity of metrics as assigned by the random forest at the File-level (RF, Platform v3.1→v3.2).

VII. Conclusion

In this paper, we sought to revisit some of the major findings in the prediction literature by taking into account the cost of additional software quality assurance efforts. We experimentally evaluated the performance of effort-aware models using data from three subprojects (Platform, JDT and PDE) from the Eclipse Foundation across three releases. Our major findings include the following:

- At the file-level, process metrics still outperform product metrics when considering quality assurance efforts. We show a 2.6 times improvement between process metric and product metrics;
- At the file-level, random forest produce the best predictor performance compared to linear models and regression trees;
- Package-level predictions are less effective than file-level predictions. When we test or review 20% (of the lines) of all modules based on the predicted fault density, we could detect almost 74% of faults using file-level models versus 62% of faults using package-level models;
- At the package-level, lifting up prediction results is better than building package-level models using lifted metrics.

The major limitation of this paper is that we used only a dataset collected from one foundation. Our future work is to confirm our results using other project datasets. We also plan to explore other more appropriate measures as a proxy for the effort instead of lines of code.

Acknowledgements

We are grateful to the reviewers for their valuable comments. This research is being conducted as a part of the Next Generation IT Program by the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

[1] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886–894, 1996.

[2] M. Pighin and R. Zamolo, "A predictive metric based on discriminant statistical analysis," in *Proc. Int'l Conference on Software Engineering (ICSE'97)*, 1997, pp. 262–270.

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

[4] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proc. Int'l Conference on Predictor Models in Software Engineering (PROMISE'09)*, 2009, pp. 1–10.

[5] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[6] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. Int'l Conference on Software Engineering (ICSE'09)*, 2009, pp. 16–24.

[7] T. M. Khoshgoftaar and E. B. Allen, "Modeling software quality with classification trees," in *Recent Advances in Reliability and Quality Engineering*. Singapore: World Scientific, 1999, pp. 247–270.

[8] O. Mizuno and T. Kikuno, "Prediction of fault-prone software modules using a generic text discriminator," *IEICE Transactions on Information and Systems*, vol. E91-D, no. 4, pp. 888–896, 2008.

[9] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, "Software defect association mining and defect correction effort prediction," *IEEE Transactions on Software Engineering*, vol. 32, no. 2, pp. 69–82, 2006.

[10] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proc. Int'l Conference on Software Engineering (ICSE'08)*, 2008, pp. 531–540.

[11] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Proc. of European Conference on Software Maintenance and Reengineering (CSMR'10)*, 2010, pp. 109–118.

[12] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[13] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. Int'l Conference on Software Engineering (ICSE'06)*, 2005, pp. 284–292.

[14] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. Int'l Conference on Software Engineering (ICSE'08)*, 2008, pp. 181–190.

[15] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *Proc. of the 2006 ACM/IEEE international symposium on Empirical software engineering (ISESE'06)*, 2006, pp. 18–27.

[16] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proc. Int'l Workshop on Predictor Models in Software Engineering (PROMISE'07)*, 2007, p. 9.

[17] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002.

[18] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

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

[20] Y. Kamei, A. Monden, S. Morisaki, and K. ichi Matsumoto, "A hybrid faulty module prediction using association rule mining and logistic regression analysis," in *Proc. Int'l Symposium on Empirical Software Engineering and Measurement (ESEM2008)*, 2008, pp. 279–281.

[21] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.

[22] Y. Jiang, B. Cukic, and Y. Ma, "Techniques for evaluating fault prediction models," *Empirical Software Engineering*, vol. 13, no. 5, pp. 561–595, 2008.

[23] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. Int'l Conference on Software Engineering (ICSE'06)*, 2006, pp. 452–461.

[24] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *The Journal of Systems and Software*, vol. 83, no. 1, pp. 2–17, 2010.

[25] A. L. Edwards, *Introduction to Linear Regression and Correlation*. W.H.Freeman & Co Ltd, 1976.

[26] L. Breiman, J. Friedman, C. J. Stone, and R. Olshen, *Classification and regression trees*. Chapman and Hall/CRC, 1984.

[27] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[28] R, "The R Project for Statistical Computing," http://www.r-project.org/, last viewed: 17-Apr-2010.

[29] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. Int'l Conference on Mining Software Repositories (MSR'05)*, 2005, pp. 1–5.

[30] MSR Mining Challenge 2008, http://msr.uwaterloo.ca/msr2008/challenge/index.html, last viewed: 17-Apr-2010.

[31] Frank Sauer, "Eclipse Metrics plugin," http://sourceforge.net/projects/metrics, last viewed: 17-Apr-2010.

[32] S. Sato, A. Monden, and K. Matsumoto, "Evaluating the applicability of reliability prediction models between different software," in *Proc. Int'l Working on Principles of Software Evolution (IWPSE'02)*, 2002, pp. 97–102.

[33] C. Kirsopp and M. Shepperd, "Making inferences with small numbers of training sets," *IEE Proceedings Software*, vol. 149, no. 5, pp. 123–130, 2002.

[34] A. S. Foulkes, *Applied Statistical Genetics with R*. Springer New York, 2009.