Analysis of Software Measurement Metrics on Open Source Projects

Project Report
Software Measurement (SOEN 6611)
Professor Jinqiu Yang

Team G

Student	Student ID	Email
Sareh Farid	27845782	sarehfarid@gmail.com
Seyed Hamed Valiollahi Bayeki	40057506	hamedv90@gmail.com
Ahmad Memari (Arash)	40088010	memari.ahmad@gmail.com
Sharareh Keshavarzi	40087339	keshavarzis02@gmail.com
Hanieh QasemiBoroujeni	40057756	hanie21_ghasemi@yahoo.com

Repository: https://github.com/sarehfar/Measurement-Project

UNIVERSITÉ Concordia UNIVERSITY

Analysis of Software Measurement Metrics on Open Source Projects

Software Measurement Project Report

Sareh Farid
Dept of Software Engineering and Computer Science
Concordia University
Montreal, Canada
sarehfarid@gmail.com

Seyed Hamed Valiollahi Bayeki
Dept of Software Engineering and Computer Science
Concordia University
Montreal, Canada
hamedv90@gmail.com

Ahmad Memari (Arash)
Dept of Software Engineering and Computer Science
Concordia University
Montreal, Canada
memari.ahmad@gmail.com

Sharareh Keshavarzi
Dept of Software Engineering and Computer Science
Concordia University
Montreal, Canada
keshavarzis02@gmail.com

Hanieh QasemiBoroujeni
Dept of Software Engineering and Computer Science
Concordia University
Montreal, Canada
hanie21_ghasemi@yahoo.com

***Abstract*—A software metric is a measure that allows getting a quantitative value of software features or specifications. The objective of measuring the software quality is to use the received results for planning the budget, schedule, estimating costs, testing and QA, debugging, etc. Still, the main goal is to measure quality. As we all know, many infrastructure problems are directly related to code quality or undetected risks, which increase IT costs and additional maintenance efforts. In this case of study, several forms of analysis and metrics have been used to assess four open-source java projects for Statement coverage, Branch Coverage, Mutation Score, McCabe Complexity, Relative Code Churn and Post Release Defect Density and investigated the correlation among them.**

Keywords—Statement coverage, Branch coverage, McCabe complexity, Mutation score, Code churn, Post release defect density, Test suite

I. INTRODUCTION

The goal of tracking and analyzing software metrics is to determine the quality of the current product or process, improve that quality and predict the quality once the software development project is complete. Without metrics, it would be almost impossible to quantify, explain, or demonstrate software quality. Furthermore, gathering development metrics will help us find ways to minimize errors in software and make operations more efficient. In this study, four open-source software, two of them more than 100k LOC, have been measured and analyzed by the below metrics: Statement Coverage,Branch Coverage, Mutation Score, McCabe Complexity, Relative Code Churn and Post Release Defect Density, besides, the correlation among them has been found

II. SELECTED PROJECTS

A. Apache Commons Codec

Apache Commons Codec (TM) software provides implementations of common encoders and decoders such as Base64, Hex, Phonetic and URLs. Codec was formed as an attempt to focus development effort on one definitive implementation of the Base64 encoder. While this package contains an abstract framework for the creation of encoders and decoders, Codec itself is primarily focused on providing functional utilities for working with common encodings [1].

Size: 51638 LOC
Version Number: 1.14, 1.13, 1.12, 1.11, 1.10
Language Written: JAVA
Project link: https://github.com/apache/commons-codec
Issue-tracking System: Jira
Version-control System: Git

B. Apache Commons Collections

The Java Collections 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 [2].

Size: 138143 LOC
Version Number: 4.4, 4.3, 4.2, 4.1, 4.0
Language Written: JAVA
Project link: https://github.com/apache/commons-collections
Issue-tracking System: Jira
Version-control System: Git

C. Apache Commons Configuration

The Commons Configuration software library provides a generic configuration interface which enables a Java application to read configuration data from a variety of sources. Configuration objects are created using configuration builders. Different configuration sources can be mixed using a CombinedConfigurationBuilder and a CombinedConfiguration. Additional sources of configuration parameters can be created by using custom configuration objects. This customization can be achieved by extending AbstractConfiguration or AbstractHierarchicalConfiguration [3].

Size: 150507 LOC
Version Number: 2.7, 2.6, 2.5, 2.4, 2.3
Language Written: JAVA
Project link: https://github.com/apache/commons-configuration
Issue-tracking System: Jira
Version-control System: Git

D. Apache Maven Doxia

Doxia is a content generation framework that provides powerful techniques for generating static and dynamic content, supporting a variety of markup languages. The actual component is the base component from the whole Doxia suite, with the core parser and sink APIs and their implementation in supported markup languages. It is used by the Doxia Sitetools extension, that adds site and documents support [4].

Size: 95732 LOC
Version Number: 1.9.1, 1.9, 1.8, 1.7, 1.6
Language Written: JAVA
Project link:https://github.com/apache/maven-doxia
Issue-tracking System: Jira
Version-control System: Git

III. METRICS

A. Metric 1: Statement Coverage (coverage metric)

Statement and branch coverage are both test coverage metrics. The statement coverage metric is also called: line coverage or segment coverage. Statement coverage tracks the executed program statements. The metric value of a program is the number of executed statements in it, divided by the total number of statements. It covers only the true conditions.

A = Number of statements executed
B = Total number of statements in source code
Statement Coverage = (A/B) * 100

B. Metric 2: Branch Coverage (coverage metric)

The branch coverage metric is also called: Decision coverage. It counts the percentage of decision branches being executed by tests. This metric tries to ensure that each one of the possible branches from each decision point is executed at least once and thereby ensuring that all reachable code is executed. Branch Coverage covers both the true and false conditions, unlike the statement coverage.

A = Number of Decision Statements Executed
B = Total Number of Decision Outcomes
Branch Coverage = (A/B) * 100

C. Metric 3: Mutation Score (Test-suite Effectiveness metric)

This metric measures the ability of the test suite to remove mutants, which used to measure the effectiveness of the suite. Having an efficient test suite is important because it uses test collection cases in the program. The main purpose of this metric is to find real faults, so a high mutation score can find more real faults.

Mutation Score = (Number of mutants killed/Total number of mutants) * 100

D. Metric 4: McCabe Complexity (Complexity metric)

In order to measure the complexity of the program, we use the McCabe Cycloramic complexity metric. This metric measures the complexity of the program by measuring the number of linearly independent paths that can be executed. It is used to determine the number of test cases, which is very helpful in achieving complete branch coverage.

MC = E - N + 2P
E = Number of edges in the graph
N = Number of Nodes in the graph
P = Number of connected components.

E. Metric 5: Code Churn (Software maintenance metric)

We use Relative Code Churn as a metric for maintenance metrics. This metric measures the amount of code changes over a time period. It uses version control systems to detect changes in system history, which uses a comparison technique to estimate the differences between new and old versions of the file. This method finds the number of lines added, changed or deleted over a period of time.

Relative code churn = Number of deleted lines + number of added lines

F. Metric 6: Post-release defect density (Software Quality metric)

Post-release defect density is the metric we use to measure quality metrics. It is calculated by dividing the number of known defects by the size of the software. We are going to use post-release defect density as the measurement criteria, and collect our data from open-source software. This measurement uses the issue tracker to measure the number of defects found per 1000 lines of source code after releasing the software. Increasing post-release defect density shows more faults, so the quality of code declines. This metric could be used as a quality inspector; the zero score of this metric indicates a very high-quality program and a high score post-release defect density shows low-quality software.

Defect density = Number of known defects / Size of the Software(KSLOC)

Number of known defects is the total number of post-release defects

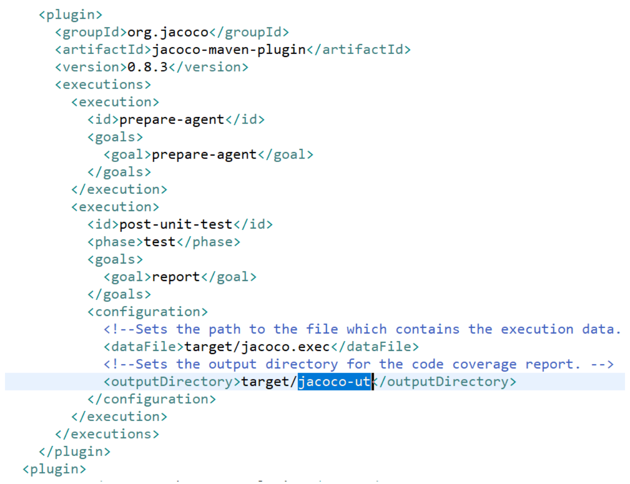
Size of the Software (KSLOC) is the total number of source lines of code.

IV. DATA COLLECTION

A. Metric 1: Statement Coverage

Statement coverage can be calculated using Jacoco. JaCoCo is a free code coverage library for Java, which has been created by the EclEmma team.

In order to run with JaCoCo, we need to declare this maven plugin in our pom.xml file. After that, we need to open a terminal or command prompt and direct to the root directory of the project and execute the maven build command. Then, the results will be stored in target/jacoco-ut directory of the project. There is an index.html showing the results in html format.



```
  <plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.3</version>
    <executions>
      <execution>
        <id>prepare-agent</id>
        <goals>
          <goal>prepare-agent</goal>
        </goals>
      </execution>
      <execution>
        <id>post-unit-test</id>
        <phase>test</phase>
        <goals>
          <goal>report</goal>
        </goals>
        <configuration>
          <!--Sets the path to the file which contains the execution data.
          <dataFile>target/jacoco.exec</dataFile>
          <!--Sets the output directory for the code coverage report. -->
          <outputDirectory>target/jacoco-ut</outputDirectory>
        </configuration>
      </execution>
    </executions>
  </plugin>
<plugin>
```

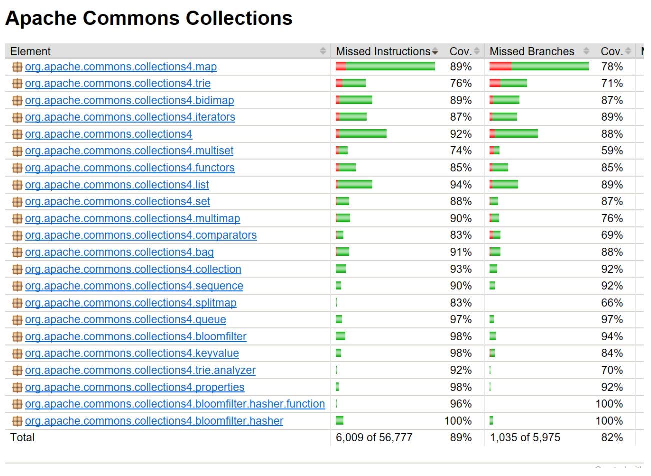
Figure 1: JaCoCo plugin in pom java

The statement coverage calculated using the following formula.

A = Total number of lines
B = Total number of missed lines
Statement Coverage = A / (A+B) * 100

B. Metric 2: Branch Coverage

In order to calculate the branch coverage we need to take the similar steps as for the statement coverage collection described in the previous section. We produce branch coverage with Jacoco report, which gives us both csv and html report. The branch coverage is calculated and can be found in target/jacoco-ut/index.html file. Like statement coverage, this coverage is also class based.



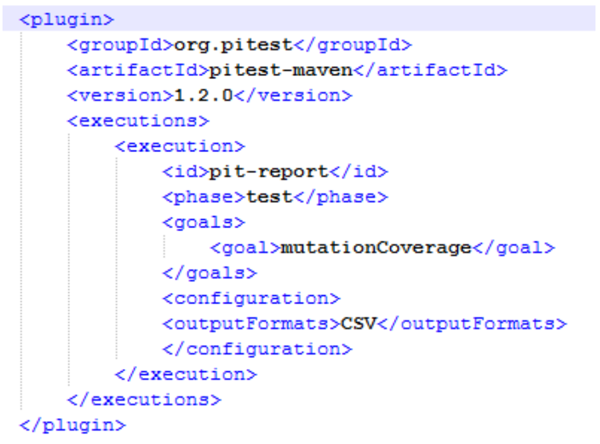
Apache Commons Collections

Element	Missed Instructions	Cov.	Missed Branches	Cov.
org.apache.commons.collections4.map		89%		78%
org.apache.commons.collections4.trie		76%		71%
org.apache.commons.collections4.bidimap		89%		87%
org.apache.commons.collections4.iterators		87%		89%
org.apache.commons.collections4		92%		88%
org.apache.commons.collections4.multiset		74%		59%
org.apache.commons.collections4.functors		85%		85%
org.apache.commons.collections4.list		94%		89%
org.apache.commons.collections4.set		88%		87%
org.apache.commons.collections4.multimap		90%		76%
org.apache.commons.collections4.comparators		83%		69%
org.apache.commons.collections4.bag		91%		88%
org.apache.commons.collections4.collection		93%		92%
org.apache.commons.collections4.sequence		90%		92%
org.apache.commons.collections4.splitmap		83%		66%
org.apache.commons.collections4.queue		97%		97%
org.apache.commons.collections4.bloomfilter		98%		94%
org.apache.commons.collections4.keyvalue		98%		84%
org.apache.commons.collections4.trie.analyzer		92%		70%
org.apache.commons.collections4.properties		98%		92%
org.apache.commons.collections4.bloomfilter.hasher.function		96%		100%
org.apache.commons.collections4.bloomfilter.hasher		100%		100%
Total	6,009 of 56,777	89%	1,035 of 5,975	82%

Figure 2: Statement Coverage and Branch Coverage for Apache Common Collection project

C. Metric 3: Mutation Score (Test-suite Effectiveness)

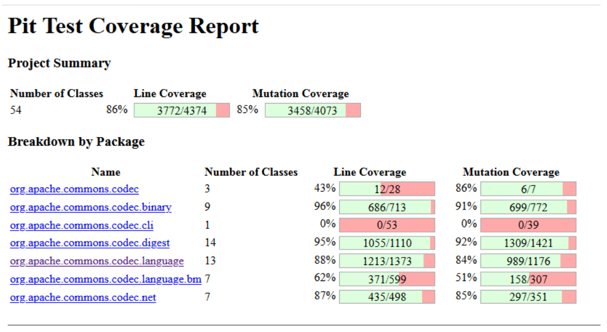
the tool that we use for measuring the mutation score is Pitest. We have to add the following code as a plugin in the pom file under the build to generate the mutation report.



```
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>1.2.0</version>
    <executions>
        <execution>
            <id>pit-report</id>
            <phase>test</phase>
            <goals>
                <goal>mutationCoverage</goal>
            </goals>
            <configuration>
            <outputFormats>CSV</outputFormats>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Figure 3: PIT plugin in pom java

By adding this code in the pom file, a new folder is created in the target folder, which contains the report .The report could be in both html or csv. There is a sample of the html report.We use the csv format but because csv report do not calculate the mutation score for each class . To calculate it we write a java program to calculate the mutation score for each class .



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
54	86%	3772/4374	85%	3458/4073

Breakdown by Package

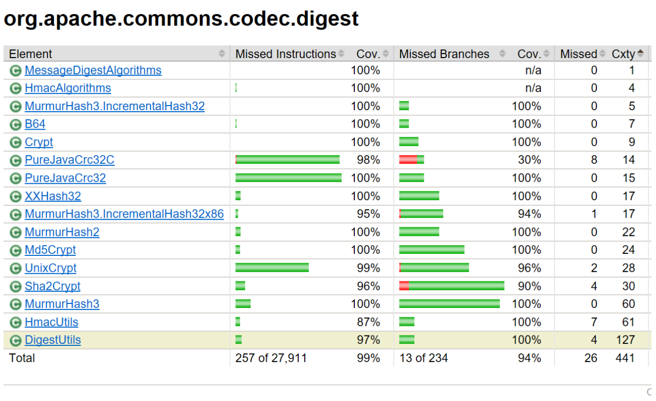
Name	Number of Classes	Line Coverage		Mutation Coverage	
org.apache.commons.codec	3	43%	12/28	86%	6/7
org.apache.commons.codec.binary	9	96%	686/713	91%	699/772
org.apache.commons.codec.cli	1	0%	0/53	0%	0/39
org.apache.commons.codec.digest	14	95%	1055/1110	92%	1309/1421
org.apache.commons.codec.language	13	88%	1213/1373	84%	989/1176
org.apache.commons.codec.language.bm	7	62%	371/599	51%	158/307
org.apache.commons.codec.net	7	87%	435/498	85%	297/351

Figure 4: Pitest mutation report

D. Metric 4: McCabe Complexity

Cycloramic complexity calculating tool and steps are like metric 1 and 2. We used JaCoCo (EclEmma) by declaring the maven plugin in each project's pom.xml file, next to running mvn clean and then mvn install in Eclipse. We called our report folder, jacoco-ut. Therefore, after running these commands, we had our result in target/jacoco-ut folder as jacoco.csv (in csv format) and index.html (in html format).

The Complexity result in both formats are included in two-column named covered complexity and missed complexity. Considering MaCabe Complexity is a class-wise complexity, we calculated each class complexity by adding both covered and missed complexity for that class.



org.apache.commons.codec.digest

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty
MessageDigestAlgorithms		100%		n/a	0	1
HmacAlgorithms		100%		n/a	0	4
MurmurHash3.IncrementalHash32		100%		100%	0	5
B64		100%		100%	0	7
Crypt		100%		100%	0	9
PureJavaCrc32C		98%		30%	8	14
PureJavaCrc32		100%		100%	0	15
XXHash32		100%		100%	0	17
MurmurHash3.IncrementalHash32x86		95%		94%	1	17
MurmurHash2		100%		100%	0	22
Md5Crypt		100%		100%	0	24
UnixCrypt		99%		96%	2	28
Sha2Crypt		96%		90%	4	30
MurmurHash3		100%		100%	0	60
HmacUtils		87%		100%	7	61
DigestUtils		97%		100%	4	127
Total	257 of 27,911	99%	13 of 234	94%	26	441

Figure 5: Class wise McCabe Complexity for Apache Common Codec project

As can be seen in above picture the McCabe Complexity of DigestUtils in Digest package of Apache Common Codec project is equal to 127+4 = 131.

E. Metric 5: Code Churn

In this case study, we compared five specific versions of each project and compared the changes with the master branch.

The absolute measures and methods of data collection are described below:

Total LOC: is the number of lines are computed by LOCMetrics.

Churned LOC: is the sum of the added and changed lines of code between a master branch and a new version of the files (five specific versions of each project)

Deleted LOC: is the number of lines of code deleted between the Master and the new version of the files. The churned LOC and the deleted LOC are computed by the version control systems using a file comparison utility like diff.

File count: is the number of files compiled, available in Git Repository for each release version.

Churn count: is the number of changes made to the files.

F. Metric 6: Post-release defect density

In order to count the number of bugs, we use the Jira bug tracker, which gives us access to the number of bugs based on their versions and shows their status. The link of Jira for each project exists on the project website. Navigating to each project Jira issue tracker, the desired version can be selected by activating the "Affects Version" in advance search part. To calculate post-release defect density, we use KLOC for each project from LOCMetrics, which is a free application used for counting the number LOC.

V. DATA ANALYSIS

A. Spearman Correlation Coefficient

One of the best techniques to find the strength and correlation between two sets of variables is Spearman's correlation coefficient. This analyzing technique is usually used for non-normalized data which is appropriate for our cases.

Having two columns of data to calculate the Spearman's correlation, these are the steps to produce this analysis:

- Rank the data in each column from “1” for the biggest number in a column. The smallest value in the column would have the largest number,

- Add another column titled (d) to the data, which specifies the difference in the ranks,

- Square the differences (d^2), in order to remove the negative values,

- Sum up the d^2 amounts,

- Using the following formula, calculate the coefficient (Rs).

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)}$$

Where

ρ = Spearman's correlation coefficient

di = Ranks difference between corresponding variables

n = number of observations

B. Correlation Between Code Coverage and McCabe Complexity

We expected that the class that has higher code coverage, including statement coverage and branch coverage, will have lower McCabe complexity. To obtain the correlation, we calculate the branch coverage, statement coverage and McCabe complexity at the class level for each project. Jacoco's report provides branch coverage in percentage by default. In order to have statement coverage and McCabe complexity, we use the following formula

Statement coverage = (Line covered) / (Line covered + Missed line)*100

McCabe complexity = Covered complexity + Missed complexity

We calculate the spearman coefficient (R) for each project and use csv report of Jacoco to produce the Spearman graph. The result of each project is shown as follows:

Project	Branch coverage (1)	Statement coverage (2)	McCabe complexity (4)	Spearman 1&4	Spearman 2&4
Apache Commons Codec	91%	97%	22	-0.28	-0.25
Apache Commons Collections	82%	89%	14	-0.33	-0.34
Apache Commons Configuration	83%	87%	18	-0.35	-0.38
Apache Maven Doxia	68%	61%	21	0.03	-0.12

Table 1. Correlation between coverage and McCabe complexity

As is shown in this table all results for statement coverage and most of the results for branch coverage are weak to moderate negative amount. Only one positive weak in Spearman correlation between branch coverage and McCabe complexity which can be neglected. So the correlation between coverage and McCabe complexity is small negative correlation which proves our hypothesis.

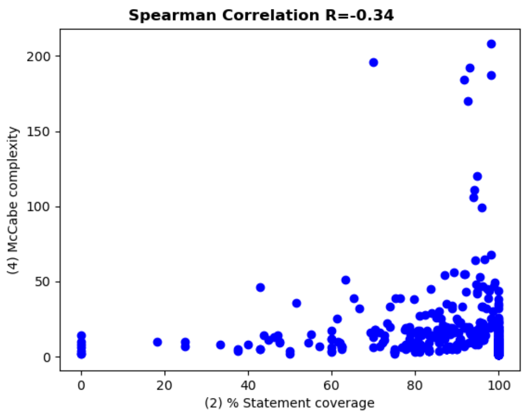


Figure 6: Spearman Correlation between Statement coverage and McCabe complexity for Apache Commons Collections

C. Correlation Between Code Coverage and Mutation Score

We calculate the correlation between code coverage and mutation score in class level. We expected that the results show strong correlation between mutation score and both statement coverage and branch coverage . In the following table you can find the results of correlation for all four projects.

Project	Spearman Correlation between Mutation score and Statement coverage	Spearman Correlation between Mutation score and Branch coverage
Apache Commons Codec	0.53	0.56
Apache Commons Collections	0.52	0.27
Apache Commons Configuration	0.63	0.58
Apache Maven Doxia	0.55	0.51

Table 2. Correlation between coverage and Mutation metric

As it is shown in the table mutation score and both branch coverage and statement coverage are high correlated which approve the hypothesis.

For calculating the correlation we use spearman correlation and bellow there is a sample of scatter plot for Apache Common Codec.

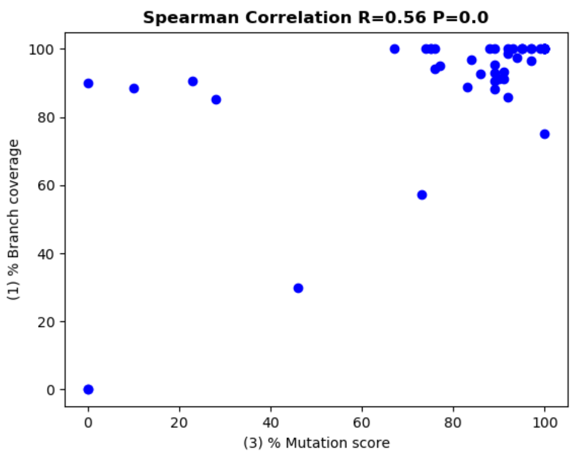


Figure 7: Correlation between coverage and McCabe complexity

D. Correlation Between Code Coverage and Post-release Defect Density

The correlation between coverage, including statement coverage and branch coverage and post release density is done based on version release. For each project we choose five versions and find the defects for each version to calculate the defect density. Then we put Jacoco report for statement coverage and branch coverage for the same five versions. The only exception is that we couldn't run maven for Apache Maven Doxia version 1.6 and earlier versions.

Correlation Between Statement Coverage and Post Release Defect Density			
Project	Version	Post Release Defect Density	Statement Coverage
Apache Commons Codec	1.1	0.746909661	95.50%
	1.11	0.202654778	93.37%
	1.12	0.033153201	93.64%
	1.13	0.193479733	93.97%
	1.14	0.093469591	95.06%
Apache Commons Collections	4	1.335643636	90.88%
	4.1	0.477202057	87.23%
	4.2	0.239134334	89.05%
	4.3	0.029824483	89.07%
	4.4	0.132645542	89.15%
Apache Commons Configurati on	2.3	0.146945788	90.37%
	2.4	0.067528784	90.39%
	2.5	0.033694614	90.39%
	2.6	0.100818873	90.39%
	2.7	0.011267733	90.43%
Apache Maven Doxia	1.6	0.250115438	N/A*
	1.7	0.416501013	84.32%
	1.8	0.378214826	84.27%
	1.9	0.206387699	84.12%
	1.9.1	0.017140017	84.10%

Table 3. Correlation Between Statement Coverage and Post Release Defect Density
* We couldn't run maven for Apache Maven Doxia version 1.6

The spearman correlation between statement coverage and post release defect density is -0.04, which is a very weak negative correlation.

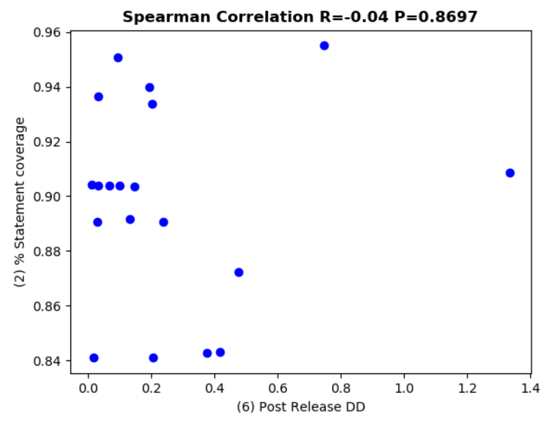


Figure 8: Spearman correlation between Statement Coverage and Post Release Defect Density

The following table shows the correlation between branch coverage and post release defect density. Here spearman correlation is -0.19 which is weak negative number.

Correlation Between Branch Coverage and Post Release Defect Density			
Project	**Version**	**Post Release Defect Density**	**Branch Coverage**
Apache Commons Codec	1.1	0.746909661	92%
	1.11	0.202654778	90%
	1.12	0.033153201	90%
	1.13	0.193479733	90%
	1.14	0.093469591	91%
Apache Commons Collections	4	1.335643636	82%
	4.1	0.477202057	78%
	4.2	0.239134334	81%
	4.3	0.029824483	81%
	4.4	0.132645542	81%
Apache Commons Configuration	2.3	0.146945788	83%
	2.4	0.067528784	83%
	2.5	0.033694614	83%
	2.6	0.100818873	83%
	2.7	0.011267733	83%
Apache Maven Doxia	1.6	0.250115438	NA*
	1.7	0.416501013	66.90%
	1.8	0.378214826	66.70%
	1.9	0.206387699	68.30%
	1.9.1	0.017140017	71.50%

Table 4. Correlation Between Branch Coverage and Post Release Defect Density
* We couldn't run maven for Apache Maven Doxia version 1.6

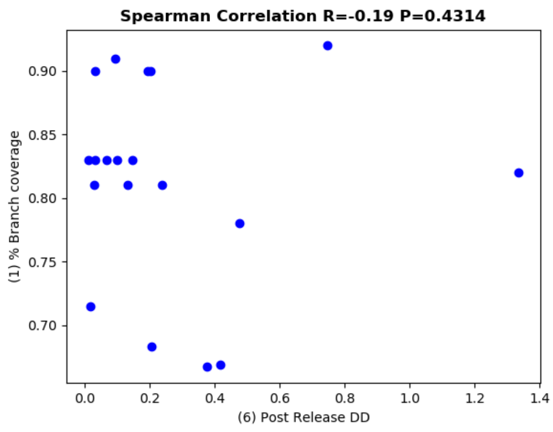


Figure 9: Spearman correlation between Branch Coverage and Post Release Defect Density

E. Correlation Between Relative Code Churn and Post-release Defect Density

High code churn can mean lots of things have changed and need testing or reviewing, so can have direct effect on Post-release Defect Density. In other word, we use code churn to predict the defect density in software systems. 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. Throughout the measuring we assume a statistical significance at 99% confidence corresponding positive increase in the defects/KLOC. This is indicated by the statistically significant positive Spearman rank correlation coefficient.

Correlation Between Code churn and Post Release Defect Density			
Project	**Version**	**Post Release Defect Density**	**Code Churn**
Apache Commons Codec	1.1	0.746909661	16342
	1.11	0.202654778	11564
	1.12	0.033153201	9361
	1.13	0.193479733	6930
	1.14	0.093469591	343

Correlation Between Code churn and Post Release Defect Density			
Project	**Version**	**Post Release Defect Density**	**Code Churn**
Apache Commons Collections	4	1.335643636	54686
	4.1	0.477202057	37326
	4.2	0.239134334	27138
	4.3	0.029824483	8456
	4.4	0.132645542	23091
Apache Commons Configuration	2.3	0.146945788	29787
	2.4	0.067528784	11462
	2.5	0.033694614	8199
	2.6	0.100818873	8098
	2.7	0.011267733	18
Apache Maven Doxia	1.6	0.250115438	30109
	1.7	0.416501013	26645
	1.8	0.378214826	20469
	1.9	0.206387699	5379
	1.9.1	0.017140017	87

Table 5. Correlation Between Code churn and Post Release Defect Density

From the below observations it can be seen that R = 0.73 that shows a strong correlation and we conclude that an increase in relative code churn measures is accompanied by an increase in system defect density.

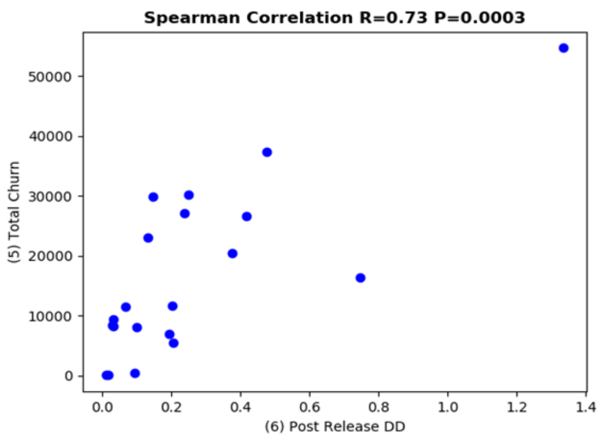


Figure 10: Spearman correlation between Code Churn and Post-release Defect Density

VI. RESULTS

From data analysis we witness that as we expect mutation score has a high positive correlation with statement coverage and branch coverage which means by increasing mutation score code coverage would increase .Code churn and post release defect density are high correlated which means an increase in relative code churn measures is accompanied by an increase in system defect density.

The correlation between post release defect density and code coverage metrics (both statement coverage and branch coverage) is weak negative which means as code coverage increase post release defect density decrease. Also the hypothesis about the correlation between McCabe Complexity and code coverage, statement coverage and branch coverage , is proved which means that the correlation between McCade Complexity and code coverage is small negative.

VII. RELATED WORK

In this case study, we have used six different metrics and four open source Java projects have been measured by those metrics to achieve correlation between them as well as excellent results.

Statement Coverage: Statement coverage is commonly used as a measure of test suite quality Studies show that in large, mature projects, simply measuring the change to statement coverage does not capture the nuances of code evolution[5].

McCabe's cyclomatic complexity: A number of studies have investigated the correlation between McCabe's cyclomatic complexity number with the frequency of defects occurring in a function or method. Some studies find a positive correlation between cyclomatic complexity and defects, when functions and methods have the highest complexity tend to also contain the most defects[6].

Branch Coverage: Various studies show that random testing is an effective way of detecting faults[7]. Random testing is also attractive because it is easy to implement and widely applicable. For example, when insufficient information is available to perform systematic testing, random testing is more practical than any alternative . Many practitioners think that, to evaluate the effectiveness of a strategy, branch coverage –the percentage of branches of the program that the test suite exercises – is the criterion of choice.

Code Churn: In order to find code churn metrics that we could trust, we looked to the studies have been done. This is very important, as studies have shown that using lines of code churned or simple size metrics alone is a poor way to measure defect density[8]. By computing the ratios of certain pairs of metrics, one can obtain much stronger indicators of defect density.

Mutation Score: There has been much research work on the various kinds of techniques seeking to turn Mutation Testing into a practical testing approach. However, there is little survey work in the literature on Mutation Testing.

VIII. REFERENCES

[1] Apache Commons Codec (Last Published: 30 December 2019) https://commons.apache.org/proper/commons-codec/

[2] Apache Commons Collections (Last Published: 09 July 2019) https://commons.apache.org/proper/commons-collections/

[3] Apache Commons Configuration (Last Published: 11 March 2020) https://commons.apache.org/proper/commons-configuration/

[4] Apache Maven Doxia (Last Published: 2020-02-13) http://maven.apache.org/doxia/doxia/

[5] A Large-Scale Study of Test Coverage Evolution http://www.cs.cmu.edu/~mhilton/docs/ase18coverage.pdf

[6] An Analysis and Survey of the Development of Mutation Testing https://web.eecs.umich.edu/~weimerw/481/readings/mutation-testing.pdf

[7] Code churn dashboard https://web.wpi.edu/Pubs/E-project/Available/E-project-021413-170930/unrestricted/CodeChurnDashboard.pdf

[8] Is Branch Coverage a Good Measure of Testing Effectiveness? http://se.ethz.ch/~meyer/publications/testing/coverage.pdf