Empirical Study of Just-in-Time Prediction of Performance Regression Introducing Change

Kundi Yao, Jinfu Chen
Department of Software Engineering
Concordia University
Montreal, Canada
email: {ku_yao,fu_chen}@encs.concordia.ca

I. Introduction

The rise of large-scale software systems (e.g., Amazon.com and Google Gmail) has posed an impact on people's daily lives from mobile devices users to space station operators. The increasing importance and complexity of such systems makes their quality a critical, yet extremely difficult issue to address. Failures in such systems are more often associated with performance issues, rather than with feature bugs [1]. Therefore, performance assurance activities are an essential step in the release cycle of large software systems.

Performance assurance activities aim to identify and eliminate performance regressions in each newly released version. Examples of performance regressions are response time degradation, higher than expected resource utilization and memory leaks. Such regressions may compromise the user experience, increase the operating cost of the system, and cause field failures. The slow response time of the United States' newly rolled-out healthcare.gov illustrates the importance of performance assurance activities before releasing a system. Failure in detecting such regressions would result in significant financial and reputational repercussions.

Prior software quality research typically focus on functional bugs rather performance issues. For example, post-release bugs are often used as code quality measurement and are modelled by statistical modeling techniques in order to understand the relationship between different software engineering activities and code quality [2]. In addition, bug prediction techniques are proposed to prioritize software quality assurance efforts [3]–[5] and assesses the risk of code changes [6]. However, performance regressions are rarely targeted in spite of their importance.

On the other hand, prior study of JIT defect prediction focus more on the risk of commit based on bug rather than performance regressions. Predicting performance regressions remains a task that is conducted after fact, i.e., after the system is built and deployed in the field or dedicated performance testing environments. However, large amounts of resources are required to detect, locate, understand and fix performance regressions at such a late stage in the development circle; while the amount of required resources would be significantly reduced if developers were notified whether a code change introduces performance regressions during development.

In this research, we perform an empirical study on the JIT prediction of performance regression introducing code changes. By examining the identified performance regression introducing changes, we find that performance regression introducing changes are prevalent during software development. The identified performance regressions are often associated with complex syndrome, i.e., multiple performance metrics have performance regression. In order to build a change risk model to predict JIT performance regression, we combine the basic commit-level measures proposed by Mockus and Weiss [7], which are based on the characteristics of code changes, such as the number of modified subsystems and the purpose of the code change with the performance related measures we added, such as changing conditions and passing expensive parameters.

The rest of this proposal is organized as follows: Section II presents the prior research that is related to this project. Section III presents our subject systems and our approach of predicting performance regression introducing code changes. Section IV presents our research questions and result. Section V presents the threats to the validaty of our study. Finally, Section VI concludes this paper.

II. Related Work

In this section, we present the related prior research to this paper in three aspects: 1) Prediction of JIT software defect and 2) empirical study on performance.

A. Prediction of JIT software defect

Mockus and Weiss [7] are the first one to utilize a number of change measures to build linear regression to predict the probability of failure for a software change. Such change measures include size, duration, diffusion, and type, as well as the experience of the developers who implemented it. However, the discovered defect-inducing changes may be incomplete, which is a potential threat to the correctness of the prediction model. Kamei et al. [8]–[10] conduct a series of studies on defect prediction. Kamei et al. [8] present some change measures and builds a logistic regression model to predict just-in-time software defect. The change measures consist of 14 factors derived from six open-source projects and five commercial projects and are grouped into five dimensions. The prediction model the paper builded is able to predict defect-inducing changes with 68 percent accuracy and 64 percent recall. The finding also shows that which part of the factors play more important role in defect-inducing changes. Kamei et al. [9] construct JIT model based on cross-project models, by using larger pool of training data and combining models from other projects to establish their JIT models, and the result shows performance of within-project model outperforms cross-project model. Kamei et al. [10] present an overview in defect prediction field, which intends to introduce and help readers understand previous studies on defect prediction, and highlight some important challenges for future works.

To build the prediction model, researchers have to extract and select what kind of metrics to perform the prediction. Zhang et al. [11] build a universal defect prediction model for a large set of projects by combining context factors and clustering similar projects to determine the different software metrics sets. Shivaji et al. [12] realise that the more features prediction model learned, the more insufficient performance the model predicts, so they perform multiple feature selection algorithms to reduce the factors to predict software bug in a high performance. He et al. [13] study the feasibility of defect-predictor built with simplified metrics in different scenarios, and offer suggestions on choosing datasets and metrics, the result shows the predictor based on minimum metric subset, specific requirements of accuracy and complexity can provide satisfactory performance.

There are various kinds of classification models to predict software defect. Tourani et al. [14] build logistic regression models to study the impact of human discussion metrics on JIT predicting models, result shows a strong correlation between human discussion metrics and defect-prone commits. Tsakiltsidis et al. [15] use four machine learning methods to build models to predict performance bugs and the most satisfying model is based on Logistic Regression with all attributes added.In order to find whether or not unsupervised models perform better than the supervised models in effort-aware just-in-time defect prediction, Yang et al. [16] consider fourteen change metrics and build simple unsupervised and supervised models to predict software defect to determine whether they are of practical value. The results show that many simple unsupervised models perform better than the state-of-the-art supervised models in effort-aware just-in-time defect prediction.

Compared with above papers, we add another factors related to performance regression to build the prediction model. Another is that prior research utilize SZZ algorithm to identify whether or not a change will introduce a defect. However, if the repository does not report the defect we cannot map the defect back to the defect-inducing change. In our project, we focus on performance bugs. What is more, we run the performance test cases or benchmark so that we can collect the performance regression related to code changes.

B. *Empirical studies on performance*

Empirical studies are conducted in order to study performance issues. Jin et al. [17] studies 109 real world performance issues that are reported from five open source software. Based on the studied 109 performance bugs, Jin et al. [17] develop an automated tool to detect performance issues. Zaman et al. [18], [19] conducted both qualitative and quantitative studies on performance issues. They find that developers and users face problems in reproducing performance bugs. More time is spent on discussing performance bugs than other kinds of bugs. Huang et al. [20] studied real world performance issues and based on the findings, they propose an approach called performance risk analysis (PRA), to improve the efficiency of performance regression testing.

Luo et al. [21] propose a recommendation system, called PerfImpact, to automatically identify code changes that may potentially be responsible for performance regression between two releases. Their approach searches for input values that expose performance regressions and compare execution traces between two releases of a software to identify problematic code changes. Hasan et al. [22] create energy profiles as a performance measurement for different Java collection classes. They find that the energy consumption can have large difference depending on the operation.

Prior studies on performance typically are based on either limited performance issue reports or release of the software. However, the limit amount of issue reports and releases of the software hides the prevalence of performance regressions. In our paper, we evaluate performance at commit level. Therefore, we are able to identify more performance regressions and are able to observe the prevalence of performance regression introducing changes in development.

III. Approach

In this section we will explain our methodology in more detail. At first we depict our subject, including the open-source projects we choose and the experimental environment we set up. Then we present each step of our approach.

A. *Subject systems*

We choose two open-source projects, *Hadoop* and *RxJava* as the subject systems of our case study. *Hadoop* [23] is a distributed system infrastructure developed by the Apache Foundation. *Hadoop* performs data processing in a reliable, efficient, high fault tolerance, low cost and scalable manner. We choose *Hadoop* since it is highly concerned with its performance and has been studied in prior research in mining performance data [24]. *RxJava* is a library for composing asynchronous and event-based programs by using observable sequences and it carries the JMH benchmarks test options. *RxJava* is a *Java* VM implementation of reactive extensions. *RxJava* provides a slew of performance micro-benchmarks, making it an appropriate subject for our study. We choose the most recent releases of the two subject systems. The overview of the two subject systems is shown in Table I.

B. *Predicting performance regression introducing changes*

In this subsection, we present our approach of predicting performance regression introducing changes. The overview of our approach is shown in figure 1. In general, we extract every commit and measures from the version control repositories (Git) of our subject systems and identify impacted test cases of each commit. Afterward, we evaluate performance of each commit using either the related test cases or performance micro-benchmark. Then we perform statistical analysis on the performance evaluation results to identify performance regression. Finally, we build a prediction model based on the change measures to predict the JIT performance regression introducing changes.

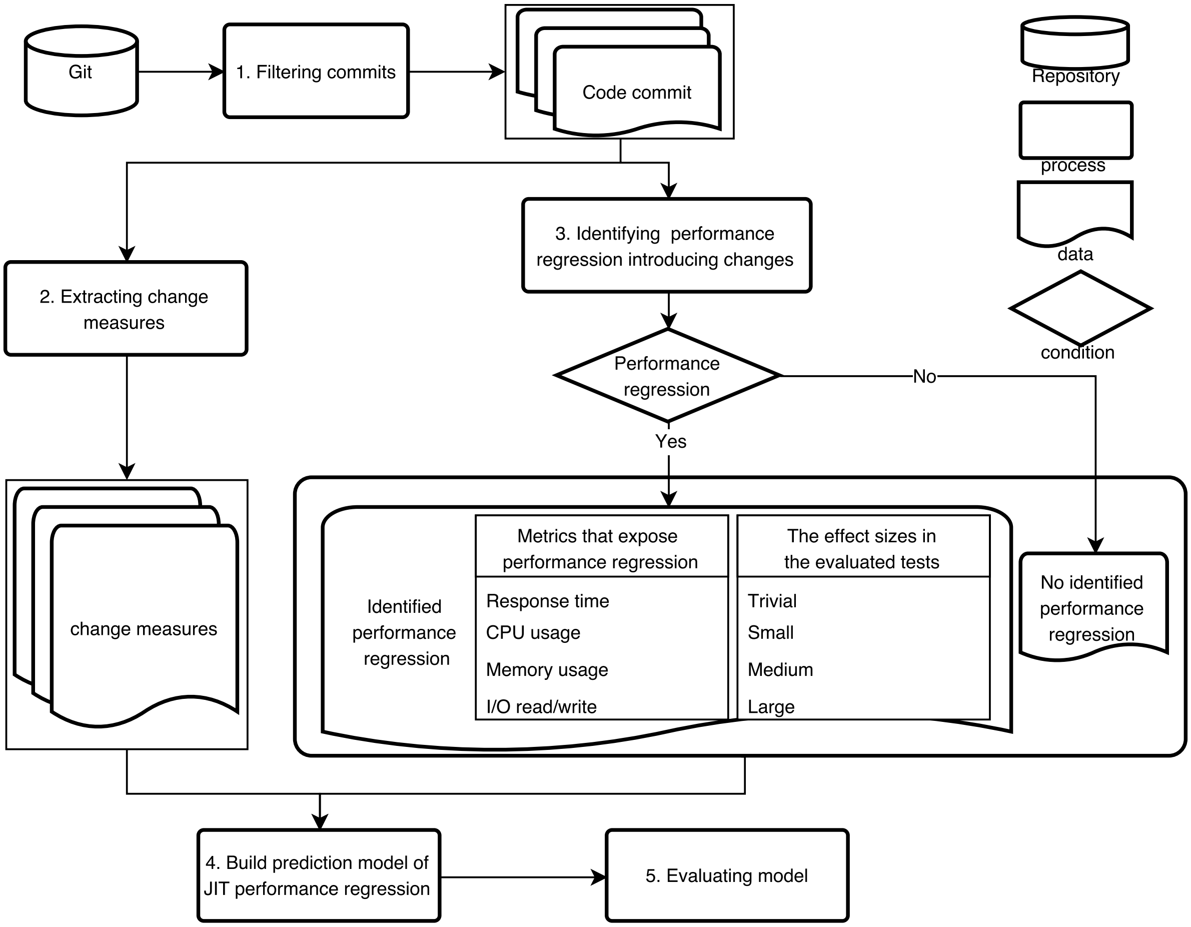
Git
1. Filtering commits
Code commit
Repository
process
data
condition
3. Identifying performance regression introducing changes
2. Extracting change measures
Performance regression
No
Yes
change measures
Identified performance regression
Metrics that expose performance regression
Response time
CPU usage
Memory usage
I/O read/write
The effect sizes in the evaluated tests
Trivial
Small
Medium
Large
No identified performance regression
4. Build prediction model of JIT performance regression
5. Evaluating model

Fig. 1: An overview of our approach that predicts performance regression introducing changes

TABLE I: Overview of our subject systems.

Subjects	Version	Total lines of code (K)	# files	# tests
Hadoop	2.6.0	1,496	6,086	1,664
	2.6.1	1,504	6,117	1,679
	2.6.2	1,505	6,117	1,679
	2.6.3	1,506	6,120	1,681
	2.6.4	1,508	6,124	1,683
	2.6.5	1,510	6,127	1,685
	2.7.0	1,552	6,413	1,771
	2.7.1	1,556	6,423	1,775
	2.7.2	1,562	6,434	1,784
	2.7.3	1,568	6,439	1,786
RxJava	2.0.0	164	1,107	76
	2.0.1	242	1,513	76
	2.0.2	243	1,524	76
	2.0.3	244	1,524	76
	2.0.4	244	1,526	76

1) Filtering commits: As the first step of our approach, we start off by filtering commits in order to focus on commits that are more likely to introduce performance regressions. In particular, we use *git log* command to list all the files that are changed in each commit. We only extract the commits that have source code changes, i.e., changes to *.java* files.

In practice, there may exist multiple commits that are made to accomplish one task, making some of the commits temporary. We would like to avoid considering the performance regressions that are introduced in such temporary commits. Since *Hadoop* uses JIRA as their issue tracking system and *RxJava* uses the internal issue tracking system in Github, we use the issue id that is mentioned in each commit message to identify the task of each commit. If multiple commits are associated with the same issue, we only consider the snapshot of the source code after the last commit.

2) Extracting change measures: To conduct our research, we extract the domain commit-level change measures and the file-level performance-relevant change measures from the CVS repositories of the projects. The overview of the change measures is shown in Table II.

Commit-level change measures. Inspired by previous studies from *Mockus et al.* and *Kamei et al.* [7] [8], we inherit their change measures and extend these measrues to performance prediction area. We consider 12 change measures grouped into five dimensions. (i)

1) **Diffusion.** We use the diffusion of changes as metrics to indicate the probability that a change can introduce performance regression. In this dimension, first we investigate NS (Number of modified subsystems), ND (Number of modified directories) and NF (Number of

modified files). From filtered commits, we use *git diff* command to compare every two consecutive commits and we get a list of changed files for every commit, each file path is divided into three parts: subsystem, directory and file, as Fig 2 shows, name of each subsystem is defined by the root directory name of each path, *.java* file part represents the name of file, and the remaining part in the middle is used to identify the name of a directory. The correspoding measures can be conducted according to their occurance. As for Entropy, we calculate Shannon entropy using the following equations:

$$Entropy = -\sum_{i=1}^{n} (p_i * \log_2 p_i) \tag{1}$$

$$p_i = \frac{LOC\ changes\ in\ file\ i}{LOC\ changes\ in\ all\ files} \tag{2}$$

i means the number of files, and logarithm is used to normalize the data. Larger entropy value implies more information transmitted, which indicates more changes are caused in this context.

2) **Size.** We use LOC (Lines of Code without comment line and empty lines) as metrics to evaluate the Size changes. LA (LOC added) and LD (LOC deleted) can be obtained directly from the result of *git diff*, and we use *cloc* to get LT (LOC before the change). *cloc* is an open-source project designed to count lines of source code in many programming languages. The larger size of a change, the higher probability of introducing performance regression.

3) **Purpose.** In this dimension we mainly investigate the rationale of current changes. When programmers decide to change the code, their initial purpose can vary a lot, such as fixing bug, adding new features, improving features, etc. However, bug fix sometimes can be a dangerous action because it may unintentionally introduce new defects, which have high probability to cause performance regression. In our approach, we use issue tracking systems to define whether the type of current commit is a bug fixing. We use *JIRA*, a widely adopted commercial issue tracking system, to gather information about hadoop commits. Unlike hadoop, Rxjava does not have its own independent JIRA database, but we can also fetch detailed issue reports from Github for further analysis. We use scripts to crawl issue type of each commit from both JIRA and Github, because their information is authentic and up to date. Then we can determine whether a change is because of bug fixing or not.

4) **History.** In History dimension, we mainly discuss two kinds of change measures: NDEV (The number of developers touched a file) and AGE (The average time interval between last and current change). Previous findings indicate that files touched by more people are more likely to introduce a problem [25], and recent changes are easier to cause faults [26]. Those unexpected deficiencies, as by-product brought by changes, are highly risky to cause performance degradations. We use *git blame* command followed by touched file name, this will display who and when was each line of code touched. We count the number of different developers to obtain NDEV value. As for AGE, because we already have a list of touched files between two consecutive commits, we calculate the average time of each file and save them into a list. Because the value of time parameter is relatively large, so we need to preprocess our data before we continue our analysis.We use the following formula to normalize and rescale the data.

$$AGE = \frac{T_{Avg} - T_{Min}}{T_{Max} - T_{Min}} \tag{3}$$

The three parameters in the equation refer to the average,Inspired by previous studies from *Kamei et al.* and *Mockus et al.*, we inherit their change measures and extend these measrues to performance prediction area. maximum and minimum number in previous list.

5) **Experience.** The Experience dimension is to evaluate the programmer familiarity with the system. In our study, we mainly consider EXP (Developer experience) and REXP (Recent developer experience) as important indicators to quantify programmers' experience with a system. Increasing familiarity can reduce the likelihood of making mistakes [7], and experienced specialists can capture "best practice" in software design to diminish software performance problems [27]. Because all of our commits are from most recent releases, we acquire information of all contributors from *Open Hub*, a public directory of open source software with up to date information. Those information contains the total number of commits to a project, and the number of commits within recent 12 months, which can act as measures to developers' experience in our study. We crawl EXP and REXP data of all the contributors from *OpenHub* and save them into different local files, the next step is to match contributor information to different commits. For RxJava, first we use *git log* command to acquire all the commits and name of contributors. Therefore, we can merge those results into a larger local dataset according to name matching, and we can query information of contributors from it using commits. However, the situation in hadoop is completely different. Unlike Rxjava, there exists an agency between contributors and source repository. The middle layer is called Hadoop Project Management Committe (PMC), which contains people with direct access to hadoop repository. When programmers want to make a change in hadoop, their changes will be reviewed by a PMC member to ensure the quality. If it is approved, the reviewer will submit to hadoop repository with information *"Contributed by RealContributorName"*. In consequent, from git command we can hardly tell the real contributor of a commit because only the information of commiter will be shown. So we crawling names of real contributors from JIRA and query contributor experience data from local file by name matching.

Performance-relevant change measures. Simultaneously,

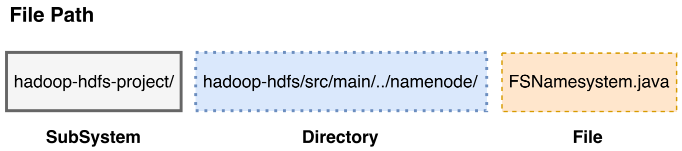

Fig. 2: The division of file path

TABLE II: Summary of domain and performance-relevant change measures

Dim.	Name	Definition
Diffusion	NS	Number of modified subsystems
	ND	Number of modified directories
	NF	Number of modified files
	Entropy	Distribution of modified code across files
Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code before the change
Purpose	FIX	Whether or not the changes fix a bug
History	NDEV	Number of developers that changed the modified files
	AGE	The average time interval between the last and the current change
Experience	EXP	Developer experience
	REXP	Recent developer experience
Perf.	CC	Number of changing condition
	CL	Number of changing loop
	IL	Number of introducing locks or synchronization
	EV	Number of expensive variable
	FC	Number of changing function call

we extract performance-relevent change measures automatically, including CC (changing conditions), CL (changing loops), FC (changing function calls), IL (Introducing locks and synchronization) and EV (expensive variables). The first three measures are most relevent to performance [21]. Also, after we find the performance regression we check the source code manully and find that IL and EV are also the most root-cause of regression.

CC can change the code that is executed and may cause more operation eventually executed by the software, leading to performance regressions. CL may significantly slow down performance. Locks are expensive actions for software performance. IL means that introducing locks and synchronization can suspend threads waiting on a lock until released, causing performance degradation on response time. FC is that developers may introduce expensive function calls with the function execution expensive by itself or executed a large number of time. EV means that some variables are more expensive to be held in memory and need more resources to visit or operate.

To extract performance-relevant change measures, we only need to focus on the file corresponding test case. We utilize tool *srcML* [28] to convert the source code of the same file from current commit and its parent commit to XML file. Afterward, we employ *diff* tool and regular expression to compare the source code.

3) Identification of performance regression introducing changes: In more detail, we give a list of particular step:

Identifying impacted tests. In order to evaluate performance of each code commit, we use the tests and performance micro-benchmarks that are readily available in the source code of our subject systems. As mature software projects, each subject system consist of a large amount of test cases. For example, *Hadoop* release2.7.3 contains 1786 test cases in total. Exercising all test cases may cause two issues to our performance evaluation: 1) the test cases that are not impacted by the code change would dilute the performance impact from the code changes and introduce noise in the performance evaluation and 2) the large amounts of un-impacted test cases would requires extra resources for performance evaluation (e.g., much longer test running time).

Therefore, in this step, we leverage a heuristic to identify impacted tests for each commit. In particular, we find that *Hadoop* test cases follow a naming convention that the name of the test files contain that same name of the source code files being tested. For example, a test file named *TestFSNamesystem.java* tests the functionality of *FSNamesystem.java*. Hence, for each changed source code file in a commit, we automatically identify the test files.

Leveraging micro-benchmarks for *RxJava*. Fortunately, *RxJava* provides a slew of micro-benchmarks with the goal of easing performance evaluation. We find that these performance micro-benchmarks are designed to evaluate performance of the software as a cross-cutting concern, instead of evaluating any particular features separately. Therefore, we opt to run all 76 micro-benchmarks from *RxJava*. In the rest of this paper, we also refer these micro-benchmarks as test cases to ease the description of our results.

Evaluating performance. In this step, we exercise the prepared test cases and the performance micro-benchmarks to evaluate performance of each commit. We setup our performance evaluation environment based on Azure node type Standard F8s (8 cores, 16 GB memory). In order to generate statistically rigorous performance results, we adopt the practice of repetitive measurements [29] to evaluate performance. In particular, each test or performance micro-benchmark are executed 30 times independently. We collect both domain level and physical level performance metrics during the tests. We measure the response time of each test case as domain level performance metric. A shorter response time indicating better performance of the software. We use a performance monitoring software named *psutil* [30] to monitor physical level performance metrics, i.e., the CPU usage, Memory usage, I/O read and I/O write of the software, during the test.

Statistical analyses on performance evaluation. Statistical tests have been used in prior research and in practice to detect whether performance metric values from two tests reveal performance regressions [31]. After having the performance evolution results, we perform statistical analyses to determine the existence and the magnitude of performance regression in a statistically rigorous manner. We use Students t-test to examine if there exists statistically significant difference (i.e., p-value < 0.05) between the means of the performance metrics. A p-value < 0.05 means that the difference is likely not by chance. A t-test assumes that the population distribution is normally distributed. Our performance measures should be approximately normally distributed given the sample size is large enough according to the central limit theorem [32]. T-test would only tells us if the differences of the mean between the performance metrics from two commits are statistically significant. On the other hand, effect sizes quantify such differences.

Researchers have shown that reporting only the statistical significance may lead to erroneous results (i.e., if the sample size is very large, p-value can be small even if the difference is trivial). We use *Cohen's d* to quantify the effects [33]. *Cohen's d* measures the effect size statistically and has been used in prior engineering studies [34], [35]. *Cohen's d* is defined as:

$$Cohen's\ d = \frac{mean(x1) - mean(x2)}{s}$$

where *mean(x1)* and *mean(x2)* are the mean of two populations, and s is the pooled standard deviation [36].

$$effect\ size = \begin{cases} trivial & \text{if } Cohen's\ d \leqslant 0.2 \\ small & \text{if } 0.2 < Cohen's\ d \leqslant 0.5 \\ medium & \text{if } 0.5 < Cohen's\ d \leqslant 0.8 \\ large & \text{if } 0.8 < Cohen's\ d \end{cases}$$

4) Data Preprocessing: Before utilize the attributes to build our prediction models, we need to employ data preprocessing to make sure the data quality, including accuracy, completeness, consistency and timeliness. In our project, we perform data cleaning to fill in missing values, data transformation to normalize the raw data, and remove redundant data.

Missing value happens in the *FIX, EXP and REXP* in our study. Because a few commits without issue report so we cannot extract the purpose of this commit (FIX). And a few commits' contributors are not in the *Hadoop* contributor official list [37] so we cannot calculate the measures EXP and REXP. In this case, we use a global constant to fill in the missing value. We replace all missing values by the same constant "Unknown" and "1" in measure FIX and EXP & REXP, respectively.

Data normalization or standardization can help avoid data skew and give all measures an equal weight. The values of different attributes may have large different range due to various measurement units. It tends to give such an attribute with smaller units greater effect or weight. In particular, we perform *Min-max normalization* to transform our original data.

Redundancy is an essential issue in the data integration, which means that a measure may be redundant if it is derived from another measure or a set of measures. We employ correlation analysis to remove highly correlated measures. In particular, we use *Correlation-based Feature Selection* and *Information Gain Rate* technique to solve this problem.

5) Build prediction model of JIT performance regression: **Model building.** TFirst, we want to build logistic regression model to predict the existence of performance regression introducing change, we will build th. We choose logistic regression model (logistic regression classification can be easy to interpret) firstlybecause it is easy to interpre. Second, we build a ordinal model to predict wthether or not the change causes regression. And to predict the magnitude of performance regression introducing change, we build a ordinal model to predict the effect size effect size of performance regression. Our ordinal class consists of not significant difference, small, medium and large regression. Standard classification algorithms consider the class value as a nominal quantity so that it cannot operate this ordering information. Thrid, we use the coefficient and odds ratio to compare and find the important change measures of performance regression.

Model evaluation. We utilize three metrics to evaluate classifier performance, including *precision*, *recall* and *AUC* [38]. It depends on confusion matrix which contains four terms consisting of *true positives, true negatives, false positives and false negatives*. We perform stratified k-fold cross-validation for estimating accuracy and determine the *k* as 10 due to its relatively low bias and variance.

In particular, we use the most popular machine learning tool *Weka* [39] to train the dataset and build the corresponding model. Fristly we build the model and predict the text data within project. Afterward, we employ the model to predict text data cross-project.

IV. Case Study

Our study aims to answer three research questions. For each research question, we present the motivation of the question, the approach that we use to answer the question.

RQ1: How well can we predict the existence of performance regression introducing change?

Motivation. Prior research has conducted empirical studies on JIT quality assurance [6], using the reported bugs in issue reports. However, there may exist much more performance issues, such as performance regressions, that are not reported as JIRA issues. First, we evaluate performance of on each code commit instead of depending on JIRA issues. Intuitively, we may uncover instances of performance regressions that are not reported. Second, in this research question, we focus on the prediction of performance regression rather functional bugs.

Approach. To answer RQ1, we build a logistic regression prediction model for the risk of performance regression introducing change baesd on the commit-level and file-level measures in Table II. With the approach presented in Section III, we obtain the results of performance evaluation (1 is regression, 0 is not regression) for every test case in our subject systems. In particular, we not only consider a test having performance regression if the response time is statistically significantly longer, but also think over the resource utilization. Sometimes performance regressions may not cause impact on response time but rather cause a higher resource utilization. The high resource utilization, although may not directly impact user experience, may cause extra cost when deploying, operating and maintaining the system, with lower scalability and reliability. Therefore, we also use the physical metrics, i.e., CPU usage, Memory usage, I/O read and I/O write, as measurements of performance regressions.

To validate how well the model predict performance regression introducing changes, we use two metrics, *precision* and *recall* to measure the model. At the same time, to verify the stability of the prediction, we employ 10-fold cross validation to test the prediction model.

Results. Performance regressions are not rare instances. We find 243 and 91 commits that contain at least one test with performance regression in at least one performance metric for *Hadoop* and *RxJava*, respectively. In total of 1,270 executed tests from *Hadoop* and 7,600 executed tests from *RxJava*, 129 and 1,410 have statistically significantly slower response time with medium or large effect sizes, respectively. When examining the effect sizes of the detected performance regressions, we find that there exist more performance-regression-prone tests with large effect sizes than medium (see Table III). In addition, we detect more tests with performance regressions in CPU and Memory usage, than other performance metrics. Since CPU and Memory usage both have large impact on the capacity of the software systems, these regressions may impact reliability or financial cost of the software system.

Our predictor achieves an average precision of 70.3% and recall of 82.1%. We employ our prediction model into these two systems and the result is shown in Table IV. We can find that the average precision of 72.5% in *Hadoop* is higher than the average precision of 68.2% in *Rxjava*. We infer that there are two reasons: 1) It is the size of the dataset that causes this distinction. The number of tuple is 1120 in *Hadoop* and 7600 in *Rxjava*. But sometimes excessive data can lead to counteractive result, because the prediction model cannot be well fitting to all the data. Redundant data can even result in overfitting, which can significantly decrease the predictive performance. 2) The measures EXP and REXP are incomplete (missing value) in *Hadoop* and we fill in the missing value by using a global constant. The filled-in value may not be correct and bias the original data.

Our predictor performs better in the class of *IO* regression. In the four classification types, we find that the preditor performs better in the metrics of *IO* in *Hadoop* (76.3%) and in *Rxjava* (70.8%). This is because unlike *CPU* and *Memory*, which allow dynamic resources allocattion according to different context, *IO* excludes interference from external factors and enables the model to capture more accurate changing patterns.

We find that performance regression introducing changes are prevalent phenomenon. Logistic regression model can achieve high precision in the prediction of the existence performance regression.

RQ2: How well can we predict the magnitude of performance regression introducing change?

Motivation. In RQ1, we find that our predictor performa well in prediction of the existence of performanve regression. If we can predict the magnitude of these performance regressions, we may provide guidance or automated tooling support for developer to prevent the large effect on performance regression.

Approach. In RQ1, we find that there exist prevalent performance regressions that are introduced by code changes. To address RQ2, we build a ordinal prediction model for the magnitude of performance regression introducing change based on the measures (see Table II) and effect size of the regression. we obtain the results of effect size (large, medium, small regression and not significant difference) for every test case in our subject systems. We also consider four types of performance counters.

We follow two steps in our approach to predicting the magnitude of performance regression. First, we use *Weka* to build the ordinal model OrdinalClassClassifier [40], which allows standard classification algorithms to be applied to ordinal class problems to predict the effect size of regression in four performance metrics, *Runtime, CPU, Memory and IO*. Second, we compare the accuracy between different metrics and compare the efficiency between ordinal model with the logistic regression model builded in RQ1.

To validate how well the model predicts the magnitude of performance regression introducing changes, we also use two metrics, *precision* and *recall*. And perform 10-fold cross validatoin to verify the stability of the prediction.

Results. We find that there exist more performance-regression-prone tests with large effect sizes than medium and small (see Table III). Such results imply that developers may not ignore these performance regressions since they may have large impact on system performance.

Our predictor achieves an average precision of 54% percent and recall of 66% percent. We employ our magnitude prediction model into these two systems and the result is shown in Table V. We can find that the average precision of 59% percent in *Hadoop* is higher than the average precision of 48% percent in *Rxjava*. The average precision of 60% in performance counter *Runtime* is higher than other three performance counters (around 50%). We infer that the correlation is low between the effect sizes calculated with runtime and the physical counters. On the other hand, the effect sizes calculated with the physical counters may have medium or large correlations with each other. We believe that these correlations are often due to the nature of the software itself (e.g., database accessing software may have its CPU usage highly correlated with I/O).

Our predictor performs better in the large effect than medium effect on performance regression. 189 tests are large effect and 149 tests are medium effect on regression in *Hadoop*. And 1745 benchmarks are large effect and 1355 benchmarks are medium effect on regression in *Rxjava* (see Table III). The average precision of 56% percent in large effect is higner than the average precision of 47% percent in medium effect on regression (see Table VI). It reflects that the change causing regression with large effect size carries more abvious change features. It is super meaningful because most developers focus more on the performance regression with large effect because in many cases, performance regression with large effect can cause large effect on users. Similarly, the precision of 69.5% in *Runtime* is higner than other performance counters in different effect sizes.

Magnitude predictor performs worse than the existence predictor. The average precision of 70.3% percent in the existence predictor is higner than the average precision of 54% percent in the ordinal magnitude predictor. We infer the reason is that the two kinds of models train on the same dataset (change measure is the same). However, in the ordinal predictor it need to renognize five classes rather two classes in the logistic regrssion model. To imropve the accuracy of ordinal model, we should add more change measures based on the measures in the logistic model.

TABLE III: Results of identifying performance regression introducing changes in different metric classifications.

Total number of tests with performance regressions in different metrics.												
	Total executed tests	Any metric	Response time		CPU		Memory		I/O read		I/O write	
			large effect	medium effect	large effect	medium effect	large effect	medium effect	large effect	medium effect	large effect	medium effect
Hadoop	1,270	338	87	42	202	97	167	74	75	28	75	17
RxJava	7,600	3,100	745	665	659	487	919	489	657	449	38	0

TABLE IV: Precision and recall of predicting the existence of performance regression.

	Runtime(%)		CPU(%)		Memory(%)		IO(%)	
	pre.	rec.	pre.	rec.	pre.	rec.	pre.	rec.
Hadoop	70.5	83.8	69.1	80.0	73.9	85.8	76.3	77.2
Rxjava	68.1	82.5	70.0	83.6	63.7	79.8	70.8	84.1
Average	69.3	83.2	69.6	81.8	68.8	82.8	73.6	80.7

TABLE V: Precision and recall of predicting the magnitude of performance regression.

	Runtime(%)		CPU(%)		Memory(%)		IO(%)	
	pre.	rec.	pre.	rec.	pre.	rec.	pre.	rec.
Hadoop	67.5	73.5	48.8	54.1	54.7	64.9	65.7	72.8
Rxjava	52.2	64.2	45.7	67.6	45.9	61.5	48.5	69.7
Average	59.9	68.9	47.3	60.8	50.3	63.2	57.1	71.3

Our predictor achieves an average precision of 54% percent and recall of 66% percent. We find that there exist more performance-regression-prone tests with large effect sizes than medium. Our predictor performs better in the large effect than medium effect on performance regression.

RQ3: What are the important measures of performance regression introducing change?

Motivation. A lot of change measures have impact on the performance regression and Each change measure has different impact on the software quality [6]. If we can find the important change measures of performance regerssion we may give the guidance to support developer to focus more on the particular factors.

Approach. To address RQ3, we analyze and compare the regression coefficients of the logistic models from RQ1. To measure the effect of every change metric, we use the value of coefficients and odds ratio [41] in the logistic regression model. The coefficients are in fact the weights that are applied to each attribute before adding them together. We can keep all of the metrics at their original value, except for the metric whose effect we wish to measure. The effect of a metric can be positive or negative. A positive effect means that a higher value of the factor increases the likelihood, whereas a negative effect means that a higher value of the factor decreases the likelihood of the dependent variable. The odds ratio is the exponent of the corresponding coefficient in the logistic regression model. It indicates how large of an influence a change to that value will have on the prediction. We will compare the two indices and find the more important measures, and try to interpret why the measures are more important.

TABLE VI: Precision of predicting large and medium performance regression.

	Runtime(%)		CPU(%)		Memory(%)		IO(%)	
	large	med	large	med	large	med	large	med
Hadoop	65.6	34.2	55.3	45.4	41.7	48.5	46.8	43.4
Rxjava	73.3	51.6	67.6	52.4	65.8	49.2	32.1	39.6
Average	69.5	47.9	61.5	48.9	53.8	48.8	39.5	41.5

TABLE VII: Coeffients and odds ratio with the corresponding measures in the logistic regression model.

		Runtime	CPU	Memory	IO
Coefficients	Hadoop	AGE(2.39)	LD(0.40)	LT(0.74)	LT(1.38)
		LD(1.47)	En.(0.17)	LD(0.63)	LA(1.15)
		LA(1.04)	Syn(0.15)	Loop(0.57)	Loop(0.19)
	Rxjava	LD(0.74)	Syn(0.48)	LD(1.29)	Fix(0.07)
		Syn(0.18)	En.(0.45)	Syn(0.21)	Loop(0.03)
		Fix(0.09)	Fix(0.38)	En.(0.13)	En.(0.02)
Odds ratio	Hadoop	AGE(10.6)	LD(1.49)	LT(2.11)	LT(3.97)
		LD(4.35)	En.(1.19)	LD(1.87)	LA(3.18)
		LA(2.84)	Syn(1.13)	Loop(1.71)	Loop(1.19)
	Rxjava	LD(2.22)	Syn(1.62)	LD(3.64)	Fix(1.07)
		Syn(1.18)	En.(1.51)	Syn(1.23)	Loop(1.02)
		Fix(1.09)	Fix(1.41)	En.(1.12)	En.(1.02)

Results. Change measure in the dimension of *size* are more important than other dimension of change measures in the commit-level metrics. LA are more important change measure in the dimension of *size*. The result of coefficients is shown in Table VII. Due to limited space, we only show the top three coefficients and odds ratios with the coresponding measures. Size dimension plays an risk-increasing role in the performance regression introducing changes. The finding is similar to prior research. Measures LT and LA in the dimension of size are more important than other measures.

We also find that the risk-increasing measures are different between different performance counters. In the domain level metric of *Runtime*, the change measure *AGE* is more important than other measures except the *size* change measures. However, in the physical metrics, the performance-relevant change measures such as *Loop* and *Syn* are more important. So the important change measures depend on the performance conter class.

Loop and Syn is more important other change measures in the dimension of performance-relevant meteics. Our study focused on performance regression and

Our predictor achieves an average precision of 54% percent and recall of 66% percent. We find that there exist more performance-regression-prone tests with large effect sizes than medium. Our predictor performs better in the large effect than medium effect on performance regression.

V. Threat to Validity

A. External Validity

Generalizing our results. In our case study, we only focus on fifteen releases from two open source systems, i.e., *Hadoop* and *RxJava*. Both of the subject systems are mainly written in *Java* languages. Some of the findings might not be generalizable to other systems or other programming languages. Future studies may consider more releases from more systems and even different programming languages (such as C#, C++).

B. Internal Validity

Subjective bias of manual analysis. The manual analysis for root-causes of performance regression is subjective by definition, and it is very difficult, if not impossible, to ensure the correctness of all the inferred root-causes. We classified the root-causes into six categories; however, there may be different categorizations. Combining our manual analysis with controlled user studies on these performance regressions can further address this threat.

Selection of performance metrics. Our approach requires performance metrics to measuring performance. In particular, we pick one commonly used domain level and four commonly used physical level performance metrics based on the nature of the subject systems. There exist a large number of other performance metrics. However, practitioners may require system-specific expertise to select an appropriate set of performance metrics that are important to their specific software. Future work can include more performance metrics based on the characteristic of the subject systems.

C. Construct Validity

Monitoring performance of subject systems. Our study is based on the ability to accurately monitor performance of our subject systems. This is based on the assumption that the performance monitoring library, i.e. *psutil* can successfully and accurately providing performance metrics. This tool monitoring library is widely used in performance engineering research [29], [42]. To further validate our findings, other performance monitoring platforms (such as PerfMon [43]) can be used.

Issue report types. We depend on the types of issues that are associated with each performance regression introducing commit. The issue report type may not be entirely accurate. For example, developers include extra code changes in issue reports with type *documentation*. Firehouse-style user studies [**?**] can be adopted to better understand the context of performance regression introducing changes.

VI. Conclusion

Techniques that are designed to detect performance regression are done after the system is deployed in performance testing or user environment. However, detected performance regressions are difficult to fix at this late stage. In this paper, we conduct an empirical study on performance regression introducing changes in two open source software *Hadoop* and *RxJava*. We evaluate performance of every commit by executing impacted tests or performance micro-benchmarks. By comparing performance metrics that are measured during the tests or performance micro-benchmarks, we identify and study performance regressions introduced by each commit. In particular, this paper makes the following contributions:

- To the best of our knowledge, our work is the first to evaluate and to study performance regressions at the commit level.
- We propose a statistically rigorous approach to identifying performance regression introducing code changes. Further research can adopt our methodology in studying performance regressions.
- We find that performance regressions widely exist during development, and often are introduced after bug fixing.
- We find six root-causes of performance regressions that are introduced by code changes. 12.5% of the manually examined regressions can be avoided or their performance impact may be reduced.

Our findings call for the need of frequent performance assurance activities (like performance testing) during software development, especially after fixing bugs. Although such activities are often conducted before release [**?**], while developers may find it challenging since many performance issues may be introduced during the release cycle. In addition, developers should resolve performance regressions that are avoidable. For the performance regressions that cannot be avoided, developers should evaluate and be aware of their impact on users. If there exist a large impact on users, strategies, such as allocating more computing resources, may be considered. Finally, in-depth user studies and automated change impact on performance are future directions of this work.

References

[1] E. Weyuker and F. Vokolos, “Experience with performance testing of software systems: issues, an approach, and case study,” *Transactions on Software Engineering*, vol. 26, no. 12, pp. 1147–1156, Dec 2000.

[2] A. E. Hassan, “Predicting faults using the complexity of code changes,” in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88.

[3] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for eclipse,” in *Predictor Models in Software Engineering, 2007. PROMISE’07: ICSE Workshops 2007. International Workshop on*, May 2007, pp. 9–9.

[4] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE ’05. New York, NY, USA: ACM, 2005, pp. 284–292.

[5] N. Nagappan, T. Ball, and A. Zeller, “Mining metrics to predict component failures,” in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE ’06. New York, NY, USA: ACM, 2006, pp. 452–461.

[6] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A large-scale empirical study of just-in-time quality assurance,” *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, June 2013.

[7] A. Mockus and D. M. Weiss, “Predicting risk of software changes,” *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[8] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A large-scale empirical study of just-in-time quality assurance,” *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, June 2013.

[9] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2072–2106, 2016.

[10] Y. Kamei and E. Shihab, "Defect prediction: Accomplishments and future challenges," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 5. IEEE, 2016, pp. 33–45.

[11] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 182–191.

[12] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 552–569, April 2013.

[13] P. He, B. Li, X. Liu, J. Chen, and Y. Ma, "An empirical study on software defect prediction with a simplified metric set," *Information and Software Technology*, vol. 59, pp. 170–190, 2015.

[14] P. Tourani and B. Adams, "The impact of human discussions on just-in-time quality assurance: An empirical study on openstack and eclipse," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 189–200.

[15] S. Tsakiltsidis, A. Miranskyy, and E. Mazzawi, "On automatic detection of performance bugs," in *Software Reliability Engineering Workshops (ISSREW), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 132–139.

[16] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 157–168.

[17] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 77–88.

[18] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: a case study on firefox," in *Proceedings of the 8th working conference on mining software repositories*. ACM, 2011, pp. 93–102.

[19] S. Zaman, B. Adams, and Hassan, "A qualitative study on performance bugs," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, ser. MSR '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 199–208.

[20] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance regression testing target prioritization via performance risk analysis," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 60–71.

[21] Q. Luo, D. Poshyvanyk, and M. Grechanik, "Mining performance regression inducing code changes in evolving software," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 25–36.

[22] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy profiles of java collections classes," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 225–236.

[23] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[24] M. D. Syer, W. Shang, Z. M. Jiang, and A. E. Hassan, "Continuous validation of performance test workloads," *Automated Software Engineering*, vol. 24, no. 1, pp. 189–231, 2017.

[25] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM, 2010, p. 18.

[26] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[27] C. U. Smith and L. G. Williams, "More new software performance antipatterns: Even more ways to shoot yourself in the foot," in *Computer Measurement Group Conference*. Citeseer, 2003, pp. 717–725.

[28] "srcml," 2017. [Online]. Available: http://www.srcml.org/

[29] T.-H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 666–677.

[30] "psutil," 2017. [Online]. Available: https://github.com/giampaolo/psutil

[31] H. M. Alghmadi, M. D. Syer, W. Shang, and A. E. Hassan, "An automated approach for recommending when to stop performance tests," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 279–289.

[32] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1001–1012.

[33] L. A. Becker, "Effect size (es)," *Accessed on October*, vol. 12, no. 2006, pp. 155–159, 2000.

[34] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. Sjøberg, "A systematic review of effect size in software engineering experiments," *Information and Software Technology*, vol. 49, no. 11, pp. 1073–1086, 2007.

[35] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on software engineering*, vol. 28, no. 8, pp. 721–734, 2002.

[36] J. Hartung, G. Knapp, and B. K. Sinha, *Statistical meta-analysis with applications*. John Wiley & Sons, 2011, vol. 738.

[37] A. Hadoop, "Apache hadoop open source project: Contributions listing," 2017. [Online]. Available: https://www.openhub.net/p/Hadoop/contributors/

[38] "An introduction to {ROC} analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861 – 874, 2006", note = {ROC} Analysis in Pattern Recognition, issn = 0167-8655, author = Tom Fawcett.

[39] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009.

[40] E. Frank and M. Hall, *A Simple Approach to Ordinal Classification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 145–156.

[41] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan, "Understanding the impact of code and process metrics on post-release defects: A case study on the eclipse project," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10. New York, NY, USA: ACM, 2010, pp. 4:1–4:10.

[42] T. M. Ahmed, C.-P. Bezemer, T.-H. Chen, A. E. Hassan, and W. Shang, "Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: An experience report," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 1–12.

[43] R. Enbody, "Perfmon: Performance monitoring tool," *Michigan State University*, 1999.