The Learning Curves in Open-Source Software (OSS) Development Network

Youngsoo Kim
School of Information Systems
Singapore Management University
80 Stamford Road
Singapore, 178902
yskim@smu.edu.sg

Lingxiao Jiang
School of Information Systems
Singapore Management University
80 Stamford Road
Singapore, 178902
lxjiang@smu.edu.sg

ABSTRACT

We examine the learning curves of individual software developers in Open-Source Software (OSS) Development. We collected the dataset of multi-year code change histories from the repositories for five open source software projects involving more than 100 developers. We build and estimate regression models to assess individual developers' learning progress (in reducing the likelihood they may make a bug). Our estimation results show that developer's coding experience does not decrease bug ratios while cumulative bug-fixing experience leads to learning progress. The results may have implications and provoke future research on project management about allocating resources on tasks that add new code versus tasks that debug and fix existing code. We also find that different developers indeed make different kinds of bug patterns, supporting personalized bug prediction in OSS network. We found the moderating effects of bug types on learning progress. Developers exhibit learning effects for some simple bug types (e.g., wrong literals) or bug types with many instances (e.g., wrong `if` conditionals).

1. INTRODUCTION

As the old saying goes, "practice makes perfect." Learning from actual coding can be effective for developers to gain new knowledge and horn their skills. Such learning can be a life-long journey for both novice and expert developers, with continually appearing new technologies and new problem domains. No matter whether a developer is a novice or an expert, software bugs can inevitably occur in their code.

The generally encouraging consensus is that developers would be able to reduce the number of bugs they make when they accumulate more knowledge and skills. This implies that the "patterns" of code (including bugs and bug fixes) made by a developer may change over time. On the other hand, many bug prediction techniques have been proposed to identify likely locations in a code base that may contain bugs [13, 15, 18, 21, 23, 29, 40, 51]. Some of the prediction techniques are further "personalized" to take individual developers' code patterns into consideration to improve prediction accuracy [19, 31]. Most of the techniques utilize information about historical bugs happened in the code base, such as code metrics, program dependencies among components, and social factors in a software project [6, 7, 49], to make predictions. Such studies then implicitly assume that the "patterns" of bugs in a code base rarely change and past bugs can be used to predict future ones, which may appear contradicting with the above consensus.

ICEC '14, August 05 - 06 2014, Philadelphia, PA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-2618-6/14/08...$15.00.
http://dx.doi.org/10.1145/2617848.2617857

In this paper, we examine such an implicit assumption made by many bug prediction techniques, with the aim to find a resolution for this contradiction. We attempt to answer following research questions:

RQ1 Do developers exhibit learning effects in a project by accumulating coding and/or bug fixing experience?
RQ2 Do developers show different learning curves depending on bug types?
RQ3 Do developers show different distribution of bug types?
RQ4 What characteristics of a developer and a project may affect the learning curved of the developer?

Our study have intriguing implications on project management about how to split efforts on tasks that add new code versus tasks that debug and fix existing code. For example, within a short period of time (say, a month), when a lot of new code is added by a developer, a manager may want to assign more testing and debugging tasks to the developer instead of coding more new code; it helps, on one hand, to detect and fix possible bugs, and on the other hand, to improve developers' learning progress via bug fixing. It may also be useful for a developer if she continually studies the fixes for her bugs even though she is not the person who fixes them. Our main contributions in this paper are as follows:

- We are the first to identify the implicit assumption made by most bug prediction techniques that "patterns" of bugs made by developers in different projects are relatively stable, and present an empirical study to investigate the validity of the assumption;
- We collect and calculate various measures for more than 95K lines of buggy code made by more than 100 developers in five open source projects;
- We analyze developers' learning effects in the data set via empirical regression models, and reveal interesting phenomena that may provide insights for project management and future research on bug fixing.

The rest of this paper is organized as follows. Section 2 briefly surveys closely related work. Section 3 describes the dataset collected and measures. We will show our empirical models in Section 4. Section 5 presents our analysis results. Section 6 discusses some implications of our results, together with limitations and threats to validity, and possible directions for future work. Section 7 concludes.

2. RELATED WORK

We discuss related work in the area of learning models in software engineering, empirical studies on factors affecting software project performance, and bug prediction.

2.1 Learning Models in Software Engineering

There are many studies on learning models in software engineering. Hanakawa et al. [16] incorporates developers' learning curve into a simulation model to make better project plans. Singh et al. [41] develops a learning model for developers and finds that patterns of learning are different in different learning states. Chouseinoglou et al. [11] uses a model to assess the learning characteristics of a software developer organizations (SDO). Abu et al. [2] proposes an improved learning model for software test processes. Singh et al. [41] examine the developer's learning dynamics in OSS utilizing hidden Markov Model (HMM). In their paper, the performance measure is just the contribution to software project. None of previous studies focus on the relationship between bug-fixing code amount and learning effects. We are the first to investigate the effect of bug-fixing on developers' bug ratios.

2.2 Empirical Studies on Project Performance

The performance of a software project can be measured in various ways, such as developer productivity, code quality, and maintenance costs. Many studies have analyzed various factors that may affect project performance. Ramasubbu and Balan [37] use regression models to identify that geo-dispersion of developers has great impact on software productivity and quality. Banker et al. [4] finds that the improvement of software development practices can improve the maintenance performance. Harter et al. [17] finds that higher process maturity can lead to higher software quality. Krishnan et al. [24] investigates the relationship between various measures (e.g., product size, personnel capability, software process) and the software quality. Abreu and Premraj [1] propose that communication frequency of developers may affect the amount of bug-introducing changes. Bettenburg and Hassan [6] focus on the impact of social information of developers on software quality. Our study in this paper reveals the relationship between bug-fixing amount and learning effects and software quality indirectly.

2.3 Bug Prediction

There are many studies on bug prediction [13,15]. Some focus on cross-project bugs [27,30,35,46,51], building prediction models with data from different projects. Some build prediction models for individual developers [19,31]. Many of them rely on machine learning techniques to classify changes or files as possible bugs [19,21,40]. To find bug-introducing changes (bug-origin analysis), many studies [22,42,43,50] try to identify the bug-fixing changes first, which is similar to our work. These studies make it more effective to predict software bugs, but they do not take developers' learning effects into consideration. They assume that "bug patterns" of developers in different projects and different years are relatively stable. We are the first to identify this implicit assumption and investigate it with empirical analysis. Some bug prediction techniques use information beyond code. Some use developer social networking [1,7,33,49]. Some [25] also use developers' micro-interaction patterns captured by the Mylyn plugin for Eclipse to predict bugs. Our analysis focuses on factors in code; we plan to extend our analysis to consider factors beyond code as well.

3. DATA AND MEASURE

3.1 Data

We collect code change histories from the repositories for five open source software projects mostly written in Java (Apache Ant, Apache Commons Compress, Apache Commons Lang, Apache Solr/Lucene, and Eclipse Platform). The data spans multiple years from 2000 to 2013, involving more than 100 developers. We extract and calculate various measures about the characteristics of the project and the code changes (including bugs and bug fixes), such as the location of the bug, the developer who introduces the bug or fixes the bug, the introduction time of the bug, the type of the bug, the complexity of the code involved, etc. The bugs we analyze span more than 95K lines of code across different versions of the projects.

3.2 Measures

We calculate basic measures for individual bugs, then use them to calculate the measures for developers and projects. Here we describe the measures for individual code changes, and how we collect and calculate them. We focus on open source projects mostly written in Java. We also use the git repositories (`http://git-scm.com/`) of the projects [8].

In order to find a bug and its introducing commits, we first locate a bug fix, then trace back to its origin, in a way similar to previous studies [22,36,43]. To get a bug fix, we search among all commit log messages for ones containing the keyword "fix" or"bug." Then we verify the search results manually to ensure the selected commits are really source code bug fixes. For example, if the log for a commit is "Fix JavaDoc", this commit is not a source code bug fix. After verifying the bug fixes, we get the `diff` between each fix commit and its parent commit (i.e., the last commit before the fix commit) so that we can identify the changed lines in the parent commit. These lines of code are treated as buggy; newly added lines in the fix commit are ignored. And, we count each buggy line as one bug. Then, we use the command "git blame" to get information about each buggy line (such as the last developer who changed the buggy line and the date when the line was last changed). The measures for individual bugs are described in more details below.

$Location_j$**:** This measure is for identification purpose. It provides the project name, the commit id in the project respository, the package name, the filename, and the line number for the buggy line of code j identified as above.

$IntroDate_j$**:** It is the date when the buggy line of code j was committed into the project repository for the first time. Although there are threats to validity of $IntroDate$ obtained in this way [8], it seems to be sufficient approximation as in other studies that use information about bug origins [19,36,43].

$IntroDeveloper_j$**:** It is the developer who introduced the buggy line j into the repository. Similar to $IntroDate$, we mostly rely on "git blame" of the `diff` to get the developer who last changed j as the $IntroDeveloper_j$.

$FixDate_j$**:** It is the date when the bug j was fixed in the repository, i.e., the date of the fix commit.

$FixDeveloper_j$**:** It is the developer who fixed the bug i, i.e., the developer who committed the fix into the repository.

$BugType_j$**:** The type of the buggy line j is based on our classification. We classify the type of each bug based on the syntax of the bug, following the study on syntax-based classification of bug fixes [32]. In this paper, to decide the bug type, we first construct the abstract syntax tree (AST) for the source file containing the bug, then identify a minimum subtree that contains all code in the buggy line. Then, we count the number of occurrences of each tree node type in the subtree, and give some node types (e.g., `if` and `for` nodes) higher priorities based on common patterns shown in [32]. Then, we choose the node type with the highest weighted occurrence number as the type for the bug. In the ASTs constructed by Eclipse JDT (`http://www.eclipse.org/jdt/`), there are more than 80 node types. With a preliminary study, many of the node types have relatively small numbers of bugs. Thus, we merge some "semantically" related node types. In the end, we have 13 bug types, which also helps to simplify some of our empirical analysis as described in Section 4. Table 1 lists the 13 merged bug types and their descriptions.

#	Bug Type	Descriptions
1	Types	Code for defining and using Java types (e.g., type casting, "`instanceof`", enum, type parameters, etc.)
2	Def-use	Code for defining and using variables (e.g., variable declarations, assignments, array accesses, field accesses, "`this`", etc.)
3	Error handling	Code for assertion, exception handling
4	Scoping	Code for identifying scopes (e.g., "{", "}", etc.)
5	Literals	Constants (e.g., "`hello`", "`123`", "`null`", etc.)
6	Change control	Code that changes the control flow (e.g., "`break`", "`continue`", "`return`", etc.)
7	Branching	Code involving conditionals (e.g., "`if`", "`switch`", etc.)
8	Looping	Code involving loops (e.g., "`for`", "`while`", etc.)
9	Non-essentials	Code that has little effect on functionality or easily caught by compilers (e.g., empty statement, annotations, comments, imports, labels, etc.)
10	Expressions	Code involving expressions (e.g., infix expression, parenthesized expressions, etc.)
11	Methods	Code involving method declarations and invocations
12	Synch	Code involving synchronization
13	Modifiers	Code involving modifiers (e.g., "`public`", "`private`", "`static`", etc.)

Table 1: Syntax-based Bug Types, classified from 80+ AST node types from Eclipse JDT.

$ContextFunctionality_j$**:** The "functionality" of the surrounding code of the bug j. We classify the functionality of the file containing the bug by applying topic modelling on the comments contained in the file. We extract comments as bags of words, apply stop-word removal (e.g., keywords in Java) and stemming, and use JGibbLDA (`http://jgibblda.sourceforge.net/`) to identify topics. We hypothesize that different packages/components in a program may have different functionality, and thus pick an arbitrary number 35 as the number of topics for JGibbLDA within the range of numbers of packages in our subject programs (between 13 and 302 as counted by JavaNCSS (`http://www.kclee.de/clemens/java/javancss/`) for their last versions in our data set). We use the topic assigned by JGibbLDA to a file with the highest probability as the topic for the file.

We can calculate many measures for projects. Many of them can be aggregated from the measures for individual developers and individual code changes and bugs within given time frames (months). When needed, we use the last commit before the current time period t as the beginning of t, and use the first commit as the beginning of the first period.

$ProjectSize_{pt}$**:** The amount of code (i.e., the total number lines of code) in the project p at the beginning of the time period t. A project size can be viewed as a proxy for the accumulative effects of many code changes by many developers in the project. To calculate these numbers, we first "git checkout" the last commit before the time t to get the specific revision of p, then use a code metric tool JavaNCSS to count the code amount.

$ProjectComplexity_{pt}$**:** The code complexity of the project p at the beginning of the time period t. We also use JavaNCSS to calculate the cyclomatic complexity (CC) of Java code, and we use the sum of the CC of all functions in a project as the complexity for the project. One can see that $ProjectComplexity$ is highly correlated with $ProjectSize$.

$DeveloperSize_{pt}$**:** The number of developers who made some commits into p during the time period t.

$ProjectCodeAmount_{pt}$**:** The amount of code committed into p during the time period of t. We can obtain these numbers by summing up the $CodeAmount$ of all developers in the project during the period of t.

$ProjectBugAmount_{pt}$**:** The amount of bugs (i.e., the total number of lines of buggy code) committed into the project repository p during the time period of t. These numbers can be the summation of the $BugAmount$ of all developers in the project during the period of t.

$ProjectBugTypeAmount_{bpt}$**:** The amount of bugs of type b committed into p during the time period of t. These numbers can be the summation of the $BugTypeAmount$ of all developers in the project during the period of t.

$ProjectBugRatio_{pt}$**:** The ratio of buggy code for the project p during the time period of t. It is $ProjectBugAmount_{pt}$ divided by $ProjectCodeAmount_{pt}$.

Since each commit is often associated with a unique developer name or email, we can calculate most measures for individual developers by aggregating the measures for individual code changes or bugs within given time frames (months).

$CodeAmount_{ipt}$**:** The amount of code (i.e., the total number of lines of code, including deleted, added, changed lines) committed by the developer i into p during the time period of t. These numbers can be summed up from the `diffs` of all commits made by the developer. We omit `diffs` in non-Java files.

$BugAmount_{ipt}$**:** The amount of bugs (i.e., the total number of lines of buggy code) committed by the developer i into p during the time period of t. These numbers are the sum of all bugs whose $IntroDate$ is in the period of t and $IntroDeveloper$ is i.

$BugTypeAmount_{ibpt}$**:** The amount of bugs of type b committed by the developer i into p during the time period of t. These numbers are calculated in a similar way as $BugAmount$, except that $BugTypeAmount$ additionally considers bug types.

$BugFixAmount_{ipt}$**:** The amount of code committed by the developer i into p, for the purpose of fixing bugs, during the time period of t. These numbers are the sum of all bugs whose $FixDate$ is in the period of t and $FixDeveloper$ is i.

$BugTypeFixAmount_{ibpt}$**:** The amount of code committed by the developer i into p, for the purpose of fixing bugs of type b, during the time period of t. These numbers are similar to $BugFixAmount$, except that $BugTypeFixAmount$ considers bug types too.

$BugRatio_{ipt}$**:** The ratio of buggy code for the developer i in p during the time period of t. It is $BugAmount_{ipt}$ divided by $CodeAmount_{ipt}$.

4. ECONOMETRIC MODEL

4.1 Learning Curve Models

The form of the learning curve is formulated as $y(x) = ax^b$, where y is a performance variable (one of the bug ratios in this paper), x represents *cumulative learning experience*, a is an initial bug ratio without learning activities, and b is the learning rate of an individual. Learning experience in this paper is measured by either the amount of codes of an individual makes or the amount of buggy code an individual fixes. *Cumulative* learning experience at a point of time t is the total amount of codes or bug fixes made by an individual through $t-1$ in a project. Taking a natural log transformation of both sides and adding covariates of interest and control variables, we obtain the following regression equation (1):

$$\begin{aligned} ln(BugRatio_{ipt}) &= \beta_0 + \beta_1 ln(CumulativeCodeAmount_{ipt-1}) \\ &+ \beta_2 ln(CumulativeBugFixAmount_{ipt-1}) \\ &+ \beta_3 CodeAmount_{ipt} \\ &+ \beta_4 ln(BugFixAmount_{ipt}) \\ &+ \beta_5 DeveloperSize_{pt} \\ &+ \beta_6 ProjectCodeAmount_{pt} \\ &+ \beta_7 ProjectComplexity_{pt} \\ &\quad (\text{ or } + \beta_7 ProjectSize_{pt}) \\ &+ \zeta_i + \delta_p + \mu_{ipt} \end{aligned} \tag{1}$$

The unit of analysis is the bug ratio of individual developers. $BugRatio_{ipt}$ is the bug ratio of i^{th} developer during the period of time (month) t in a project p; it is our key measure for learning effects and the dependent variable in our regression model. We aim to explain the change in $BugRatio_{ipt}$ with respect to the independent (explanatory) variables at the right hand side of Equation (1).

We have two variables as proxies to measure the transition (increment) of project-specific knowledge stock:

$CumulativeCodeAmount_{ipt-1}$**:** It is the cumulative total of $CodeAmount_{ipt}$ that an individual i has committed into p *throughh* time $t-1$. The main objective of the variable is to estimate learning progress induced from cumulative coding experience. If β_1 is negative and statistically significant, then the developers show the learning curve (i.e., decrease of bug ratios) as they increase coding experience in a focal project.

$CumulativeBugFixAmount_{ipt-1}$**:** In a similar fashion, this measures the cumulative total of bug-fix amounts an individual i has made *through* time $t-1$ in a project p. We use it to test whether the individual developer's bug-fixing experience can induce the decrease of bug ratios. Note that a developer can fix her own bugs as well as bugs made by other developers.

Our regression model also includes $CodeAmout_{ipt}$—the amount of code a developer i commits into p during the period of time t, and $BugFixAmount_{ipt}$—the amount of code for bug fixes a developer i commits into p during the period of time t. They are used to capture the scale effects. We also have four project-specific variables in the model to check the impact of project-related characteristics on the developers' learning effects: $DeveloperSize_{pt}$, $ProjectCodeAmount_{pt}$, $ProjectComplexity_{pt}$, and $ProjectSize_{pt}$. We find that $ProjectComplexity$ is highly correlated with $ProjectSize$ (correlation coefficient is 0.998); thus, we do not use them together in the model to avoid a multicollinearity problem [14, 44].

We adopt a fixed effects model ζ_i to control for individual developer heterogeneity, and δ_p to control for the individual project heterogeneity, respectively. The error component, μ_{ipt} is an idiosyncratic error term and it varies across t as well as across developer i and project p.

4.2 The Moderating Effect of Bug Types on Learning Progress

Our model assesses the overall learning curve of developers in the deduction of bug ratios with respect to coding or bug-fixing experience without distinguishing *bug types*, assuming implicitly that developers' learning progress is independent of bug types. Relaxing the assumption, our next question is whether learning curves differ according to bug types.

We cannot apply the same regression model as Equation (1) to estimate the learning progress in each bug type because some developers did make bugs in just a few bug types in each time period. As described in Section 3.2, we have 13 different bug types. We empirically confirmed that a month (our unit of time) is too short to calibrate the change of bug ratios in each bug type. Therefore, we split individual developers' working periods into two periods, each period with the same length of months, and calculate the bug ratios in each bug type by individual developers in each of the two time periods. Hence, we compare 13 multivariate means of two period groups.

We first perform a multivariate analysis of variance (MANOVA [14, 44]) to test whether the bug ratios in every bug type are different across the two time periods. Furthermore, we perform analysis of variance (ANOVA [14, 44]) in order to evaluate the individual learning progress in each bug type. We check whether there is significant decrease of bug ratios between the two periods.

Bug ratios can be affected by a life cycle of software development. To overcome the potential spurious influence, we apply differences–in–differences technique (DID [3]) in measuring bug ratios in each bug type. That is, bug ratios in each bug type represent the difference between individual developer's bug ratios and average bug ratios at the project level, in each bug type, for the same time frames.

4.3 Different Bug Patterns Across Developers

Bug ratios of an individual developer may reflect developer's knowledge stock and heterogeneous experiences. So another analysis we perform is to test whether the bug ratios in every bug type vary across developers. Because bug ratios in bug types can be clarified as multiple dependent variables, we perform a multivariate analysis of variance (MANOVA) for this purpose.

5. EMPIRICAL RESULTS

Table 2 gives some statistics about the projects. In total, the projects involve more than 200 developers who make commits to the repositories. 117 of them, based on our measures, have committed *buggy* code. Table 2 also shows that

Project Name	Time Span	Project Size	Dev. Size	Cumulative Bug Amt.	Self-Fixed Bug Amt.	% of Self-Fixed
Apache Ant	2000.1–2013.12	93658	30 (46)	34817	8967	25.8%
Apache Commons Compress	2003.11–2013.12	19967	9 (20)	3271	1499	45.8%
Apache Commons Lang	2002.7–2013.12	44455	15 (41)	1549	759	49.0%
Apache Solr / Lucene	2010.3–2013.12	421295	36 (40)	33812	8987	26.6%
Eclipse Platform	2001.5–2013.12	60252	27 (63)	22111	7576	34.3%

Table 2: Basic Project Descriptions. "Time Span" is the period of time between the first commit and the last commit into the project in our data set. "Project Size" is the number of lines of code in the last commit. "Dev. Size" is the number of developers who commit some *buggy* code into the project repository during the time span; and the number in the parentheses is the total number of developers who make a commit (no matter whether the commit contains code or buggy code). "Cumulative Bug Amt." is the *cumulative* total amount of buggy code committed into the project during the time span. "Self-Fixed Bug Amt." is the amount of buggy code fixed by the same developer who commits it, and "% of Self-Fixed" is the corresponding percentage.

Project	Cumulative $CodeAmount_{jpt}$ for each developer (Lines of Code)	Mean	Min	Max	Standard Deviation
Ant	with NO bugs	4053	0	54713	13571
	with bugs	73821	320	1014953	190611
Commons Compress	with NO bugs	1385	0	10787	3385
	with bugs	14691	231	65139	23265
Commons Lang	with NO bugs	7077	0	171391	33533
	with bugs	51101	332	473406	122483
Solr / Lucene	with NO bugs	1702	0	5281	2484
	with bugs	140698	263	1673065	318849
Eclipse Platform	with NO bugs	5910	0	157791	26771
	with bugs	53179	54	288889	75520

Table 3: Summary statistics (mean, min, max, standard deviation) of the cumulative code amounts of every developer in projects.

only a small portion of buggy code (25%–50%) is fixed by the same developer who commits it into the project repository.

Table 3 gives summary statistics about the *cumulative* amount of code made by developers with and without bugs in each project. It indicates that the developers without bugs contribute much less code than developers with bugs, and they do not exhibit learning effects (i.e., their bug ratios are always zero), we leave those developers out of our analyses.

Table 4 lists some descriptive statistics for the variables used in our regression model. The correlation matrix for the variables are in Table 5. The baseline correlations provide initial support for our learning curves of individual developers in OSS. $ln(BugRatio_{ipt}$ has a negative correlation with both experience variables, $ln(CumulativeCodeAmount_{ipt})$ and $ln(CumulativeBugFixAmount_{ipt})$. This indicates that an increase in the experiences is associated with the reduction in bug ratios. But the correlation cannot fully guarantee the learning effects due to developer's heterogeneity and so we run the regression model with control variables and fixed effects factors. In fact, our analysis shows that there are no learning effects induced from cumulative coding experience (see Section 5.1).

5.1 Learning Curve

Our estimation results are generally stable across different model specifications as shown in the columns in Table 6, so we focus on the results in the column for Model 3 which has all of the control variables except $ProjectSize_{pt}$. As can be seen in the rows in Table 6 for $ln(CumulativeCodeAmount_{ipt-1})$ and $ln(CumulativeBugFixAmount_{ipt-1})$, the estimates indicate that cumulative coding experience does not decrease

VID	Variables	#	Mean	Std. Dev.	Min	Max
V0	Date				200001	201312
V1	$ln(BugRatio_{ipt})$	1084	-4.2	1.6	-11.6	0.0
V2	$ln(CumulativeCodeAmounts_{ipt-1})$	2200	10.0	2.2	0.0	14.3
V3	$ln(CumulativeBugFixAmounts_{ipt-1})$	1332	5.6	2.0	1.8	9.7
V4	$CodeAmounts_{ipt}$	2330	4120.0	24745.1	1.0	634736.0
V5	$BugFixAmounts_{ipt}$	2331	40.4	435.7	0.0	14448.0
V6	$ProjectCodeAmount_{pt}$	2331	46616.4	103869.9	2.0	664740.0
V7	$DeveloperSize_{pt}$	2331	9.7	6.6	1.0	25.0
V8	$ProjectComplexity_{pt}$	2331	37364.8	32668.0	0.0	113839.0
V9	$ProjectSize_{pt}$	2331	126099.2	120569.3	0.0	413933.0

Table 4: Summary statistics (mean, min, max, and standard deviation) for the variables used in our regression model.

VID	V1	V2	V3	V4	V5	V6	V7	V8	V9
V1	1.000								
V2	-0.246	1.000							
V3	-0.158	0.556	1.000						
V4	-0.291	0.141	-0.005	1.000					
V5	-0.029	0.062	0.040	0.049	1.000				
V6	-0.050	0.038	-0.196	0.291	0.027	1.000			
V7	-0.188	0.200	-0.125	0.076	0.029	0.433	1.000		
V8	-0.254	0.282	0.046	0.024	0.020	0.225	0.867	1.000	
V9	-0.251	0.281	0.025	0.029	0.021	0.241	0.878	0.998	1.000

Table 5: Correlations between variables.

Independent Variables	Dependent Variable: $ln(BugRatio_{ipt})$			
	Model 1	Model 2	Model 3	Model 4
$ln(CumulativeCodeAmount_{ipt-1})$	0.0076 (0.0386)		-0.0099 (0.0947)	-0.0132 (0.0943)
$ln(CumulativeBugFixAmount_{ipt-1})$		-0.2442*** (0.0458)	-0.2204*** (0.0665)	-0.2263*** (0.0660)
$CodeAmount_{ipt}$	-0.0015*** (0.0000)	-0.0014*** (0.0000)	-0.0014*** (0.0000)	-0.0014*** (0.0000)
$BugFixAmount_{ipt}$	-0.0001 (0.0001)	0.0000 (0.0001)	0.0000 (0.0001)	0.0000 (0.0001)
$ProjectCodeAmount_{pt}$			0.0000 (0.0000)	0.0000 (0.0000)
$DeveloperSize_{pt}$			0.0188 (0.0251)	0.0000 (0.0000)
$ProjectComplexity_{pt}$			0.0000 (0.0000)	
$ProjectSize_{pt}$				0.0179 (0.0251)
Constant	-4.1688*** (0.3987)	-2.5939*** (0.2692)	-3.0593*** (0.9873)	-3.0387*** (0.9869)
N	1015	647	643	643
Within R^2	0.0870	0.1203	0.1256	0.1253
Adjusted R^2	0.2587	0.2428	0.2392	0.2390
Prob. > F (Prob. > $\chi 2$)	0.0000	0.0000	0.0000	0.0000
• Results significant at $p < 0.01$ are indicated by ***; results significant at $p < 0.05$ are indicated by **; results significant at $p < 0.1$ are indicated by *.				

Table 6: Learning Curve Estimates.

bug ratios while cumulative bug-fixing experience leads to learning progress.

The coefficient of the cumulative coding experience is negative (-0.0099) but "insignificant", indicating that bug ratios would not decrease even though the cumulative code amount made by an individual increases. That is, there is no learning relationship between coding experience alone and the likelihood for a developer to make a bug. Whereas, $ln(CumulativeBugFixAmount_{ipt}$ shows a significant negative coefficient, supporting the learning curve that developers are less likely to make bugs as their bug-fixing experience increases. These findings give us the intriguing insight that developers' performance (bug ratios) may not improve through just coding experience, while their performance significantly improves by fixing bugs made by either themselves or other developers.

Our model with control variables is estimated to explore and control alternative explanation for the results. The sig-

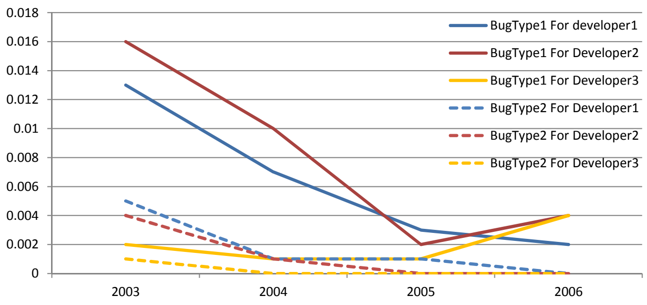

Figure 1: Bug ratios for two bug types for different developers in one project over years 2003–2006.

nificant and negative coefficient of $CodeAmount_{ipt}$ shows negative scale effects that a developer is likely to make relatively less bugs (lower bug ratios) with more coding in a period of time.

All the project-specific time variant variables ($DeveloperSize_{pt}$, $ProjectCodeAmount_{pt}$, $ProjectComplexity_{pt}$, and $ProjectSize_{pt}$) are not significant. We can partly conclude that project-specific factors do not significantly affect an individual developer's learning progress. However, we should be cautious of this conclusion because our project fixed effects models may capture the variation of bug ratios across projects.

5.2 Learning Curve by Bug Types

We explore whether different developers in one project may reduce their bug ratios for a particular bug type over periods of time. The bug ratios of three developers for two bug types in one project over years 2003–2006 are shown in Figure 1. It indicates that in the same project, the likelihood for a developer to make a bug of some particular types may change across different years.

The result from MANOVA is that the bug ratios of individual developers across bug types are statistically different between two time periods, supporting that there is learning progress in some bug types. For sensitivity analysis, we repeated the test with different ways to split time periods ("40% versus 60%" and "60% versus 40%", in addition to "50% versus 50%", see Section 4.2) and got the same results.

Furthermore, we identify developers' learning effects for each of the bug types with analysis of variance (ANOVA) (Table 7). The columns under "Developers" summarize the results of ANOVA, showing the difference of developers' bug ratio distribution for every bug types. Our analysis supports the heterogeneous distributions of bug types across developers and bug types. More than 100 developers show similar bug ratios in only 4 out of 13 bug types. The columns under "Learning" indicate that developers show learning effects in 6 out of 13 bug types. As a developer accumulates more experience, the bug ratio in the 6 bug types significantly decreases. We notice that developers exhibit clearer learning effects in bug types (1) that are relatively simple, such as Type 5 involving wrong literals and Type 9 involving bugs that are non-essential for code functionality (e.g., importing needed libraries, adding annotations, etc.), and (2) that have relatively large numbers of instances, such as Type 7 involving errors in conditionals and Type 11 involving method declarations and invocations. We plan in near future to explore reasons for the phenomena observed in this analysis and investigate possible effects of different ways for classifying bug types (see the discussion in Section 6).

		Developers		Learning	
#	Bug Type	F-statistics (Prob > F)	Results	F-statistics (Prob > F)	Results
1	Types	2.50 (0.0000)	Different	0.00 (0.9646)	No Learning
2	Def-use	1.90 (0.0000)	Different	1.67 (0.1970)	No Learning
3	Error handling	0.75 (0.9717)	Same	1.51 (0.2198)	No Learning
4	Scoping	4.09 (0.0000)	Different	4.43 (0.0355)	Learning Effects
5	Literals	2.47 (0.0000)	Different	4.35 (0.0373)	Learning Effects
6	Change control	1.53 (0.0008)	Different	4.62 (0.0319)	Learning Effects
7	Branching	2.49 (0.0000)	Different	6.12 (0.0135)	Learning Effects
8	Looping	0.80 (0.9297)	Same	1.36 (0.2444)	No Learning
9	Non-essentials	2.12 (0.0000)	Different	3.47 (0.0629)	Learning Effects
10	Expressions	0.73 (0.9779)	Same	2.63 (0.1054)	No Learning
11	Methods	2.03 (0.0000)	Different	2.96 (0.0858)	Learning Effects
12	Synch	1.27 (0.0435)	Different	0.10 (0.7502)	No Learning
13	Modifiers	1.01 (0.4358)	Same	0.06 (0.8085)	No Learning
		Between groups: 105 Within groups: 979		Between groups: 1 Within groups: 1083	

Table 7: ANOVA Estimation Results.

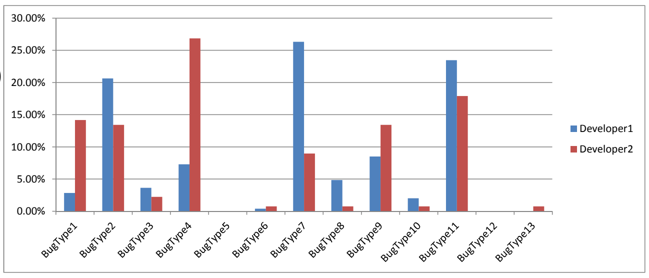

Figure 2: Distributions of bug types made by different developers in same project in the same year.

5.3 Different Bug Types across Developers

We explore whether different developers in the same project may make different kinds of bugs. Figure 2 shows the distributions (in %) of bugs of different types made by two different developers in the same project in the same year. Different developers may indeed make different kinds of bugs in the same project during the same time period, justifying the need of studies on personalized bug prediction [19, 31]. In addition, as shown in Table 2, most buggy code is fixed by a developer different from the one who commits the buggy code. This phenomenon provokes interesting future research to analyze the relationship between learning effects and factors that involve various forms of developer interaction.

We want to test the null hypothesis that bug ratio distributions are not statistically different across individuals. For this purpose, we estimate a multivariate F value (Wilks' λ) and Hotelling's trace [44], based on a comparison of the error variance/covariance matrix and the effect variance/covariance matrix. The statistics reject the null hypothesis. That is, different developers do overall make different kinds of bugs.

6. THREATS TO VALIDITY & FUTURE WORK

We now discuss forms of threats to validity in our study and our mitigation for the threats, and propose future work.

Threats to construct validity concern whether the measures we use are really related to the characteristics of developers, projects, and code changes they are supposed to measure. One main source of this kind of threats in our study is possible biases in bug identification. We follow common practices used in the literature [22, 36, 43] to identify bug fixes and discover bug origins to reduce biases. However, each `diff` related to a bug may still contain non-buggy code; we may need other techniques to help reduce falsely identified bugs [20, 45]. We also follow the literature to classify the types of bugs and bug fixes [32]; this kind of classification can be easily scaled to large code bases, but is mostly based on the syntax of code, not on the semantic or functionality of the code, nor bug severity or priority or difficulty. Thus, we have used topic modeling in the exploratory study to analyze bugs with respect to code of different "functionality" (or, topics). As future work, we can consider using more semantic-aware classification or root-cause analysis techniques to identify bug types [10, 26, 34, 47]. There are likely other factors that affect developers' learning effects but not considered in our analyses, such as the interactions among developers that are not recorded in the project repositories and implicit interactions happened through code (e.g., reading/changing each other's code). We plan in near future to extract such implicit interactions from code and other kinds of interactions captured in various data sources (e.g., bug reports, messages in mailing lists and discussion fora, wiki edits, etc.) to have further analyses on those factors.

Threats to internal validity concern the ability of our study to establish a valid link between measures for learning effects and other measures considered as independent variables, regardless of the validity of the measures themselves. We ensure the scripts and analysis code used in our study are implemented correctly. We use exploratory analyses to identify some factors that may affect learning effects, and then use regression models to analyze learning effects while controlling for those factors. As future work, we can try with different empirical models with more comprehensive set of data measures, including measures that consider interactions among developers across projects to provide more statistically significant analysis results.

Threats to external validity concern whether our analysis results can be generalized. Our empirical study includes multiple projects involving more than 100 developers over multiple years, and our analysis results are similar for all those projects. While these settings give evidences for the generalizability of our results, we only used five open source projects, mostly written in Java. More studies on projects using different languages, different business model (close-source), different development and maintenance processes would help to increase our confidence in our results.

One particular interesting direction for future work is to consider factors that can improve the effectiveness of learning. For example, "social coding" is touted as a better way to code [5, 6, 12, 28, 48, 49], examining factors in socially coded projects and comparing them with non-socially coded ones may provide insights how developers can learn more effectively from each other's coding experience.

7. CONCLUSION

The results have intriguing implications on project management about how to split efforts on tasks that add new code versus tasks that debug and fix existing code. Software project managers may consider assigning more testing and debugging tasks to a developer instead of coding new code, if a lot of new code is added by a developer. It helps, on one hand, to detect and fix possible bugs, and on the other hand, to improve developers' learning progress via bug fixing. It may also be useful for a developer to consider studying the fixes for her bugs more often even though she is not the person who fixes them. It would be even better if there are mechanisms available to facilitate collaboration and learning on bug fixing among developers.

Our analysis results show different bug ratios and different distributions of bug types in different projects, supporting project-specific and/or developer-specific bug prediction approach. This implies the need of transferring knowledge about bugs across projects for the purpose of bug prediction, which also justifies related work on across-project bug prediction [27, 30, 35, 46, 51].

8. REFERENCES

[1] R. Abreu and R. Premraj. How developer communication frequency relates to bug introducing changes. In *Joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 153–158, 2009.

[2] G. Abu, J. W. Cangussu, and J. Turi. A quantitative learning model for software test process. In *38th Annual Hawaii International Conference on System Sciences (HICSS)*, pages 78b–78b, 2005.

[3] J. D. Angrist and J.-S. Pischke. *Mostly harmless econometrics: An empiricist's companion.* Princeton University Press, January 2009.

[4] R. D. Banker, G. B. Davis, and S. A. Slaughter. Software development practices, software complexity, and software maintenance performance: A field study. *Management Science*, 44(4):433–450, Apr 1998.

[5] A. Begel, J. Bosch, and M.-A. Storey. Social networking meets software development: Perspectives from github, msdn, stack exchange, and topcoder. *IEEE Software*, 30(1):52–66, 2013.

[6] N. Bettenburg and A. E. Hassan. Studying the impact of social structures on software quality. In *IEEE ICPC*, pages 124–133, 2010.

[7] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *ISSRE*, pages 109–119, 2009.

[8] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. Germán, and P. T. Devanbu. The promises and perils of mining git. In *MSR*, pages 1–10, 2009.

[9] B. W. Boehm. *Software Engineering Economics.* Prentice Hall PTR, 1981.

[10] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong. Orthogonal defect classification—a concept for in-process measurements. *IEEE TSE*, 18(11):943–956, Nov 1992.

[11] O. Chouseinoglou, D. İren, N. A. Karagöz, and S. Bilgen. AiOLoS: A model for assessing organizational learning in software development organizations. *Information and Software Technology*, 55(11):1904–1924, 2013.

[12] L. A. Dabbish, H. C. Stuart, J. Tsay, and J. D. Herbsleb. Social coding in GitHub: transparency and collaboration in an open software repository. In *Computer Supported Cooperative Work (CSCW)*, pages 1277–1286, 2012.

[13] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *MSR*, pages 31–41, 2010.

[14] J. Fox. *Applied Regression Analysis and Generalized Linear Models.* SAGE Publications, Inc, 2nd edition, 2008.

[15] E. Giger, M. Pinzger, and H. C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *MSR*, pages 83–92, 2011.

[16] N. Hanakawa, S. Morisaki, and K.-i. Matsumoto. A learning curve based simulation model for software development. In *ICSE*, pages 350–359, 1998.

[17] D. E. Harter, M. S. Krishnan, and S. A. Slaughter. Effects of process maturity on quality, cycle time, and effort in software product development. *Management Science*, 46(4):451–466, Apr 2000.

[18] H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *ICSE*, pages 200–210, 2012.

[19] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *ASE*, pages 279–289, 2013.

[20] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *ICSE*, pages 351–360, 2011.

[21] S. Kim, E. J. W. Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE TSE*, 34(2):181–196, 2008.

[22] S. Kim, T. Zimmermann, K. Pan, and E. J. W. Jr. Automatic identification of bug-introducing changes. In *ASE*, pages 81–90, 2006.

[23] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *ICSE*, pages 489–498, 2007.

[24] M. S. Krishnan, C. H. Kriebel, S. Kekre, and T. Mukhopadhyay. An empirical analysis of productivity and quality in software products. *Management Science*, 46(6):745–759, Jun 2000.

[25] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro interaction metrics for defect prediction. In *SIGSOFT FSE*, pages 311–321, 2011.

[26] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *KDD*, pages 557–566, 2009.

[27] 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, Mar 2012.

[28] G. Madey, V. Freeh, and R. Tynan. The open source software development phenomenon: An analysis based on social network theory. In *Americas conf. on Information Systems (AMCIS)*, pages 1806–1813, 2002.

[29] N. Nagappan, T. Ball, J. Anvik, L. Hiew, and G. C. Murphy. Evidence-based failure prediction. *Making Software: What Really Works, and Why We Believe It*, page 415, 2010.

[30] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *ICSE*, pages 382–391, 2013.

[31] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Programmer-based fault prediction. In *6th International Conference on Predictive Models in Software Engineering*, page 19, 2010.

[32] K. Pan, S. Kim, and E. J. W. Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[33] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *FSE*, pages 2–12, 2008.

[34] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, Jan 1987.

[35] F. Rahman, D. Posnett, and P. T. Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *SIGSOFT FSE*, page 61, 2012.

[36] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. Bugcache for inspections: Hit or miss? In *ESEC/FSE*, pages 322–331, 2011.

[37] N. Ramasubbu and R. K. Balan. Globally distributed software development project performance: an empirical analysis. In *ESEC/SIGSOFT FSE*, pages 125–134, 2007.

[38] R. Reagans, L. Argote, and D. Brooks. Individual experience and experience working together: Predicting learning rates from knowing who knows what and knowing how to work together. *Management Science*, 51(6):869–881, 2005.

[39] F. E. Ritter and L. J. Schooler. The learning curve. In *International encyclopedia of the social and behavioral sciences*. Pergamon, 2002.

[40] S. Shivaji, E. J. W. Jr., R. Akella, and S. Kim. Reducing features to improve bug prediction. In *ASE*, pages 600–604, 2009.

[41] P. V. Singh, Y. Tan, and N. Youn. A hidden markov model of developer learning dynamics in open source software projects. *Information Systems Research*, 22(4):790–807, 2011.

[42] V. S. Sinha, S. Sinha, and S. Rao. BUGINNINGS: Identifying the origins of a bug. In *ISEC*, 2010.

[43] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR*, 2005.

[44] J. P. Stevens. *Applied Multivariate Statistics for the Social Sciences*. Routledge, 5th edition, 2009.

[45] F. Thung, D. Lo, and L. Jiang. Automatic recovery of root causes from bug-fixing changes. In *WCRE*, pages 92–101, 2013.

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

[47] S. Ugurel, R. Krovetz, and C. L. Giles. What's the code? automatic classification of source code archives. In *KDD*, pages 632–638, 2002.

[48] B. Vasilescu, V. Filkov, and A. Serebrenik. StackOverflow and GitHub: associations between software development and crowdsourced knowledge. In *International Conference on Social Computing (SocialCom)*, pages 188–195, 2013.

[49] T. Wolf, A. Schroter, D. Damian, and T. Nguyen. Predicting build failures using social network analysis on developer communication. In *ICSE*, pages 1–11, 2009.

[50] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. ReLink: Recovering links between bugs and changes. In *SIGSOFT FSE*, pages 15–25, 2011.

[51] 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 *ESEC/SIGSOFT FSE*, pages 91–100, 2009.