SOEN-6611 Software Measurements

Instructor Dr Jinqiu Yang (jinqiuy@encs.concordia.ca)
Programmer On Duty (POD) Zishuo Ding (dingzishuo@gmail.com)

Team Name Team M

Team Members

Sr. No.	Name	Student ID	Email Address
1	Navroop Virk	40043821	virknavroop@gmail.com
2	Harsh Mehta	40098439	harshmehta493@gmail.com
3	Yash Golwala	40085663	golwalayash@gmail.com
4	Raghav Dutta	40067534	raghav.dutta29@gmail.com
5	Manisha Jalota	40079362	manisha.jalota223@gmail.com

Replication Package:- https://github.com/mharsh1995/Software_Measurement_Team_M

Date - 25/6/2019

A Study of Correlation Analysis among Software Metrics

Navroop Virk
Department of Computer Science & Software Engineering
Concordia University
Montreal, Canada

Harsh Mehta
Department of Computer Science & Software Engineering
Concordia University
Montreal, Canada

Yash Golwala
Department of Computer Science & Software Engineering
Concordia University
Montreal, Canada

Raghav Dutta
Department of Computer Science & Software Engineering
Concordia University
Montreal, Canada

Manisha Jalota
Department of Computer Science & Software Engineering
Concordia University
Montreal, Canada

Abstract—This paper demonstrates the study of the correlation between different software metrics such as Branch coverage, Statement Coverage, Mutation testing, McCabe complexity, Adaptive maintenance model which includes counting number of modified lines and defect density for 5 Open system projects. The Data is collected using tools such as Jaococo, PIT, CLOC and analyzing the bug tracking repositories like Jira. 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: AMEffMo: Adaptive Maintenance Effort Model,CLOC: Count Lines of Code, JaCoCo: Java Code Coverage, DD: Defect Density.

I. INTRODUCTION

This paper depicts the correlation between six metrics which are calculated for the five open system projects. The software metrics are Branch coverage, Statement coverage, Mutation Testing, Cyclomatic complexity, Adaptive maintenance model and the defect density. In order to estimate the code coverage of the projects, Coverage metrics such as Branch and Statement coverage are calculated using the Jacoco tool. This information can then be used to improve the test suite, either by adding tests or modifying existing tests to increase coverage. Mutation Testing includes modification of statements in code in order to detect faults in the program and which further uncovers ambiguities in the source code. This metric is calculated using PIT. In order to calculate the Adaptive maintenance model, Line of code and the modified lines in different versions of projects are determined using CLOC. This data is further used to calculate maintenance effort in terms of person-hours. Defect Density is calculated by analyzing the bug reports in the Jira repository for different versions.

In the above context, the purpose of this paper is to present an investigation we have conducted to correlate the above-mentioned software metrics. The correlation between the code coverage metric (Branch and statement) and Mutation Testing help us to increase the effectiveness of the test suites. Correlation between code coverage metrics with cyclomatic complexity helps us identify the complexities of classes which help us in improving the cohesion. 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. Five open source software were analyzed and the metrics were collected and correlated using the Spearman's rank order.

The remainder of this paper is organized as follows- Section II describes the metric identification, Section III describes the process of Data Collection and Analysis. Section IV depicts the results of the analysis which includes the correlation of the metrics. Section V and Section VI presents related work and conclusion of this paper.

II. METRIC IDENTIFICATION

This section describes the metrics which are identified to be applied to the projects and later used to find the correlation among themselves when applied on different versions of the project.

A. PROJECT INFORMATION

We have considered five open source projects for the data collection and analysis purpose as these projects met the requirement of LOC and have enough versions with bug reports on JIRA. All of the projects are built using maven and three of the projects have more than 100K LOC.

1) Apache File Upload: The Apache Commons File Upload is a Java Library which helps us upload large files over the HTTP protocol using the multipart/form-data content type. It provides a

simple and flexible means of adding support for multipart file upload functionality to servlets and web applications. [9].
Source-Code-
https://archive.apache.org/dist/commons/fileupload/source/ [9]
Github link -
https://github.com/mharsh1995/Software_Measurement_Team_M/tree/master/Projects/FileUpload

2) Apache Commons Collection -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 related to building the JDK classes by providing new interfaces, implementations and utilities.
Source-Code-
https://commons.apache.org/proper/commons-collections/download_collections.cgi[6]
Github link -
https://github.com/mharsh1995/Software_Measurement_Team_M/tree/master/Projects/Collections

3) Apache Commons Math - 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.[7]
Source-Code-
https://commons.apache.org/proper/commons-math/download_math.cgi[7]
Github link -
https://github.com/mharsh1995/Software_Measurement_Team_M/tree/master/Projects/Math

4) Apache Commons Pool - The Apache Commons Pool open source software library provides an object-pooling API and a number of object pool implementations.[6]
Source-Code-
https://archive.apache.org/dist/commons/pool/source/
Github link -
https://github.com/mharsh1995/Software_Measurement_Team_M/tree/master/Projects/Pool

5) Apache Commons IO - The Apache Commons IO is a library of utilities to assist with developing IO functionality. These can be useful in the implementation of utility classes, filters, comparators.[6]
Source-Code- https://www.apache.org/dist/commons/io/source/
Github link -
https://github.com/mharsh1995/Software_Measurement_Team_M/tree/master/Projects/IO

Table 1- Projects and Version

Project	Version	Line of Code (LOC)
Commons File Upload	**1.4**	**100K**
Commons Collection	**4.4**	**270K**
Commons Maths	**3.5.1**	**174K**
Commons Pool	**2.6.2**	**14.3K**
Commons IO	**2.6**	**58.4K**

B. METRIC IDENTIFICATION

This section describes the selected metrics which are used for Data Collection and Data Analysis. These metrics are calculated for various versions of each project and then the analysis is done on the collected data.

1. **Statement Coverage -** Statement coverage is a white box test design technique which involves the execution of all the executable statements in the source code at least once. It is the number of statements which are executed depending on what the user inputs. It is the (number of executed statements/Total number of statements) *100.

2. **Branch Coverage -** In Branch coverage, every outcome is tested. It makes sure that every branch is executed at least once. It also helps to determine the code which is independent and does not have any branches,
Branch Coverage = Number of Executed Branches / Total Number of Branches

3. **Mutation Score -** This is the kind of testing in which we can change the code and we will verify whether the introduced errors are detected by test cases. It measures the quality of the test cases. It is also called Fault-based Testing Strategy.
Mutation Score = (Killed Mutants / Total number of Mutants) * 100

4. **Cyclomatic Complexity -** Cyclomatic complexity is a quantitative measure of the number of linearly independent paths through the program's source code. Cyclomatic complexity is used as a benchmark to compare two different source code. [11] The program with high cyclomatic complexity is more error-prone and require more understanding of testing. It also helps us in determining the

number of test cases that will be required for complete branch coverage. [11]

5. **Adaptive maintenance Effort Model (AMEffMo) -** This model measures the maintenance effort in terms of person-hours.It was determined that the change in the number of lines of codes is highly correlated with the efforts. The effort for this model is calculated using the below equation -
   E = 78 + 0.01 DLOC where E is effort and the DLOC is the number of lines of code changed.

6. **Software Defect Density -** It is defined as the number of defects during the period of software development by the size of the software. The Algorithm used to calculate the defect density is a) Calculate the number of defects arising in different modules(release/built/cycle). b) Calculate the size of the software that can be measured in KLOC.c) At last use below equation to calculate the defect density -
   Defect Density =(Total Number of defects/size of the software).

III METHODOLOGY

A. TOOL IDENTIFICATION

This section explains the different tools used in calculations of all the metrics. We used three tools for different calculations:

1. **EclEmma** - To get the code coverage EclEmma is used which is an Eclipse Plugin as there various benefits of this tool over such as:

- We can find statement coverage, branch coverage and cyclomatic complexity using EclEmma.
- We can get results in various formats such as CSV, HTML, XLSC etc.
- This is an open source and works equally well for different versions of Java.

Apart from this benefit, there are some demerits of using this Tool. In some cases, it generates exceptions due to class files with syntactic or semantic errors. It does not consider abstract classes in the project.

Working:

To calculate the coverage of software it uses a set of the counter which is usually taken from the classes of Java. So even the code is not available it runs well. Even the instructions are counted as the smallest parts of the software.

2. **PitClipse -**
   PitClipse is an eclipse plugin that is used for mutation testing. PitClipse makes uses of mutants to find the faults and errors in a project. This also covers the code that is not even tested. Here are some advantages of PitClipse:
   a) We can integrate it with various environments such as Eclipse, IntelliJ.
   b) It is easy to deploy using PitClipse and it is the fastest tool.
   c) This is free of cost.
   d) It is open source and can generate the results in various formats such as HTML, XLSX, CSV etc.

But the cons are that it requires the java 5 Version or above. Also frequent timeouts asometimesJavait needs reconfiguration.

Working:

It generates the mutants in code and then the files are run using mutants.Failure of test cases explains that all the mutants are killed while if the test cases are passed that means all the mutants are not killed that describes the low effectiveness of test suite.The ideal test suit is one that kills all the mutants. Some of the mutators used for mutation testing are:

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

3. **CLOC -** CLOC is used to count the number of lines of code in the program. The tool also has the tendency to count the number of lines in the whole project. There are many reasons why this tool is preferred over other tools:
   1. This is also free and open source and can work with a different language.
   2. Also, with this tool, we can compare different lines of code.

Working of CLOC

Cloc makes a count of commented lines, blank lines, physical lines of code in different languages.Cloc is in Perl language.

B. DATA COLLECTION

This section describes the integration of the tools with projects. Data is collected using these tools and further analysis is performed. Every tool has different dependencies which are integrated into the development environment and then the tool was run for the data to be generated.

1) INTEGRATION OF JACOCO

JACOCO was integrated with eclipse by adding dependencies in the pom.xml . This is used to generate statement & branch coverage and cyclomatic complexity of the projects. After integration, it can generate code coverage and complexity by building the project. The coverage report is depicted as below-

Apache Commons FileUpload

Element	Missed Instructions	Cov.	Missed Branches	Cov.
org.apache.commons.fileupload		80%		75%
org.apache.commons.fileupload.disk		67%		59%
org.apache.commons.fileupload.util.mime		88%		84%
org.apache.commons.fileupload.servlet		44%		0%
org.apache.commons.fileupload.util		82%		88%
org.apache.commons.fileupload.portlet		47%		n/a
Total	1,037 of 4,944	79%	112 of 468	76%

2) INTEGRATION OF PITCLIPSE

Piteclipse was integrated with eclipse by adding dependencies in the pom.xml. It generates the mutation score and line coverage of the project. The mutation score is generated by adding below dependencies-

```
<plugin>
  <groupId>org.pitest</groupId>
  <artifactId>pitest-maven</artifactId>
  <version>1.4.2</version>
</plugin>
```

After adding the dependency , below command is used to generate the mutation score -

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

The Pit report is generated in HTML format at package and class level .For some of the projects, few 1-2 test cases are ignored.

3) INTEGRATION OF CLOC

To run CLOC , below dependencies need to be added to generate metric 5 and metric 6 data:

- Run below command in Terminal - `ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)" < /dev/null 2> /dev/null`
- After executing the first command,below command is executed - `brew install cloc` .In order to find the difference between the two versions, below command is executed-

```
cloc --diff  commons-fileupload-1.4-src  commons-fileupload-1.3-src
```

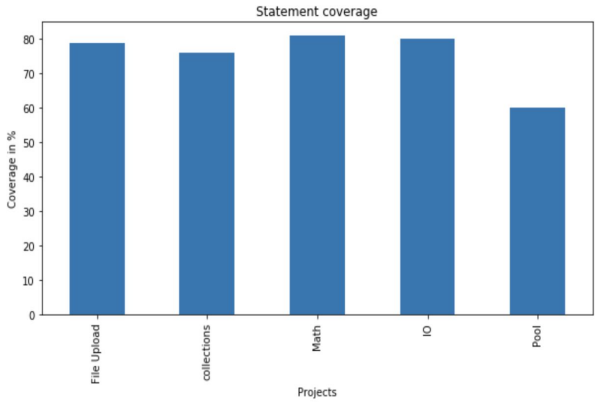
```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Java
 same                            0              0              0              0
 modified                        0              0              0              0
 added                          42            972           4187           3773
 removed                         0              0              0              0
```

- In order to calculate the line of code -

```
cloc --diff  commons-fileupload-1.4-src  commons-fileupload-1.3-src
```

C. DATA ANALYSIS

After data collection using the above mentioned tools, we created the excels in order to run the script. 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.



Graph 1: Statement coverage(in percentage) for the projects. Metric 1

```
import matplotlib.pyplot as plt

import pandas as pd
s = pd.Series([65,70,75,65,71], index=['File Upload','collections','Math','IO','Pool'])
s.plot.bar(figsize=(10,5))
plt.xlabel('Projects')
plt.ylabel('Mutation score in  %')
plt.title("Mutation Score ");
```

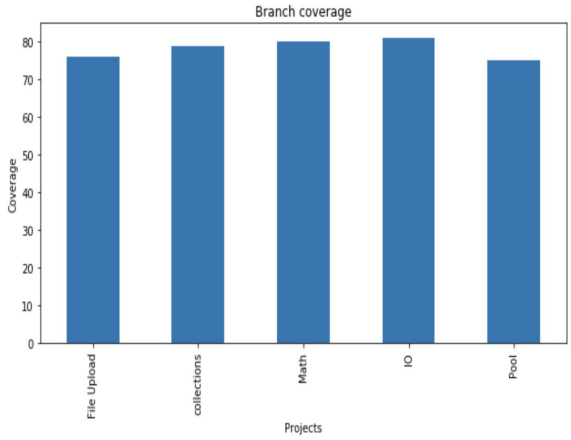
Snapshot 1 - Script used in Jupyter Notebook to analyze Data.

Version	Statement Coverage(%)	Branch Coverage(%)
FileUpload 1.2.2	74	71
FileUpload 1.3	79	76
FileUpload 1.3.1	79	76
FileUpload 1.3.2	79	76
FileUpload 1.3.3	77	74
FileUpload 1.4	80	76

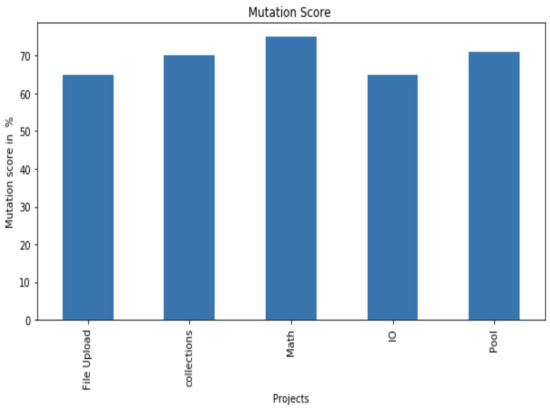
Version	Branch Coverage	McCabe Complexity
FileUpload 1.2.2	70	0.402439024390244
FileUpload 1.3	71	0.357446808510638
FileUpload 1.3.1	76	0.357446808510638
FileUpload 1.3.2	76	0.358811040339703
FileUpload 1.3.3	74	0.370762711864407
FileUpload 1.4	76	0.366876310272537

Version	Statement Coverage	McCabe Complexity
FileUpload 1.2.2	73	0.405472637
FileUpload 1.3	77	0.397590461
FileUpload 1.3.1	78	0.35881102
FileUpload 1.3.2	73	0.35881101
FileUpload 1.3.3	75	0.372590461
FileUpload 1.4	76	0.368990461

Table 1- Data Collection in File Upload



Graph 2: Branch coverage(in percentage) for the projects. Metric 2



Graph 3: Mutation score (in percentage) for the projects. Metric 3

Missed Complexity	Total Complexity	McCabe Complexity
165	410	0.402439024390244
168	470	0.357446808510638
168	470	0.357446808510638
169	471	0.358811040339703
175	472	0.370762711864407
175	477	0.366876310272537

Table 2 : McCabe Complexity for the projects. Metric 4

Version Comparisons	DLOC(lines added +modified)	Effort (hours)
v1.2.2 - v1.2.1	3773	115.73
v1.3 - v1.3.1	184	79.84
v1.3.1 - v1.3.2	44	78.44
v1.3.2 - v1.3.3	30	78.3
v1.3.3 - v1.4	772	85.72

Table 3 : Effort in person-hours for the File-Upload project. Metric 5

version	Defect count(Number of Bugs)	Size of release(Lines of code)	Defect density (per KLOC)
v1.2.2	9	6087	1.478561
v1.3	27	6558	4.11710887
v1.3.1	2	6544	0.30562347
v1.3.2	1	6576	0.152068
v1.3.3	1	7639	0.130907
v1.4	20	99131	0.20175323561

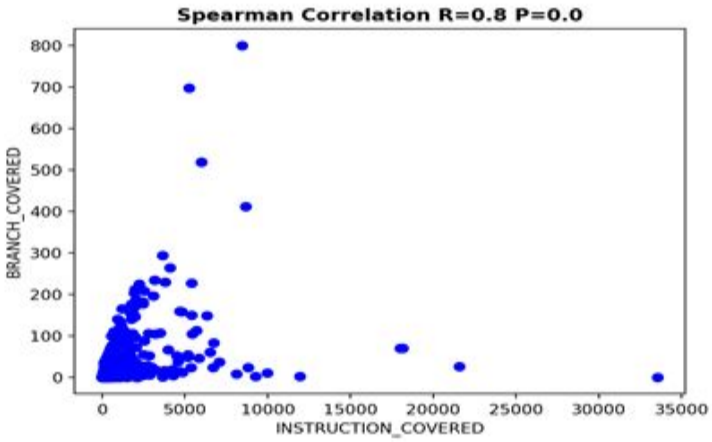
Table 4 : Defect Density for the File-Upload project. Metric 6

Results

a. Correlation between metric 1 and metric 2

The Spearman coefficient has been used for calculating the relationship between statement coverage and the branch coverage as it is normally distributed. Before calculating the coefficient, we assumed that the correlation would be highly positive. It turned out to be 0.8. As the statement coverage increases, there is an increase in branch coverage as well. The more statements covered means the more branches covered.

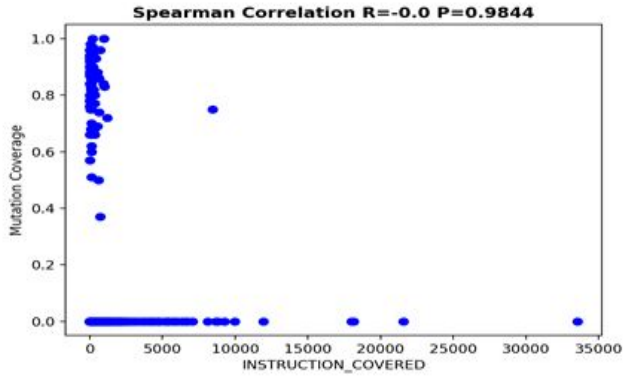
Version	Statement Coverage(%)	Branch Coverage(%)
FileUpload 1.2.2	74	71
FileUpload 1.3	79	76
FileUpload 1.3.1	79	76
FileUpload 1.3.2	79	76
FileUpload 1.3.3	77	74
FileUpload 1.4	80	76



b. Correlation between metric 1 and metric 3

The study assumed that there was a strong positive correlation between instructions covered and the mutation. However, it turned out to be a strong negative correlation. The study was performed at a project level.

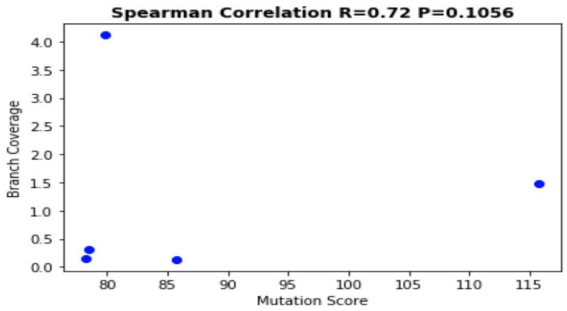
Version	Mutation Score	Statement Coverage
v1.4	66	76
v1.3.3	65	75
v1.3.2	68	73
v1.3.1	64	78
v1.3	64	77
v1.2.2	68	73



c. 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. 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 the project level as shown below

Version	Mutation Score(%)	Branch Coverage(%)
v1.4	65	76
v1.3.3	68	74
v1.3.2	68	76
v1.3.1	68	76
v1.2.2	64	71
v1.2.1	64	70



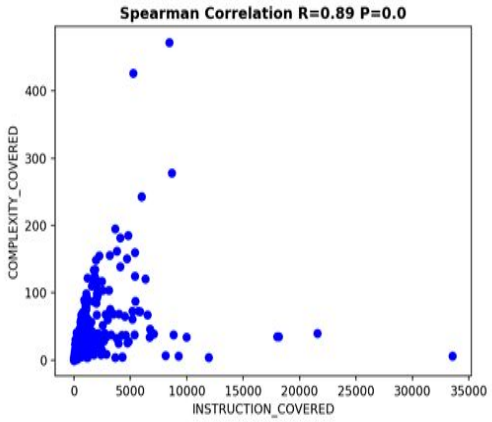
d. Correlation between metric 1 and metric 4

The correlation between cyclomatic complexity and the instruction covered was positive.
In the proposal, we assumed that metric 1 and metric4 would be positively correlated, and it was found that the increase in the number of instruction gives an increase in the complexity of a program.

It is clearly depicted from the picture below.

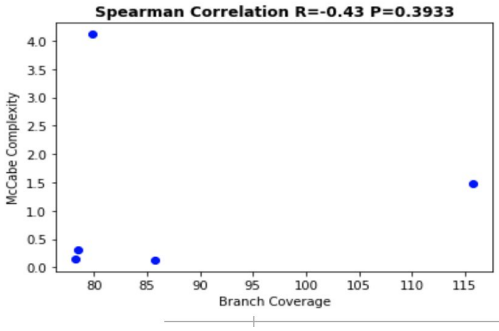
Version	Statement Coverage	McCabe Complexity
FileUpload 1.2.2	73	0.405472637
FileUpload 1.3	77	0.397590461
FileUpload 1.3.1	78	0.35881102
FileUpload 1.3.2	73	0.35881101
FileUpload 1.3.3	75	0.372590461
FileUpload 1.4	76	0.368990461

Version	Line Coverage
FileUpload 1.2.2	74
FileUpload 1.3	70
FileUpload 1.3.1	79
FileUpload 1.3.2	79
FileUpload 1.3.3	77
FileUpload 1.4	79



e. Correlation between metric 2 and metric 4

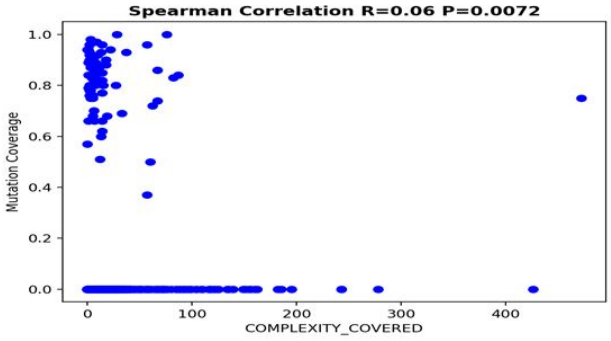
The correlation between branch covered and the complexity is again positive as per the proposal. It was found as the complexity increases, there was a gradual increase in the branch coverage.The initial assumptions were proven to be true with regards to the study conducted on these projects.



f. Correlation between metric 3 and metric 4

The correlation for the systems is turned out to be positive for 3 and 4 as well. As complexity increases, more mutants are been covered. However, for the initial state mutants had some value. The number of mutants was never zero and it kept on increasing as the complexity of a program increases.

It can be clearer from the picture below:

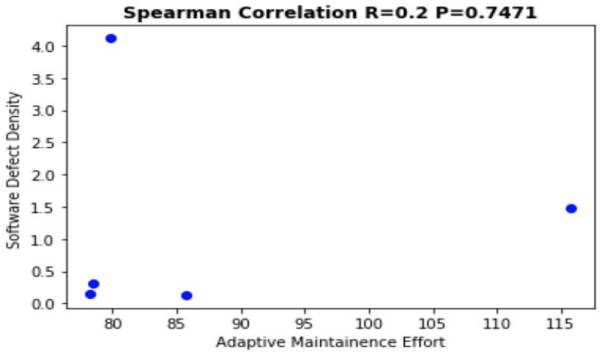


g. Correlation between metric 5 and metric 6

The correlation was found at a project level between various versions. It was found to be positive as depicted in the proposal. The number of commits in a code can produce more defects and errors in the code and vice versa.

version	Effort	Defect Density
v1.2.2	115.73	1.478561
v1.3.1	79.84	4.11710887
v1.3.2	78.44	0.30562347
v1.3.3	78.3	0.152068
v1.4	85.72	0.130907

Table 5 - Data Analysis for 5th and 6th Metric



It was again a mixed bag as some projects showed a medium positive correlation while other projects showed a negative correlation.

Script Used -

We have created a python script to generate the graphs and find correlation using Spearman coefficient between Variables. The snapshot of script is as below-

```
from scipy.stats import spearmanr
import pandas as pd

import matplotlib.pyplot as plt

df = pd.read_excel(r'C:\Users\nvirk\Desktop\Team-M\Projects\FileUpload\DataAnalysis\Correlation1_2_4.xlsx')

print(df)
for col in df.columns:
    print(col)
Adaptive_maintainence_effort = df['Adaptive_maintainence_effort'].tolist();
Defect_density = df['Defect_density'].tolist();

print(Adaptive_maintainence_effort)
print(Defect_density)
coefficient, value = spearmanr(Adaptive_maintainence_effort,Defect_density)
coefficient= round(coefficient, 2)
value= round(value, 4)
print("Spearman correlation coefficient=", coefficient, "\n", "2 sided p value=",value)
plt.xlabel("Defect_density ")
plt.ylabel("Adaptive_maintainence_effort")
plt.plot(X, Y,coefficient, value)

fig, ax1 = plt.subplots()
ax1.plot(X, Y, 'bo')
plt.xlabel("Defect_density ")
plt.ylabel("Adaptive_maintainence_effort")
plt.title(r'Spearman Correlation'+' R='+str(coefficient)+' P='+str(value),fontweight="bold")

plt.show()
```

Conclusion and future work

This report covers all six metrics. The expected correlation between all the metrics was the same except for the correlation between metrics 1 and 3

During analysis, it was observed that at a range of values which are generated after performing correlation is not enough due to the limitations of the data collected. This observation suggests that correlation values will improve if a variety of projects across diverse domains is considered.

References

[1] Laura Inozemtseva and Reid Holmes.2014. Coverage is not strongly correlated with test suite effectiveness. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). ACM, New York,NY, USA, 435-445. DOI:https://doi.org/10.1145/2568225.2568271

[2] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2014. The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects. In Proceedings of the 11th Working Conference on Mining Software Repositories(MSR 2014). ACM, New York, NY, USA, 192-201. DOI: http://dx.doi.org/10.1145/2597073.2597076

[3]DOI:https://doi.org/10.1145/2635868.2635929

[4] https://jmeter.apache.org/index.html

[5] http://commons.sourceforge.net/

[6]https://commons.apache.org/proper/commons

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

[8]https://archive.apache.org/dist/commons/fileupload/source/

[9] https://www.cs.odu.edu/~cs252/Book/branchcov.html

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

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

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

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