Exploring Correlation Among Different Software Measurement Metrices

Nirav Ashvinkumar Patel
Dept. of Engineering and Computer Science
Concordia University
Montreal, Canada
p_nir@live.concordia.ca

Himansi Maheshkumar Patel
Dept. of Engineering and Computer Science
Concordia University
Montreal, Canada
hi_t@live.concordia.ca

Krishnan Krishnamoorthy
Dept. of Engineering and Computer Science
Concordia University
Montreal, Canada
krish27794@gmail.com

Jayaprakash Kumar
Dept. of Engineering and Computer Science
Concordia University
Montreal, Canada
jayaprakash6034@gmail.com

Darwin Anirudh Godavari
Dept. of Engineering and Computer Science
Concordia University
Montreal, Canada
darwinanirudh@gmail.com

***Abstract*— Software testing is broadly utilized technique to ensure quality of software system. Code coverage measures are commonly used to measure adequacy and evaluate and improve the existing test suites. McCabe Complexity is used to measure complexity of the code. Test Suite effectiveness also plays major role in software quality assurance. One common practice is to instrument faults, either manually or by using mutation operators. There are various tools which makes it easy to perform test suite effectiveness analysis. Software systems evolve over time due to changes in requirements, optimization of code, fixes for security and reliability bugs etc. We use Relative Code Churn as a measure of maintenance efforts. Software quality also plays major role in ensuring consistent and reliable experience for end user. The post-release defect density is used as a measure of software quality. We also collect other metrics such as cyclomatic complexity and lines of code, which are used to calculate the defect density. We analyze 3 large and 2 small open-source Java projects in order to find correlation between Coverage, McCabe Complexity, Relative Code Churn and Post Release Defect Density.**

Keywords—relative code churn, defect density, coverage, mccabe complexity, mutation testing, test suite effectiveness

I. INTRODUCTION

A Critical refinement between software engineering and other, more well-established branches of engineering is the shortage of well- accepted measures, or metrics, of software development. Without metrics, the undertakings of arranging and controlling programming advancement and maintenance will stay dormant in a "craft"- type mode. Of particular concern is the need to improve the software maintenance process, given that maintenance is estimated to consume 40-75% of the software effort [4]. What differentiates maintenance from other aspects of software engineering is, of course, the constraint of the existing system. It constrains the maintenance work in two ways-the first through the imposition of a general design structure that circumscribes the possible designs of the system modifications, and the second, more specifically, through the complexity of the existing code which must be modified.

Testing is broadly accepted technique to be accepted as foundation of software reliability in practice. The expanding size and multifaceted nature of programming has required enhancements in programming testing. Nevertheless, testing is expensive and time-consuming technique and thus, software developers and managers always address the question that how much testing is enough. A commonly accepted metric in code coverage. A set of tests is considered sufficient when executing the tests causes every line, branch or path, depending on the kind of coverage desired, to be executed at least once. Nevertheless, achieving adequate coverage does not prove that the code is correct. “Program testing can be used to show the presence of bugs, but never to show their absence!” - Edsger W. Dijkstra

A metric of this latter, more specific type of system influence is McCabe’s cyclomatic complexity metric, a measure of the maximum number of linearly independent circuits in a program control graph [1]. As described by McCabe, a primary purpose of the metric it to identify software modules that will be difficult to test or maintain” [1, p. 3081, and it is therefore of particular interest to researchers and practitioners concerned with maintenance. The McCabe metric has been widely used in research, and a recent article by Shepperd cites some 63 articles which are directly or indirectly related to the McCabe metric [3].

Both industrial software developers and software engineering researchers are interested in measuring test suite effectiveness. A well-established proxy measurement for test suite effectiveness in testing research is the mutation score, which measures a test suite’s ability to distinguish a program under test, the original version, from many small syntactic variations, called mutants. Specifically, the mutation score is the percentage of mutants that a test suite can distinguish from the original version. Mutants are created by systematically injecting small artificial faults into the program under test, using well-defined mutation operators. Examples of such mutation operators are replacing arithmetic or relational operators, modifying branch conditions, or deleting statements [5].

In this study, we examine 5 large open-source Java projects which are being supported for more than 10 years, that use the JIRA bug tracking service, that provides support for bug tracking and project management. We chose all the projects from apache organization to have identical process to replicate process and have more stronger result. We download

these 5 projects that are hosted on GitHub and use Maven. GitHub is one of the largest software repositories, which hosts millions of software projects.

Out experiment consist of 5 open-source Java projects from apache with each having 5 different version as below.

TABLE I. RESEARCH HYPOTHESYS

Project	Versions	SLOC(of latest version)
Apache Common Configurations	2.1, 2.1.1, 2.2, 2.3, 2.4	847k
Apache Common Configurations	2.1, 2.1.1, 2.2, 2.3, 2.4	847k
Apache Common IO	2.2, 2.3, 2.4, 2.5, 2.6	186k
Apache Common Lang	3.4, 3.5, 3.6, 3.7, 3.8.1	80k
Apache Common Math	3.3, 3.4, 3.5, 3.6, 3.6.1	35k

Our hypothesis between several software metrics are as below.

TABLE II. RESEARCH HYPOTHESYS

	Hypothesis
H_1	Higher Test Coverage might show better test suite effectiveness
H_2	Higher Mccabe Complexity will have lower test coverage(statement and branch)
H_3	Code coverage has an insignificant correlation to the number of bugs as well as to other metrics such as defect density.
H_4	We expect the larger the proportion of churned (added + changed) code to the LOC of the new binary, the larger the magnitude of the defect density

The rest of the paper is organized as follows. In the next section we present the measures that we use in our study. The Section 2 and Section 3 describes related work and terminology for the project respectively. Section 4 describes steps followed for analysis. Finally, in Section 5 results of the project is outlined.

II. RELATED WORK

Laura Inozemtseva and Reid Holmes [7] found that there is a low to moderate correlation between coverage and effectiveness when the number of test cases in the suite is controlled for. In addition, they also found that stronger forms of coverage do not provide greater insight into the effectiveness of the suite. Their results suggest that coverage, while useful for identifying under-tested parts of a program, should not be used as a quality target because it is not a good indicator of test suite effectiveness.

Past studies have analyzed the importance of testing on the overall quality of the software. Our work is closely related to the study conducted by Mockus et al. [8], where they investigate two industrial software projects from Microsoft and Avaya with the goal of understanding the impact of coverage on test effectiveness. They also calculate the amount of test effort required to achieve different coverage levels. Their results show that increasing test coverage reduces field problems but increases the amount of effort required for testing.

Ahmed.[9] analyze a large number of systems from GitHub and Apache and propose a novel evaluation of two commonly used measures of test suite quality: statement coverage and mutation score, i.e., the percentage of mutants killed . They compute test suite quality by correlating testedness of a program element (class, method, statement, or block) with the number of bug-fixes. They define testedness as how well a program element is tested, which can be measured using metrics such as coverage and mutation score. They find that statement coverage and mutation score have a weak negative correlation with bug-fixes. Cai and Lyu use coverage and mutation testing to analyze the relationship between code coverage and fault detection capability of test cases [11]. Cai performs an empirical investigation to study the fault detection capability of code coverage and finds that code coverage is a moderate indicator of fault detection when used for all the test set [10].

Nachiappan Nagappan and Thomas Ball performed case study on Windows Server 2003 indicates the validity of the relative code churn measures as early indicators of system defect density. Furthermore, their code churn metric suite is able to discriminate between fault and not fault- prone binaries with an accuracy of 89.0 percent [6].

Munson et al. [17] observe that as a system is developed, the relative complexity of each program module that has been altered (or churned) also will change. The rate of change in relative complexity serves as a good index of the rate of fault injection. They studied a 300 KLOC (thousand lines of code) embedded real time system with 3700 modules programmed in C. Code churn metrics were found to be among the most highly correlated with problem reports [17].

Ostrand et al. [20] use information of file status such as new, changed, unchanged files along with other explanatory variables such as lines of code, age, and prior faults etc. as predictors in a negative binomial regression equation to predict the number of faults in a multiple release software system. The predictions made using binomial regression model were of a high accuracy for faults found in both early and later stages of development. [20]

Closely related to our study is the work performed by Graves et al. [9] on predicting fault incidences using software change history. Several statistical models were built based on a weighted time damp model using the sum of contributions from all changes to a module in its history. The most successful model computes the fault potential by summing contributions from changes to the module, where large and/or recent changes contribute the most to fault potential [9]. This is similar to our approach of using relative measures to predict fault potential.

III. PRELIMINARIES

A. GitHub

GitHub is one of the largest project-hosting platforms and uses the git 11 version control system. GitHub is similar to a social network, where software developers spread across the globe can collaborate. Currently, GitHub has more than 11 million users and over 28 million repositories. We clone the repositories of software projects using the command git clone {url}. We only download projects that contain a Maven pom.xml file.

B. JIRA

JIRA is a project tracker used for issue tracking, bug tracking, and efficient project management. To be able to uniformly obtain bug information for the different projects in

our dataset, we focus on projects that use JIRA for reporting bugs. For each bug, JIRA records the affected and fixed version of the software, which represent the version in which the bug was found and the version in which the bug was fixed or resolved, respectively. This information ensures that we are collecting only post release bugs, i.e., those bugs logged after the release of the particular version of the software. We collect information about all the closed and resolved bugs for a particular affected version of the software. JIRA also assigns each bug an identifier that is unique for the given software project. When developers mention this identifier in the logs of the commits that fix the bug, we are able to track the files that were changed to solve the problem.

C. Maven

Maven 4 is a software project management tool that supports building and running the software and its test cases. Maven uses information that is present in the project object model (POM) file, pom.xml. The POM file contains information about the project such as its dependencies on libraries and the order in which the different components of the project should be built. Maven primarily supports Java projects and for such projects it dynamically downloads all dependencies from a central Maven repository. It is also designed around the "build portability" theme, so that you don't get issues as having the same code with the same build script working on one computer but not on another one (this is a known issue, we have VMs of Windows 98 machines since we couldn't get some of our Delphi applications compiling anywhere else). Because of this, it is also the best way to work on a project between people who use different IDEs since IDE-generated Ant scripts are hard to import into other IDEs, but all IDEs nowadays understand and support Maven.

IV. METHODOLOGY

To test the hypothesis, we followed general procedure outlined as below.

1. Adding dependency in pom.xml
2. Running mvn install command for jacoco
3. Running goal for PIT Testing
4. Calculating Spearman Correlation using CSVs generated by above listed tools
5. Calculating defect density from JIRA
6. Defining correlation between defect density and other metrics generated by above listed tools

A. Terminology

1) Code Coverage: Software testing is used to test different functionalities of a program or system and to ensure that given a set of inputs the system produces the expected results. A test adequacy criterion defines the properties that must be satisfied for a thorough test . Code coverage, which measures the percentage of code executed by test cases, is often used as a proxy for test adequacy. The percentage of code executed by test cases can be measured according to various criteria, including the percentage of executed source code lines (line coverage - Metric 1), and the percentage of executed branches (branch coverage Metric 2).

2) Mccabe Complexity (Metric 4): Cyclomatic complexity measures the number of linearly independent paths in the source code of a software application [29]. This measure increases by 1 whenever a new method is called or when a new decision point is encountered, such as an if, while, for, &&, case, etc. Cyclomatic complexity is often useful in knowing the number of test cases that might be required for independent path testing and a file or project with low complexity is usually easier to comprehend and test.

3) Mutation Score (Metric 3):

A mutant is a new version of a program that is created by making a small syntactic change to the original program. For example, a mutant could be created by modifying a constant, negating a branch condition, or removing a method call. The resulting mutant may produce the same output as the original program, in which case it is called an equivalent mutant. For example, if the equality test in the code snippet in Figure 1 were changed to if (index >= 10), the new program would be an equivalent mutant.

Mutation testing tools such as PIT generate a large number of mutants and run the program's test suite on each one. If the test suite fails when it is run on a given mutant, we say that the suite kills that mutant. A test suite's mutant coverage is then the fraction of non-equivalent mutants that it kills. Equivalent mutants are excluded because they cannot, by definition, be detected by a unit test.

If a mutant is not killed by a test suite, manual inspection is required to determine if it is equivalent or if it was simply missed by the suite. This is a time-consuming and error-prone process, so studies that compare subsets of a test suite to the master suite often use a different approach: they assume that any mutant that cannot be detected by the master suite is equivalent. While this technique tends to overestimate the number of equivalent mutants, it is commonly applied because it allows the study of much larger programs.

```
int index = 0;
while (true) {
   index++;
   if (index == 10) {
      break;
    }
}
```

The mutation score is defined as the percentage of killed mutants with the total number of mutants. Test cases are mutation adequate if the score is 100%. Test cases are mutation adequate if the score is 100%.

4) Relative Churned Code (Metric 5): Code churn is a measure of the amount of code change taking place within a software unit over time. It is easily extracted from a system's change history, as recorded automatically by a version control system. Most version control systems use a file comparison utility (such as diff) to automatically estimate how many lines were added, deleted and changed by a programmer to create a new version of a file from an old version. These differences are the basis of churn measures.

It can be understood better by considering an example. For ex. From version 4.1 of commons-collection to 4.2, 4997 lines were added and modified. These can be divided by LOC of version 4.2, 66695, which is equals to 0.074.

5) Defect Density (Metric 6): Defect Density is the number of defects confirmed in software/module during a specific period of operation or development divided by the size of the software/module. It enables one to decide if a piece

of software is ready to be released.Defect density is counted per thousand lines of code also know as KLOC.

Defect density is computed using historical data of the subject projects from the respective issue tracking systems. Our open source subject projects are available on GitHub and having issue tracking system as JIRA, which allows us to process the status and the history of issue reports. Defect density is not directly dependent on failing tests, but on problems found and reported by project collaborators and users. Although we did not verify each reported defect individually, they are usually reported for the released versions of the systems, meaning that the defects were not found during testing. Hence we assume that higher defect density rate is an indicator of unsuccessful testing and lower quality of the test suite's the respective issue tracking systems.

To calculate defect density for particular project, we searched using applying following advance filter in JIRA issue tracking site of the particular website.

project = COLLECTIONS AND affectedVersion = 4.2 and type = Bug

B. Subject Programs

We selected 5 open source java projects from variety of application domain. The Apache common collection is Framework was a major addition in JDK 1.2. It added many powerful data structures that accelerate development of most significant Java applications. Since that time it has become the recognized standard for collection handling in Java.

The Commons Configuration software library provides a generic configuration interface which enables a Java application to read configuration data from a variety of sources. Commons Configuration provides typed access to single, and multi-valued configuration parameter. Commons IO is a library of utilities to assist with developing IO functionality. The standard Java libraries fail to provide enough methods for manipulation of its core classes. Apache Commons Lang provides these extra methods. Commons Math is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language or Commons Lang.

C. Calculating Statement, Branch and Mccabe Complexity

We used the open source tool Jacoco [12] to measure two types of coverage: statement and decision coverage and Mccabe Complexity. Adding Jacoc in Project and Runnig was a simple process. It requires to add dependency in pom.xml as below.

```
<plugin>
  <groupId> org.jacoco </groupId>
  <artifactId> jacoco-maven-plugin </artifactId>
  <version> 0.7.7.201606060606</version>
  <executions>
    <execution>
      <goals>
        <goal> prepare-agent </goal>
      </goals>
    </execution>
    <execution>
      <id> report </id>
      <phase> prepare-package </phase>
      <goals>
        <goal> prepare-agent </goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

and executing it via following command, will generate HTML and CSV reports in target/site folder.

```
mvn clean install
```

The generated html and csv reports can be found in directory target/site.

D. Calculating Mutation Score

We used the open source tool PIT [13] to generate faulty versions of our programs. Using PIT was similar to Jacoco, required to add dependency in pom.xml. In addition to that, here we run mutation testing on entire project than some classes.

```
<plugin>
  <groupId> org.pitest </groupId>
  <artifactId> pitest-maven </artifactId>
  <version> LATEST </version>
  <configuration>
    <outputFormats>
      <param> HTML </param>
      <param> CSV </param>
    </outputFormats>
  </configuration>
</plugin>
```

We run this command to run PIT testing on entire project.

mvn org.pitest:pitest-maven:mutationCoverage -X To calculate mutation score for each class from csv, we use below formula. PIT in its HTML reports also counts as per below formula.

$$\frac{KILLED + TIMED-OUT + MEMORY-ERROR}{TOTAL\ MUTANTS}$$

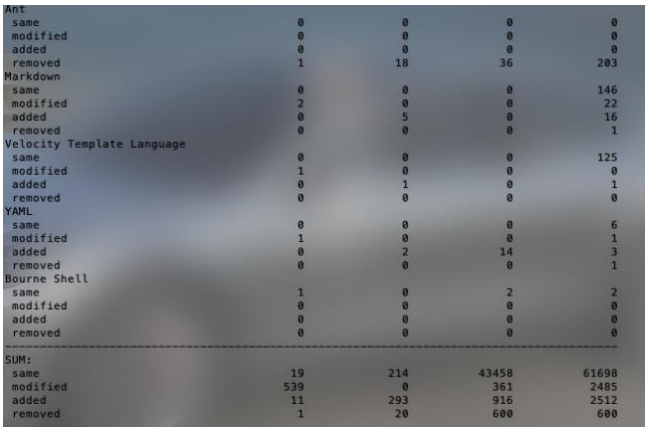
E. Measuring Relative Churned Code

We used CLOC [15] tool to calculate the number of lines which are added and modified to current version with respect to recent previous version. CLOC can be installed through number of package managers such as npm(node package manager), choco(windows package manager), yum or apt(linux package manager) and many more. It is platform independent tool and uses very famous Diff Algorithm[16]. There isn't one true diff algorithm, but several with different characteristics. The basic idea is to find a 'modification script' that will turn Text A into Text B. They use modification operations such as insertion and deletion. Usually, you'd want the minimal number of changes required, but some application also have to weigh operations depending on how much text each one affects, or other factors.

CLOC is command line tool which can be used to measure lines of code as well as different between two files or directories.

We executed below command which showed us output as per below screenshot. We considered only important change in lines of code such as Java, XML and ignoring HTML.

Cloc –diff file1.zip file2.zip



Ant				
same	0	0	0	0
modified	0	0	0	0
added	0	0	0	0
removed	1	18	36	203
Markdown				
same	0	0	0	146
modified	2	0	0	22
added	0	5	0	16
removed	0	0	0	1
Velocity Template Language				
same	0	0	0	125
modified	1	0	0	0
added	0	1	0	1
removed	0	0	0	0
YAML				
same	0	0	0	6
modified	1	0	0	1
added	0	2	14	3
removed	0	0	0	1
Bourne Shell				
same	1	0	2	2
modified	0	0	0	0
added	0	0	0	0
removed	0	0	0	0
SUM:				
same	19	214	43458	61698
modified	539	0	361	2485
added	11	293	916	2512
removed	1	20	600	600

Fig. 1. CLOC Line Difference

CLOC can also be used to measure SLOC for current version which works as a relative measure to this metric.

F. *Measuring Defect Density*

First, we search JIRA issue tracking system for particular project and allow public access to all of the issues filed in the tracking system. Proper and latest issue tracker link for the particular project can be found on project site. Then to count number of bugs raised for particular project, we apply following filter in its advance search.

For each bug, JIRA records the affected version of the software. We collected all of the closed and resolved bugs for the checked out version of the software. We perform this step manually for each software project, as each project has a unique name used by JIRA and each project has a different checked out version. We obtained the JIRA name of each project by searching the project's website. For example, the project commons-collections in our dataset, for which we use version 4.2, has JIRA name COLLECTIONS.

Then by using advanced search filter, we find number of bugs, and record it for 5 different version of each project.

G. *Spearman's Rank Correlation*

Spearman's rank correlation coefficient (p) is a nonparametric test that is used to measure the strength of monotonic relationship between sets of data [34].The value of rho ranges from −1, which signifies a perfect negative correlation, to +1, which signifies a perfect positive correlation. The value 0 shows that there is no correlation between the variables. To calculate Spearman's rho, the raw values from the datasets are arranged in ascending order and each value is assigned a rank equal to its position in the list. The values that are identical in two sets are given a rank equal to the average of their positions.

Spearman rank correlation is a commonly-used robust correlation technique [6] because it can be applied even when the association between elements is non-linear.

The Equation shown below is the formula for calculation:

$$r_s = 1 - \frac{6\sum D^2}{n\left(n^2 - 1\right)}$$

Where, d = the distance between the two variables in ranking n = number of variables. Spearman correlation coefficient,

$0 \le r_s < 0.1$ = None, $0.1 \le r_s < 0.3$ = Small, $0.3 \le r_s < 0.5$ = Moderate, $0.5 \le r_s < 0.7$ = High, $0.7 \le r_s < 0.9$ = Very High, $0.9 \le r_s \le 1.0$ = Perfect.

H. *P Value*

The p-value is the probability of obtaining a result equal to or more extreme than what was actually observed, when the null hypothesis (H0) of a study question is true. The significance level (α) refers to a preselected value of probability. If p-value is less than the significance level (α), then we can reject the null hypothesis, i.e.,our sample gives reasonable evidence to support the alternative hypothesis(H1). In this study, we select the value of α as 5% or 0.05 and if the p-value is less than 0.05, we reject the null hypothesis.

I. *Data Analysis Procedure*

All the statistical analysis was performed using Python and Pandas, which is a programming language and data analysis framework for statistical computing that is widely used in academia and industry. To compute Spearman's ρ, we use the equation, dataframe.corr(x,y, method ="spearman"), where corr() is provided by the pandas package in Python, and x and y are numeric vectors of data values of the same length.

V. RESULTS

In this section, we perform analysis and present the results.

A. *Correlation between Coverage and McCabe Complexity*

Code coverage gives us an idea of the thoroughness of testing by providing information about the amount of code that is tested. Increasing coverage, however, requires more work in terms of test case development, and may also increase the test suite running time.

We assume that class that has methods with higher McCabe complexity will have lower test coverage in terms of Statement Coverage and Branch Coverage. We performed this analysis at class level for all projects with each having 5 different version. We got spearman rho as below.

We calculated this correlation for 5 different version of each project. We found it overall negative for both statement coverage and branch coverage to McCabe complexity. We used csv report generated by Jacoco as our data source to perform correlation analysis at class level. Jacoco by default does not provide statement coverage, branch coverage and McCabe complexity in its html report. We used below formula for calculating it.

$$STATEMENTCoverage = \frac{LINECOVERED}{LINECOVERED + LINEMISSED} * 100$$

$$BRANCHCoverage = \frac{BRANCHCOVERED}{BRANCHCOVERED + BRANCHMISSED} * 100$$

$$Mccabe - Complexity = Complexity - Coverd + Complexity - Missed$$

Here Version N is for our convenience, proper version can be seen from Introduction part. Below is a correlation strength between Coverage and McCabe Complexity.

TABLE III. CORRELATION BETWEEN STATEMENT COVERAGE AND MCCABE COMPLEXITY

Project Name	V1	V2	V3	V4	V5
Commons-Collections	-0.17	-0.18	-0.22	-0.26	-0.27
Commons-Configuration	-0.17	-0.48	-0.41	-0.46	-0.43
Commons-IO	-0.33	-0.33	-0.32	-0.33	-0.34
Commons-Lang	-0.17	-0.17	-0.17	-0.18	-0.18
Commons-Math	-0.17	-0.18	-0.22	-0.26	-0.27

As Observed in Table 3, For 3 projects out of 5, there is a small negative correlation between observed entities. While for others, there is a moderate correlation.

Statement Coverage has small negative correlation with McCabe Complexity.

As Observed in Table 4, out of 25 values, 2 correlation were insignificant (None), while others being moderate or small.

TABLE IV. CORRELATION BETWEEN BRANCH COVERAGE AND MCCABE COMPLEXITY

Project Name	V1	V2	V3	V4	V5
Collection	-0.10	-0.09	-0.20	-0.19	-0.18
Configuration	-0.10	-0.35	-0.35	-0.35	-0.29
IO	-0.18	-0.19	-0.20	-0.26	-0.32
Lang	-0.08	-0.17	-0.15	-0.15	-0.14
Math	-0.10	-0.09	-0.20	-0.19	-0.18

Branch Coverage has small negative correlation with McCabe Complexity.

We performed this analysis at class level for all projects. Below is the sample plot for collection v4.2 as an example.

Average p value for both observation was about 1.261685526769464e-06, which indicate that hypothesis can't be ignored.

Below is the sample of graph plot showing the correlation for commons-collection v4.2.

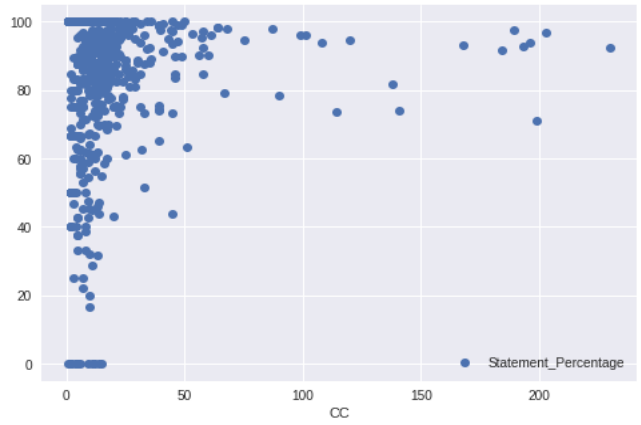


Fig. 2. Correlation between Statement Coverage and Mccabe Complexity

B. *Correlation between Coverage and Mutation Score*

Code Coverage measures how much code is covered through testing. While mutation measures how effective test suites are to detect fault. Our hypothesis is that Code Coverage and Mutation Score have positive correlation. The class with higher code coverage will have high mutation score, hence better test suite effectiveness.

We performed this correlation analysis for each project at class level. We chose to run Mutation Testing on all of the classes of the particular project. It took less than 1 hour for 4 projects and 9 hours for commons math. The correlation between defined two entities were calculated by combining two csv files generated by Jacoco and PIT Test respectively. Before that we also needed to clean and format data generated by PIT Test. This cleansing and grouping of data was done using split and fill functions of pandas framework and then grouping them using group by clause. They were combined using class name and package name as common key, which gives a table with each row having coverage and mutation score in single data unit.

TABLE V. CORRELATION BETWEEN STATEMENT COVERAGE AND MUTATION SCORE & BRANCH COVERAGE AND MUTATION SCORE

Project Name	Statement Coverage and Mutation Score	Branch Coverage and Mutation Score
Commons-Collections	0.44	0.51
Commons-Configuration	0.64	0.73
Commons-IO	0.57	0.32
Commons-Lang	0.50	0.59`
Commons-Math	0.57	0.44

Each project consist classes with range of 250 to 700. Average p value for both observation was about 5.0690252087233975e-06, which indicate that hypothesis cannot be ignored.

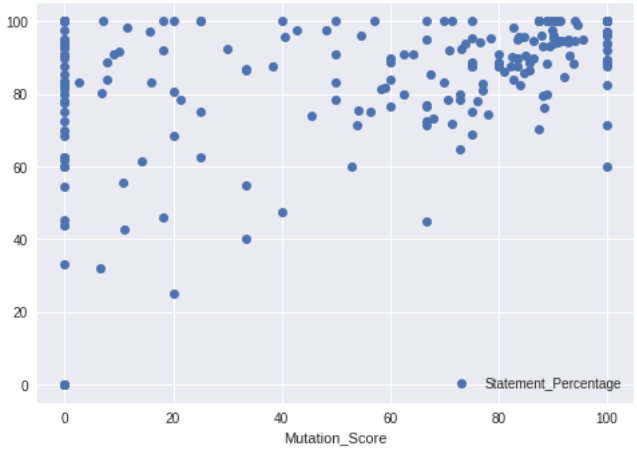


Fig. 3. Correlation between Statement Coverage and Mutation Score

From the Table V, we can observe that 4 out of 5 projects has High Correlation between Statement Coverage and Mutation Score and remaining 1 project has Moderate Correlation. Also, 3 out of 5 projects has high correlation, and the other 2 has moderate correlation.

Statement Coverage has high positive correlation with Mutation Score.

Branch Coverage has high positive correlation with Mutation Score.

C. Correlation between Coverage and Defect Density

We compared Statement and Branch Coverage with defect density of each project at project level. We found very different result for each project. Correlation between statement coverage and defect density varies from -0.9 to 0.1. Similarly, Correlation between branch coverage and defect density varies from -0.8 to -0.05.

Here we calculated coverage of project by calculating covered lines of code (by test cases) divided by total lines of code. Similarly for branch coverage as covered branches divided by total number of possible branches. This overall project level coverages were calculated using csv report generated by Jacoco.

TABLE VI. CORRELATION BETWEEN STATEMENT COVERAGE AND MUTATION SCORE & BRANCH COVERAGE AND MUTATION SCORE

Project Name	Statement Coverage and Defect Density	Branch Coverage and Defect Density
Commons-Collections	-0.5	-0.5
Commons-Configuration	-0.9	-0.3
Commons-IO	0.1	0.05
Commons-Lang	-0.8	-0.8
Commons-Math	-0.3	-0.05

As observed in Table VI, Correlation Rank varies from positive to negative and p value for the observation is also near to 1 for IO and Math, so we can put aside them while defining overall correlation, so it indicate that there is very high negative correlation between statement coverage and defect density with confidence level of 0.05. Similarly, there is moderate negative correlation between branch coverage and defect density.

Statement Coverage has very high negative correlation with Defect Density.

Branch Coverage has moderate negative correlation with Defect Density.

D. Correlation between Relative Churned Code and Defect Density

TABLE VII. CORRELATION BETWEEN RELATIVE CHURNED CODE & DEFECT DENSITY

Project Name	Spearman Rank
Collection	0.3
Configuration	-0.6
IO	0
Lang	0.5
Math	-0.1

We calculated Relative Code Churn and Defect Density for 5 version of each project. The defect density was calculated at project level by calculating number bugs raised in project's issue tracking repository. The Spearman Correlation Rank varies from negative to positive for each project. It varies from project to project. We found it perfectly distributed between positive and negative range of -1 to 1.

Relative Churned Code has no correlation with Defect Density.

As Observed in Table VII, it gives value from -0.3 to 0.6. This is significant variance in values among different projects. The result was completely opposite of our hypothesis which was strong correlation between relative code churn and defect density.

VI. THREATS TO VALIDITY

A. External Validity

These threats relate to the generalizability of the results. In this study, we have investigated 5 large and popular open source Java projects from GitHub. GitHub is one of the largest repositories and hosts millions of projects of different sizes and from various domains. We have tried to ensure that our dataset consists of projects of substantial size (SLOC >= 30K).

Finally, while our two subjects were considerably smaller than the programs used in previous studies, and they are still not large by industrial standards. Additionally, all of the projects were open source, so our results may not generalize to closed source systems.

Also all our projects were published during year 2002 to 2004, making it 15 to 17 years of history. It would be better to consider having software of different time history. Taking software which are newly released in addition to one taken in study.

B. Internal Validity

These threats are related to the environment under which experiments were carried out. We used the open source tool Jacoco [12] to measure statement , decision coverage and Mccabe Complexity. In this study, we only consider projects that use Maven, i.e., they contain a pom.xml file. It is possible that projects that do not entirely follow Maven's structure may be interpreted wrongly. This could prompt Maven wrongly computing certain measurements of LOC or miss test cases in the project, which can affect the coverage value. We have physically checked a couple of activities and they completely fit in with the Maven directory structure. While counting the delta (number of times a file is changed),we utilize a major version past to the current checked out version since it is hard to locate the accurate past version in the repository. So, we may have wrongly distinguished the occasions the files have changed.

C. Construct Validity

In our study we measured the Code, Branch coverage, McCabe Complexity, Mutation Score, Relative Churned Code and Defect Density. Code and branch coverage are straight forward to measure, but effectiveness is more nebulous, as we are attempting to measure effectiveness of test suite that has never been used in practice. In the absence of equivalent mutants, this metric has high construct validity. Unfortunately, our treatment of equivalent mutants

introduces a threat to the validity of this measurement. Recall that we assumed that any mutant that could not be detected by the program's entire test suite is equivalent. In theory, the mutants are a random subset of the entire set of mutants, so ignoring them should not affect our results. However, this may not be true. For example, if the developers frequently test for off-by-one errors, mutants that simulate this error will be detected more often and will be less likely to be classified as equivalent.

VII. FUTURE WORK

Our future plan is to analyze datasets from other open-source platforms to mitigate the external and internal validity threats. Furthermore, we plan to collect a larger dataset of projects having significant representation across low, medium, and high coverage levels and having variety of version history and lifespan of project to investigate the impact of different coverage levels, Relative Code Churn and mutation score impacts on the number of post-release bugs.

VIII. CONCLUSION

In this study, We present correlation between Code Coverage, McCabe Complexity, Test Suite Effectiveness, Defect Density and Relative Code Churn. The findings of our studies are as follows :

1. Coverage has small negative correlation with McCabe Complexity.
2. Coverage has high positive correlation with Mutation Score(Test Suite Effectiveness.)
3. Coverage has moderate to high negative correlation with defect density.
4. Relative Code Churn has no correlation with defect density. It varies between projects. It might be possible that addition or modification of CLOC might be because of bug fix in some project and for some it might be feature addition. So, we couldn't find any correlation between this two entities. Our future work highlight this issue and focus on choosing projects and version based on this kind of filters.

ACKNOWLEDGEMENT

The authors would like to acknowledge their professor, Prof. Jinqiu Yang, POD Mehdi Nejadgholi and Zishuo Ding for the continuous useful feedback.

DATASET

Our dataset, scripts, steps to reproduce study other related things used to find results of the study is publicly available on https://github.com/niravjdn/Software-Measurement-Project/

REFERENCES

[1] Edsger W. Dijkstra - Wikiquote. (2019). En.wikiquote.org. Retrieved 7 April 2019, from https://en.wikiquote.org/wiki/Edsger_W._Dijkstra

[2] T. J. McCabe, "A complexity measure," IEEE Trans. Software Eng., vol. SE-2, pp. 308-320, 1976.

[3] M. Shepperd, "A critique of cyclomatic complexity as a software metric," Software Eng. J., vol. 3, no. 2, pp. 30-36, Mar. 1988.

[4] I. Vessey and R. Weber, "Some factors affecting program repair maintenance: an empirical study," Commun. ACM, vol. 26, no. 2, pp. 128-134, Feb. 1983.

[5] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering (TSE), 37(5):649–678, 2011.

[6] Nachiappan Nagappan and Thomas Ball Code churn: a measure for estimating the impact of code change - IEEE Conference Publication. (2019). Ieeexplore.ieee.org, from https://ieeexplore.ieee.org/document/738486

[7] Inozemtseva, L., & Holmes, R. (2014). Coverage is not strongly correlated with test suite effectiveness. Proceedings Of The 36Th International Conference On Software Engineering - ICSE 2014. doi:10.1145/2568225.2568271

[8] A. Mockus, N. Nagappan, and T. Dinh-Trong, "Test coverage and post- verification defects: A multiple case study," in Proc. 3rd Int. Symp. Em- pirical Softw. Eng. Meas., 2009, pp. 291–301.

[9] I. Ahmed, R. Gopinath, C. Brindescu, A. Groce, and C. Jensen, "Can testedness be effectively measured," in Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng., 2016, pp. 547–558.

[10] X. Cai, "Coverage-based testing strategies and reliability modeling for fault-tolerant software systems," Ph.D. dissertation, Chinese Univ. Hong Kong, Hong Kong, 2006.

[11] X. Cai and M. R. Lyu, "The effect of code coverage on fault detection under different testing profiles," SIGSOFT Softw. Eng. Notes, vol. 30, no. 4, pp. 1–7, 2005.

[12] EclEmma - JaCoCo Java Code Coverage Library. (2019). Eclemma.org. Retrieved 7 April 2019, from https://www.eclemma.org/jacoco/

[13] PIT Mutation Testing. (2019). Pitest.org. Retrieved 7 April 2019, from http://pitest.org/

[14] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In Proc. of the Int'l Conf. on Soft. Eng., 2005.

[15] CLOC -- Count Lines of Code. (2019). Cloc.sourceforge.net. Retrieved 7 April 2019, from http://cloc.sourceforge.net/

[16] McQueen, T. (2006). *Algorithm::Diff - Compute `intelligent' differences between two files / lists - metacpan.org*. *Metacpan.org*. Retrieved 16April2019,from https://metacpan.org/pod/release/TYEMQ/Algorithm-Diff-1.1902/lib/Algorithm/Diff.pm