Concordia University

SOEN-6611 Software Measurements

Instructor Dr. Jinqiu Yang (jinqiuy@encs.concordia.ca)

Programmer On Duty (POD) Zishuo Ding (dingzishuo@gmail.com)
Mehdi Nejadgholi (nejadgholim@gmail.com)

Group Name Group I

Team Members

Sr. No.	Name	Student ID	Email Address
1	Himen Hitesh Sidhpura	40091993	Himens72@gmail.com
2	Chetan Paliwal	40083388	Chetanpaliwal22@gmail.com
3	Karthik B P	40094485	karthikbeepi@gmail.com
4	Sandeep Siddaramaiah	40087428	sandeepsiddaramaiah@gmail.com
5	Rohan Deepak Paspallu	40093648	paspallu.rohan@gmail.com

Replication Package https://github.com/12sandeep1994/SoftwareMeasurementProject

Date April 19, 2019.

A Study of Correlation Analysis among Software Metrics

Himen Hitesh Sidhpura
Department of Computer Science & Software Engineering
Concordia University
Montreal, Canada
himens72@gmail.com

Karthik BP
Department of Computer Science & Software Engineering
Concordia University
Montreal, Canada
karthikbeepi@gmail.com

Sandeep Siddaramaiah
Department of Computer Science & Software Engineering
Concordia University
Montreal, Canada
sandeepsiddaramaiah@gmail.com

Chetan Paliwal
Department of Computer Science & Software Engineering
Concordia University
Montreal, Canada
chetanpaliwal22@gmail.com

Rohan Paspallu
Department of Computer Science & Software Engineering
Concordia University
Montreal, Canada
paspallu.rohan@gmail.com

Abstract—This paper demonstrates a study to determine correlations between different metrics among various projects. Metrics such as Branch coverage, Statement coverage, Mutation testing, Cyclomatic complexity, Number of lines changed, and Software defect density are identified. The data for the calculation of metrics from different projects were obtained using different tools such as CLOC, EclEmma, and PitClipse. The Spearman coefficient is calculated between different metrics with the data collected from tools to determine the correlation between them.

Keywords—CLOC, JaCoCo, AMEffMo, PIT, McCabe, Complexity, Defect Density, Coverage, DLOC, Mutation Testing.

Abbreviations: CLOC: Count Lines of Code, JaCoCo: Java Code Coverage, AMEffMo: Adaptive Maintenance Effort Model, DD: Defect Density.

I. INTRODUCTION

This paper demonstrates the correlation between six different metrics that were calculated from five different projects using various tools. The six metrics calculated are Statement coverage, Branch coverage, Cyclomatic complexity, Mutation score, AMEffMo and Software defect density using different tools such as EclEmma, PitClipse, and CLOC. Metrics such as statement/branch coverage help us identify code coverage and mutation testing help us improve test suite effectiveness in terms of accuracy to detect faults in systems. Code coverage analysis is the process of finding portions of a program not used by a set of test cases, thereby resulting a quantitative measure of code coverage, which is an indirect measure of code quality. Mutation Testing is a type of software testing where we mutate (change) certain statements in the source code and check if the test cases are able to find the errors, this test can be used to improve our test cases Cyclomatic complexity is a quantitative measure of the number of linearly independent paths through the source code. Cyclomatic complexity can be used as a benchmark to compare two different source code. Metric 5 aims to calculate maintenance effort in terms of person-hours and Metric 6 helps us calculate the software quality by calculating the defect density.

This paper addresses the correlation between all the above-mentioned metrics, they are as follows. 1) Correlation between code coverage metrics such as branch coverage and statement coverage with Mutation testing helps us increase test suite effectiveness. 2) Correlation between code coverage metrics with cyclomatic complexity help us identify the complexities of classes/functions which help us in improving the cohesion. 3)Correlation between code coverage metrics with software defect density helps to identify quality of the software based on the test coverage. 4)Correlation between maintenance effort and defect density helps us to determine whether an increase in the number of defect decrease/ increase effort or not and vice-versa.

Rest of the paper is divided into following sections: Section II describes the project and metric identification, Section III describes methodology which consists of tools used, how data was collected & analyzed, Section IV presents results which elaborates the correlation which we have defined above, Section V and Section VI focuses related work and conclusion of this paper.

II. PROJECT AND METRIC IDENTIFICATION

This section describes projects which are used for doing correlation analysis and various metrics used to do correlation among themselves between various versions of the project.

A. PROJECT IDENTIFICATION

There are various projects which can be undertaken to perform correlations among various metrics. Out of the many free and open-source projects, five well known projects were considered for analysis purposes as they are well-maintained and have more than five versions with bug reports on JIRA. These projects are open-source and all of them can be built using maven. They are:

1) **Apache Commons Logging:** The Apache Commons Logging is a java-based logging utility library which

provides many API's and logging implementations which helps us in logging run-time information [1]. Log and LogFactory are the two basic abstractions used in this project [1].

Project Link: https://commons.apache.org/proper/commons-logging/ [1]

Source Code: https://github.com/apache/commons-logging [2]

2) Apache Commons Lang: The Apache Commons Lang is used as a helper class for the utilities in the java.lang API [3]. The library acts as a host for providing various functionalities such as String manipulation methods, basic numerical methods, object reflection, concurrency, creation and serialization, and System properties [3].

Project Link: https://commons.apache.org/proper/commons-lang/ [3]

Source Code: https://github.com/apache/commons-lang [4]

3) Apache Commons Collections: The Apache Commons Collections is a Collections Framework which is an advancement to the earlier versions to include various data structures that help us in faster development of Java applications [5]. It is a standard for collection handling in java. The Java collections are built on JDK classes by providing new interfaces, implementation, and utilities [5].

Project Link: https://github.com/apache/commons-collections [5]

Source Code: https://commons.apache.org/proper/commons-collections/ [6]

4) Apache Commons Math: The Apache Commons Math is a self-contained mathematics and statistics library that solves the most common problems in a java program which are not included in Java programming language or Commons Lang [7].

Project Link: https://github.com/apache/commons-math [7]

Source Code: https://commons.apache.org/proper/commons-math/ [8]

5) JFreeChart: JFreeChart is a chart library that is used for the development of various types of charts (2D or 3D, Bar charts, line charts, XY plots, Scatter Plots, etc.). It is suitable for use in various applications, applets, etc [9]. Moreover, it has a consistent and well-documented API and supports outputs from various types, like Swing, JavaFX, image files, etc [9].

Project Link: http://www.jfree.org/jfreechart/ [9]

Source Code: https://github.com/jfree/jfreechart [10]

Table 1: Project and their version used.

Project	Version	SLOC
Apache commons Lang	3.8.1	9.6
Apache commons Logging	1.1.2	79.8
Apache commons Collection	4.3	132
Apache Commons Math	3.6.1	186
JFreechart	1.5	167

Table 1 shows the five projects which were selected along with the latest versions and SLOC i.e. source lines of code at the time of study.

B. METRIC IDENTIFICATION

This section demonstrates various metrics which have been selected for the analysis and their working. This metric is calculated for various versions of each project

1) **Statement Coverage:** Statement coverage is a measurement procedure wherein the number of lines of code covered under certain circumstances is calculated based on the inputs from the users [11]. In simple words, the number of statements executed by a program based on the user inputs.

2) **Branch Coverage:** Branch coverage is a measurement procedure to calculate the number of branches executed per user input. The branch coverage mostly includes conditional testing, wherein a branch is executed when a condition is fulfilled [11] [12].

3) **Mutation Score:** Mutation score is a testing practice where the accuracy or precision of the test cases is checked [13]. In mutation testing, mutants are introduced in the main code and is tested with the designed test cases/ test suites. If the test case recognizes the changes/mutants and throws an error, the mutants are said to killed by the test case [14]. The mutation score can be calculated as:
Mutation Score = (Killed Mutants / Total number of Mutants) * 100

4) **Cyclomatic Complexity:** Cyclomatic Complexity is the measure number of linearly independent paths that can be executed in a program [15]. Cyclomatic complexity is used to determine the number of test cases/test suites to be designed for a program, in a sense of covering all the possible paths which can be executed [15].

5) Adaptive maintenance Effort Model (AMEffMo): This model is used to determine maintenance effort in person-hours [16]. While deriving this model, it was found that the number of lines of code changed and several operators changed are found to be strongly correlated to maintenance effort. Several metrics can be identified from the software development process, but among all of them, a number of lines of code changed and a number of operators changed are strongly correlated [16]. To build this model, 70% of the collected data is used while the remaining 30% is used to validate the model. In this model, simple and multiple regression is performed, and it was found that maintenance effort E in person hour is **E = 78 + 0.01 DLOC,** where DLOC is the number lines of code changed [16].

6) **Software Defect Density:** It is always desirable to understand the quality of a software system based on static code metrics [17]. In this metric, we analyze the relationships between Lines of Code (LOC) and defects. We are relating the external software

characteristic(quality) with internal product attribute (LOC). It counts each physical source line of code in a program, excluding blank lines and comments [17]. And the number of defects is the count of defects that were posted after the release of a version of the software. Defect density (DD) can be calculated as:
DD = number of defects/ KLOC, where KLOC is number of lines of code divided by thousand.

III. METHODOLOGY

A. TOOL IDENTIFICATION

This Section describes various tools used to calculate various metrics. The tools used in this study are:

1) EclEmma

It is an eclipse plugin which is used to determine the code coverage in a software project [15]. This tool is preferred due to several reasons:

- It helps to determine the metrics such as statement, branch and cyclomatic complexity.
- Provide results in various formats like CSV, HTML, XLSX etc. [15].
- Works well with all the released versions of Java.
- Open-source and well maintained.

However, it has some disadvantages:

- There is no mention of the abstract classes defined in the project [15].
- Class files with syntactic or semantic errors lead to exceptions [15].

Working of EclEmma:

It uses a set of counters to calculate the coverage of the software projects. These counters are basically extracted from the Java class files [15]. This allows the tool to be run even though there is no source code available. The smallest units counted by this tool is the Java bytecode instructions [15].

2) PitClipse

It is an eclipse plugin which performs mutation testing on Java projects [18]. This tool introduces mutants in the code to determine the faults in the source code by the given test cases [13]. Additionally, it also determines which code has never been tested as well. PIT originally stood for Parallel Isolated Test [13].

Due to various reason this tool is preferred over other tools, they are:

- Integration with our existing environment like Eclipse, Maven, etc.
- Ease of deployment and fastest compared to its competitors [13].
- Free and open-source library.
- Produced the reports in a variety of formats like HTML, XLSX, CSV etc. [13].

However, it has some disadvantages:

- It requires Java 5 or above and either JUnit or TestNG [13].
- Though the tool is fast there are instances in which there were frequent timeouts which prompted another run of this process.
- Some of the test cases have to be ignored if running of those test cases require other configurations (e.g. Server level configurations).

Working of PitClipse:

PitClipse introduces faults or mutations into the given code and the test suites are run. If the test suites fail, then the mutants have successfully been killed [13]. Otherwise, the mutants have survived which signifies that the test suites were not effective enough. Ideally, all the mutants must be killed. Pit directly applies to the bytecode by applying a set of configurable mutators [13] [19]. While performing mutation testing, the default mutators of PIT were used [19]. These mutators are:

- CONDITIONALS_BOUNDARY_MUTATOR
- INCREMENTS_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR

3) CLOC

It is a tool which is used to count lines of code in a program [20]. This tool can determine the lines of code of changes between two version of the project an also determines a total number of lines of code in the project [20]. Due to several reasons, this tool is preferred over other tools:

- Free and well maintained open-source tool.
- Can work with a wide array of programming languages [20].
- Added functionality to compare lines of code between versions.

Working of CLOC:

CLOC first determines the list of files to consider by using the 'File::' Find and ignores binary and zero sized files. After that it ignores the duplicate files in the candidate list by comparing file sizes, then, for similarly sized files, compare MD5 hashes of the file contents with 'Digest::' MD5. Then attempts to identify whether the found files contain recognized programming language code. Finally, it invokes language-specific routines to determine the lines of code. While comparing the differences the tool has a --diff switch which helps us measure the relative change in source code and comments between two versions of a file, directory, or archive. Differences reveal much more than absolute code counts of two file versions. For example, say a source file has 100 lines and its developer delivers a newer version with 102 lines. Did he add two comment lines, or delete seventeen source lines and add fourteen source lines and five comment lines, or did he do a complete rewrite, discarding all 100

original lines and adding 102 lines of all new source? The diff option tells how many lines of source were added, removed, modified or stayed the same, and how many lines of comments were added, removed, modified or stayed the same which is essential for us to calculate our Metric 5 which is effort for making changes [20].

Table 2: Tools and their version used

Tool	Version
EclEmaa	3.1.2
PitClipse	1.4.6
CLOC	1.80

B. DATA COLLECTION

This section demonstrates how tools are integrated with projects to generate data for analysis in this study. Every tool has different dependencies which are integrated into development environment and then the tool was run for the data to be generated.

1) INTEGRATION OF ECLEMMA

EclEmma was integrated with eclipse to generate the statement & branch coverage and cyclomatic complexity of the project [15]. To integrate EclEmma with eclipse, EclEmma requires that eclipse is 3.8 or higher with Java 1.5 or higher [15] to determine the code coverage and complexity.Once integrated, EclEmma can generate code coverage and complexity by building the project. The coverage report is generated as shown in Figure 1.

CLASS	INSTRUCTION_MISSED	INSTRUCTION_COVERED
AbstractSortedBidiMapDecorator	24	8
TreeBidiMap.View	0	58
AbstractDualBidiMap.EntrySet	0	70
DualTreeBidiMap.ViewMap	28	40
AbstractDualBidiMap	0	368
AbstractDualBidiMap.KeySetIterat	0	57
UnmodifiableOrderedBidiMap	2	81
UnmodifiableSortedBidiMap	5	106

Figure 1: Snapshot of coverage report generated by EclEmma for Apache Commons Collections v3.2.1

2) INTEGRATION OF PITCLIPSE

PitClipse was integrated with eclipse to generate the mutation score and line coverage of the project [18]. To integrate PitClipse with eclipse, PIT requires Java 5 or above and TestNG or JUnit to be on the classpath [13] to determine the code coverage and complexity. Once PitClipse is integrated, various dependencies need to be integrated with the project to perform mutation testing of the project [13]. To generate a mutation score, the following steps are needed to be performed:

- Step 1: The plugins are added to the pom.xml files of the projects [13].

```
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>LATEST</version>
 </plugin>
```

Figure 2: Plugins added in POM file

- Step 2: Once the dependency is added, run the command as shown in Figure 3 to generate mutation score.

```
mvn org.pitest:pitest-maven:mutationCoverage
```

Figure 3: Command Executed to generate mutation score

Once the command is executed, pit report will be generated in HTML format at package level as well as class level [13].

When mutation testing is performed, there are few test cases which need to be ignored as some of them don't have green suite i.e. they need certain configuration which are done at server level or system level [13]. For some of the projects used for analysis, 2-3 test cases were ignored.

3) INTEGRATION OF CLOC

Unlike other tools mentioned above, this tool doesn't need any integration at system-level or project-level. To run this, following dependencies are to be added and steps need to perform to generate metric 5 and metric 6 data:

- First, the dependencies of CLOC needs to be installed. These dependencies are NodeJs and Perl [20].
- Once this dependency is installed/configured. CLOC command is executed in the command line as shown in below Figure 7 [20].

```
prompt> cloc
Usage: cloc [options] <file(s)/dir(s)> | <set 1> <set 2> | <report files>
Input Options :
              --diff <set1> <set2>
              --csv  :  Write the results as comma separated values.
              --out=<file>  :  Synonym for --report-file=<file>.
```

Figure 4: Format of Command used to generate metric 5 and metric 6 data.

Figure 5 below shows the output of data using CLOC command.

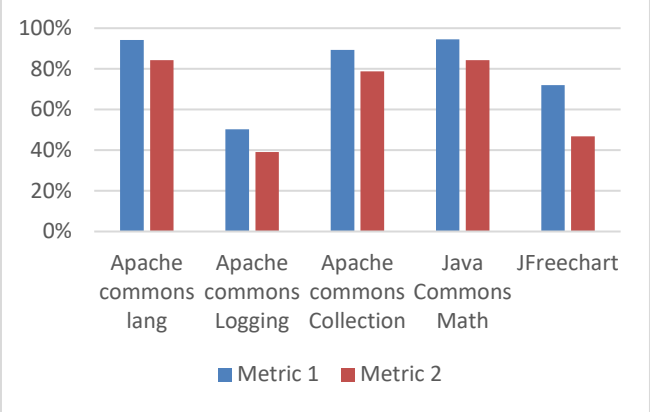
Language	files	blank	comment	code
Java				
same	4	60	8842	10960
modified	71	0	506	839
added	56	2153	8791	7688
removed	9	300	961	1842

Figure 5: Snapshot of data generated using CLOC Command for Apache Commons collections v2.1 for metric 5

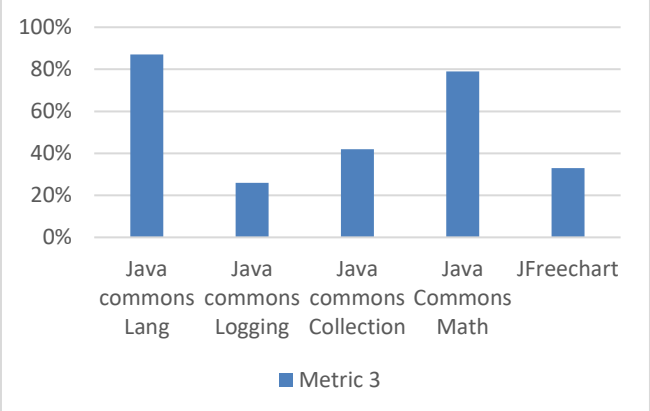
C. DATA ANALYSIS

After collecting data using the aforementioned tools, the various metrics were calculated. For metric 6, the number of bugs in specific version are identified from their JIRA bug reports. As it was observed that the data obtained is not normally distributed, Spearman correlation was preferred. To identify the correlation between metrics, the study used python 3 with Jupyter notebooks.

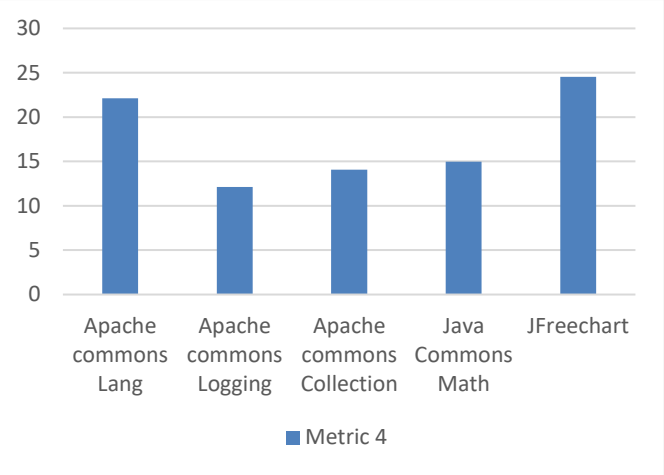
Below Graph 1, Graph 2, Graph 3, Graph 4 and Graph 5 shows metric 1, metric 2, metric 3, metric 4, metric 5 and metric 6 results for their corresponding latest versions mentioned in Table 1.



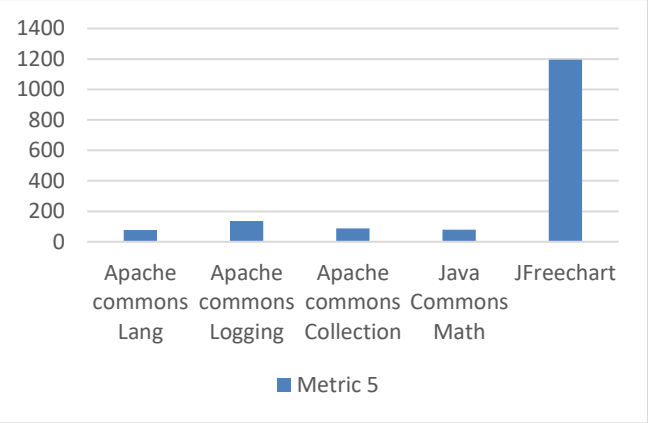
Graph 1: Branch and statement coverage(in percentage) for the project in their versions mentioned in Table 1



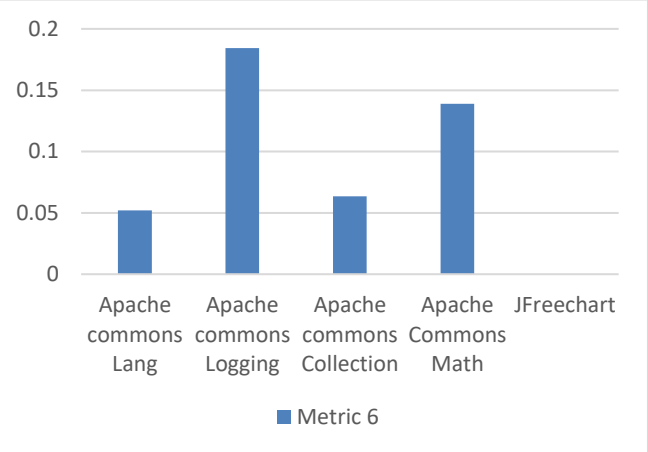
Graph 2: Mutation score for the project(in percentage) in their versions mentioned in Table 1.



Graph 3: Average cyclomatic complexity of classes of every project in their versions mentioned in Table 1.



Graph 4: Maintenance effort required(in hours) for each of the projects in their versions mentioned in Table 1.



Graph 5: Software defect density(defects per 1000 lines of code) for each of the projects in their versions mentioned in Table 1.

Note : In Graph 5 number of defects for JFreechart is zero for the version mentioned in Table 1.

IV. RESULTS

A. Correlation between Metric 1 and Metric 3

Initially, the study assumed that there is a strong positive correlation between metric 1 and metric 3 in the proposal. After performing correlation, it was found that assumption is closely aligned to the results obtained in the study as there is a strong positive correlation between statement coverage and mutation score. The study was performed at project level as shown in Figure 6.

```
print(spearmanr(statement_coverage_results, mutation_test_results))
```

```
SpearmanrResult(correlation=0.8999999999999998, pvalue=0.03738607346849874)
```

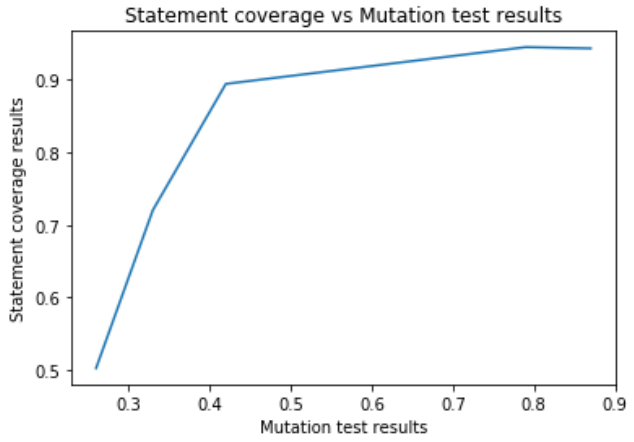


Figure 6: Correlation between Metric 1 and Metric 3

B. Correlation between Metric 2 and Metric 3:

Again, the study assumed that there was a strong positive correlation between metric 2 and metric 3 in the proposal.

```
print(spearmanr(branch_coverage_results, mutation_test_results))
```

```
SpearmanrResult(correlation=0.9999999999999999, pvalue=1.4042654220543672e-24)
```

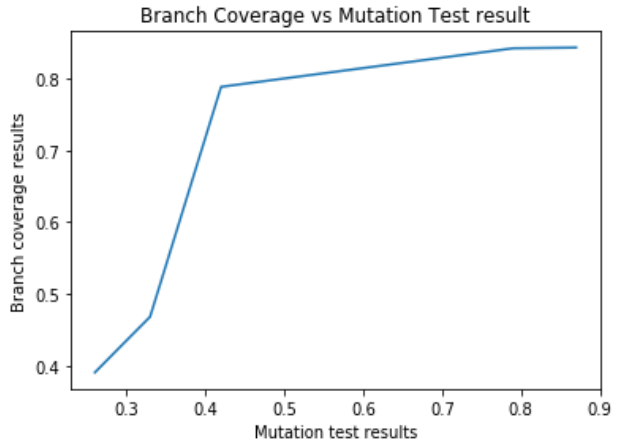


Figure 7: Correlation between Metric 2 and Metric 3

After the study was conducted, it can be concluded that it is closely aligned to the study's initial assumption as there is a strong positive correlation between branch coverage and mutation score. The study was performed at project level as shown in Figure 7.

C. Correlation between Metric 1, Metric 2 and Metric 4

Interestingly, the correlation between cyclomatic complexity and statement coverage were slightly negatively correlated for the all the projects. In the proposal, it was assumed that Metric 1 and Metric 4 are positively correlated but after performing correlation on individual classes of the projects, it was found that when complexity increased there was no gradual increase statement coverage. Hence, initial assumptions were proven to be false with regards to the study conducted on these projects.

```
spearmanr(cyclomatic_coverage_math_results, statement_coverage_math_results)
```

```
SpearmanrResult(correlation=-0.3856866917423484, pvalue=4.4957668396702057e-82)
```

```
spearmanr(cyclomatic_coverage_math_results, branch_coverage_math_results)
```

```
SpearmanrResult(correlation=0.5867474940728188, pvalue=7.525935528328529e-212)
```

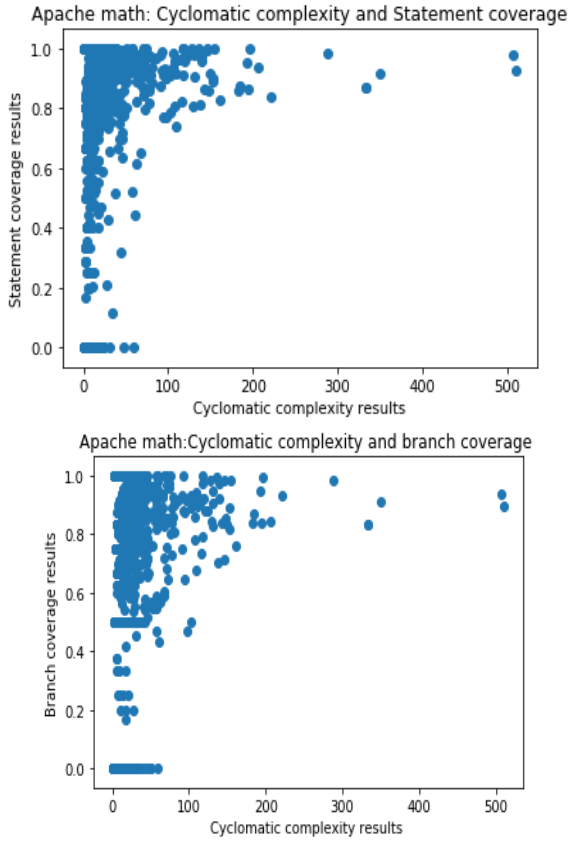


Figure 8: Correlation between Metric 1, Metric 2 and Metric 4 for individual classes in the project Apache Commons Math.

As expected, cyclomatic complexity and branch coverage were moderately positively correlated for the individual classes of all projects. Hence, the study's initial assumption that branch coverage and cyclomatic complexity are

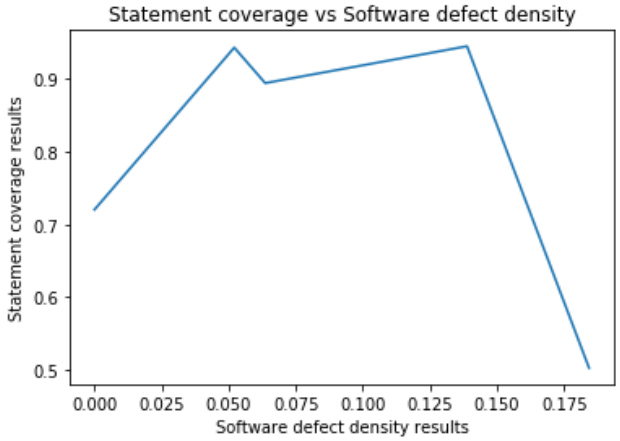
positively correlated was correct. As the complexity increased there was gradual increase in branch coverage. The study was performed at class level as shown in the Figure 8.

D. Correlation between Metric 1, Metric 2 and Metric 6

Metric 1 and Metric 6's correlation again was contrary to initial assumptions and showed mild negative correlation as in our selected projects, bug density is not directly related to the statement coverage. The same trend is noticed in Metric 2 and Metric 6 as well. The study was performed at project level as shown in Figure 9.

```
print(spearmanr(statement_coverage_results, software_defect_density))
```

```
SpearmanrResult(correlation=-0.09999999999999999, pvalue=0.8728885715695383)
```




```
print(spearmanr(branch_coverage_results, software_defect_density))
```

```
SpearmanrResult(correlation=-0.3, pvalue=0.6238376647810728)
```

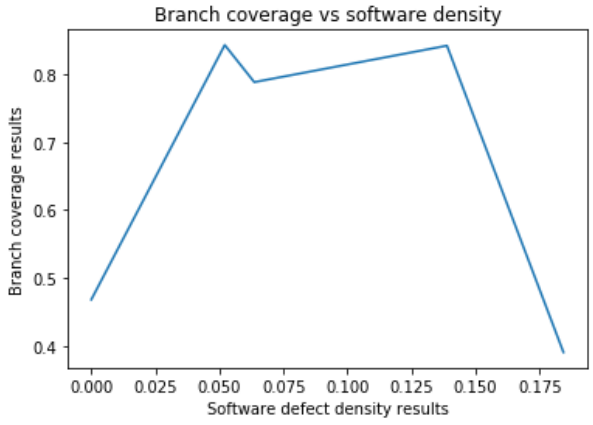


Figure 9: Correlation between Metric 1, Metric 2 and Metric 6

E. Correlation between Metric 5 and Metric 6

Metric 5 and Metric 6's Spearman coefficient again was a mixed bag as two projects showed medium positive correlation while the other projects showed negative correlation but with a very high p-value. This translated to a very obscure correlation. The study perceives, if more projects with more versions were considered, this could have given a better conclusive result. The study was performed at project level between versions.

```
print(spearmanr(software_defect_density_results,
                adaptive_maintenance_effort_results))
```

```
SpearmanrResult(correlation=0.5174825174825175, pvalue=0.08486877113393493)
```

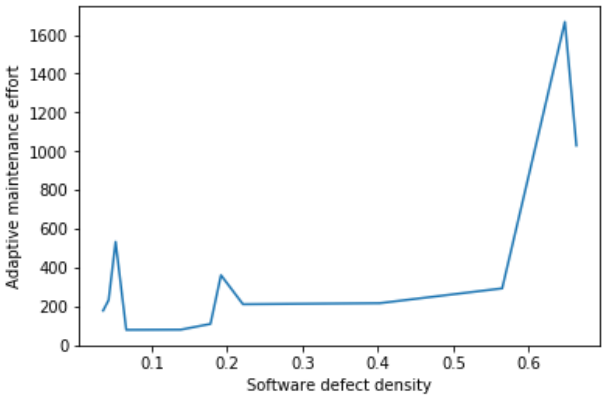


Figure 10: Correlation between Metric 5 and Metric 6 for Apache Commons Math

V. RELATED WORK

Analyzing the relation between lines of code and software quality is an intensively investigated research area. The paper, studied the impact of maintenance effort which is calculated based on change of source line of code with software quality, calculated based on software defect density.

Andersson and Runeson [21] analyzed that 20% of the largest module of a telecommunication project was responsible for 26%, 18% and 57% of the defects. The first two projects showed poor correlation between source line of change and the defect density while the third project with 57% defect showed a good correlation with software defect density.

In a similar study, Dinesh Verma and Shishir Kumar described one approach to reduce the software Defect Density Using Optimal Module [22] sizes.

In the paper "A metrics-based software maintenance effort model" [16] J. Hayes, S. Patel and L. Zhao showed a method to estimate the effort based on the number of source line of code change and a number of operators changed. They have studied all the possible metrics which can affect the maintenance effort and through correlation analysis ranked the metrics based on their impact on maintenance effort.

VI. CONCLUSION AND FUTURE WORK

This paper co-relates various metrics i.e. metric 1 to metric 6 as defined earlier. While the relations between metric 1, metric 2, metric 3 and metric 4 are almost on the expected levels, more interesting correlations were observed between metric 5 and metric 6. After analyzing various versions of the

five selected projects, the study found that for two projects the study got a positive correlation while the other three were negatively correlated. During analysis, it was observed that range of values which are generated after performing correlation is not enough due to the limitations of the data collected. This observation suggest that correlation values will improve if variety of projects across diverse domains is considered.

One of the tasks for the future work would be to run the tests on really large projects as the largest project we have selected has around 200k lines of code. If we run the tests on a large project with more versions, we may get a better result. One more task would be to select different types of projects, we have mostly considered java projects and the bugs were also mostly software bugs, it would be interesting for example to select a web project and then categorize the bugs and efforts based on UI, functionality etc.

VII. REFERENCES

[1] A. C. Logging, "https://commons.apache.org/proper/commons-logging/".

[2] L. Github, "https://github.com/apache/commons-logging".

[3] A. C. Lang, "https://commons.apache.org/proper/commons-lang/".

[4] L. Github, "https://github.com/apache/commons-lang".

[5] A. C. Collections, "https://commons.apache.org/proper/commons-collections/".

[6] C. Github, "https://github.com/apache/commons-collections".

[7] A. C. Math, "https://commons.apache.org/proper/commons-math/".

[8] M. Github, "https://github.com/apache/commons-math".

[9] JFreechart, "http://www.jfree.org/jfreechart/".

[10] J. Github, "https://github.com/jfree/jfreechart".

[11] Atlassian, "https://confluence.atlassian.com/clover/about-code-coverage-71599496.html," *About Code Coverage.*

[12] SmartBear Software, "https://support.smartbear.com/collaborator/docs/reference/metrics.html".

[13] pitest.org, "Mutation testing," *pitest.org.*

[14] GURU99, "Mutation Testing in Software Testing: Mutant Score & Analysis Example," *https://www.guru99.com/mutation-testing.html, 2019.*

[15] EclEmma, https://www.eclemma.org/.

[16] J. Hayes, S. Patel and L. Zhao, "A metrics-based software maintenance effort model," IEEE, Finland, 2014.

[17] H. Zhang, "An Investigation of the Relationships between Lines of Code and Defects," *IEEE.*

[18] MarketPlace Eclipse, "https://marketplace.eclipse.org/content/pitclipse".

[19] pitest.org, http://pitest.org/quickstart/mutators/.

[20] SoruceForge, http://cloc.sourceforge.net.

[21] C. A. a. P. Runeson, "A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems," *IEEE.*

[22] S. K. Dinesh Verma, "An Improved Approach for Reduction of Defect Density Using Optimal Module Sizes. Hindwai Publishing Corporation," *IEEE.*