Cover Letter

Dear Professor Yang,

The followed table is the information of our team members:

Name	Student ID	Email
Tian Wang	40079289	wangtianjjl@163.com
Minxue Sun	40084491	348460778@qq.com
TianCheng Xu	40079681	tianchengxu1121@gmail.com
Qing Li	40082701	liqing51269@gmail.com

Following is a link to the replication package in GitHub:
https://github.com/minxue25/SOEN6611

Yours,

Team O

Final Report - Correlation between Different Metrics

Tian Wang
Concordia University
gina cody school of engineering and computer science
Montreal, Canada
wangtianjjl@163.com

Minxue Sun
Concordia University
gina cody school of engineering and computer science
Montreal, Canada
348460778@qq.com

Tiancheng Xu
Concordia University
gina cody school of engineering and computer science
Montreal, Canada
tianchengxu1121@gmail.com

Qing Li
Concordia University
gina cody school of engineering and computer science
Montreal, Canada
liqing51269@gmail.com

***Abstract*—Correlation relationship analysis between software measurement metrics.**

In this report, we select six metrics to do the correlation analysis: statement coverage, branch coverage, mutation score, McCabe complexity, LOC difference and backlog management index. We select Apache commons collections, Spotify maven docker plugin and JFreeChart as project sample. We find relations between these software metrics by analyzing experimental data in different levels. We used Pearson Correlation to find the correlation between metrics.

In conclusion, branch coverage, statement coverage and metric mutation score is positive and very strong; branch coverage, statement coverage and McCabe complexity is negative and the strength of the association is good but not very strong; branch coverage and statement coverage and metric 6 were very small; backlog management index and change proneness were positively correlated and moderately strong.

Keywords—software measurement, metric, correlation analysis

I. Introduction

A software has many different metrics. The purpose of software testing is to check whether the software meets the requirements. Software testing can improve the quality of software. However, it is also necessary to measure software testing to determine the effectiveness of software testing. Therefore, software measurement becomes important. To study the relationship and correlation between all attributes and factors that may have a positive or negative impact on the developed software product. In the report, we analyzed the correlation between different indicators. Based on development experience, we chose three open source projects to measure based on six software metrics.

II. Related Work

Software metrics are often supposed to give valuable information for the development of software. Some researchers already analyze correlation metrics with C and C++ programs. [5]

According to researcher's analysis, we got followed conclusions:

(1) there is a very strong correlation between Lines of Code and Halstead Volume;

(2) there is an even stronger correlation between Lines of Code and McCabe's Cyclomatic Complexity;

(3) none of the internal software metrics makes it possible to discern correct programs from incorrect ones;

(4) given a specification, there is no correlation between any of the internal software metrics and the software dependability metrics.[5]

However, most of these correlations are relate to complexity. We are going to research correlation metrics in other aspects.

III. Project Description

We chose three Java open source projects and two of them have more than 100K of LOC. The main criteria for our selection of projects are:

- A Maven project
- Meet the requirements of LOC
- Has an issue tracking system.

Project 1: Apache Commons Collections4

https://commons.apache.org/proper/commons-collections/index.html

The Java project was a major addition in JDK 1.2. It added many powerful data structures that accelerate the development of the most significant Java applications. Since that time it has become the recognized standard for collection handling in Java.

For correlation analysis in different versions. We chose this project with version 3.2.2 to 4.4. The LOC of collections is 132K. It is a Maven project and it also has a complete issue-tracking system.

Project 2: Maven Docker Plugin

https://github.com/spotify/docker-maven-plugin

This is a maven project with the LOC of 6.39K. It has several released version and issue tracking system on GitHub. This plugin was the initial Maven plugin used at Spotify for building Docker images out of Java services. It was initially created in 2014 when we first began experimenting with Docker. This plugin is capable of generating a Dockerfile for you based on configuration in the pom.xml file for things like the FROM image, resources to add with ADD/COPY, etc.

Project 3: JFreeChart

https://github.com/jfree/jfreechart

JFreeChart is a free Java chart library that makes it easy for developers to display professional quality charts in their applications.

It is a Maven project and the LOC over 167K. It has an issue-tracking system. we analysis this project's data from version 1.0.18 to current master branch.

IV. METRICS DESCRIPTION

In order to better measure selected projects, six different software measurement metrics are used. These six metrics belong to different aspects of software measurement. The details will be given as follow:

Metric 1: Statement Coverage

Statement Coverage technique is used to calculate and measure the number of statements in the source code which can be executed given requirements.[1]

Statement Coverage technique can check what the source code is expected to do and what it should not. It can also check the quality of the code and the flow of different paths in the program. The main drawback of this technique is that we cannot test the false condition in it.

Statement Coverage Formula:

$$Statement\ Coverage = \frac{Number\ of\ executed\ statements}{Total\ number\ of\ statements} \times 100\%$$

Metric 2: Branch Coverage

Branch coverage aims 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. It helps us to find out which sections of code don't have any branches.[1]

Branch Coverage Formula:

$$Branch\ Coverage = \frac{Number\ of\ executed\ branches}{Total\ number\ of\ branches} \times 100\%$$

Metric 3: Mutation Testing

Mutation testing has traditionally been used to evaluate the effectiveness of test suites. Mutation testing involves the creation of many versions of a program each with a single syntactic fault. A test suite is evaluated against these program versions (i.e. mutants) in order to determine the percentage of mutants a test suite is able to identify (i.e. mutation score).[2]

Mutation Score Formula:

$$Mutation\ Score = \frac{Killed\ Mutants}{Total\ number\ of\ Mutants} \times 100\%$$

Metric 4: McCabe(Cyclomatic Complexity)

McCabe is one of metrics that measure structure complexity of a program. It was also an indicator of the testability and maintainability of a program.[3] For calculating McCabe complexity, followed elements should be counted:

Formula 1:

$$Cyclomatic\ Complexity = E - N + 2P$$

- E = the number of edges in CFG
- N = the number of nodes in CFG
- P = the number of connected components in CFG

Formula 2:

$$Cyclomatic\ Complexity = D + 1$$

- D = the number of control predicate (or decision) statements
- For a single method or function, P is equal to 1

Metric 5: LOC difference (DLOC)

The number of lines of codes changed edited, added or deleted (DLOC) were the most effective for predicting adaptive maintenance effort.

DLOC Formula:

$$DLOC = The\ number\ of\ lines\ of\ codes\ changed\ edited\ or\ added\ or\ deleted$$

Metric 6: Backlog Management Index (BMI)

Backlog Management Index (BMI) is an important metric to manage the backlog of open, unresolved problems in a Software Project.[4]

BMI Formula:

$$BMI = \frac{Number\ of\ problems\ closed\ during\ the\ month}{Number\ of\ problems\ arrives\ during\ the\ month} \times 100\%$$

V. STEPS FOR COLLECTING THE DATA

Our data collecting work can be totally divided into 4 steps:

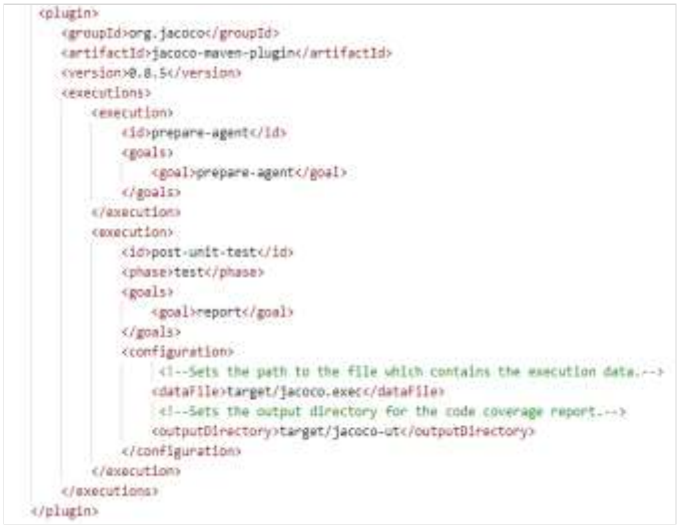
- Step 1: Adding Jacoco plugin.
- Step 2: Adding pit test plugin.
- Step 3: Calculating DLOC for different versions
- Step 4: Calculating BMI for different versions.

Step 1: Adding Jacoco plugin

In order to collect the data for statements coverage, branch coverage as well as Cyclomatic complexity, we add Jacoco plugin into each project's pom.xml file (Fig.1). An example of Jacoco report is shown in Figure 2.

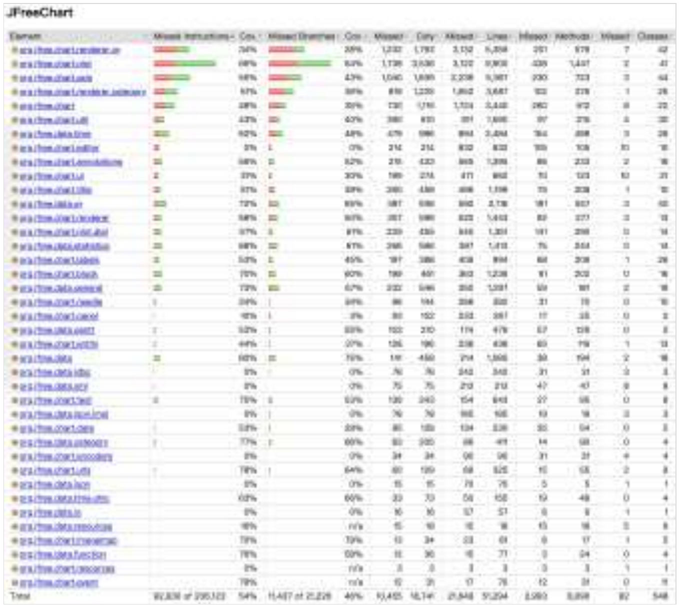
We need metrics 1,2 at both class level and version level due to the reason that we need to:

- Analyze correlation between metrics 1&2&3, 1&2 and 4 on class level.
- Analyze correlation between metrics 1 and 6, metrics 2 and 6, metrics 5 and 6 on version level.



```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.5</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>
```

Fig.1. Jacoco configuration (JFreeChart)



JFreeChart

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

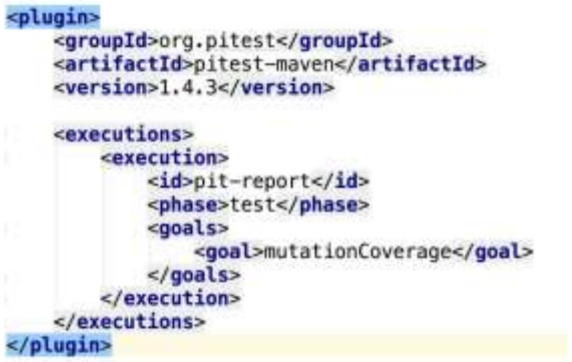
Fig.2. Jacoco report (JFreeChart)

Step 2: Adding pit plugin

pitest-maven plugin is used to get the mutation report for each project. We need to add pitest configuration into pom.xml file of each project. Here we show the code needed to add the configuration (Fig.3) and an example of pitest report. (Fig.4)

We need metrics 4 at both class level and version level due to the reason that we need to:

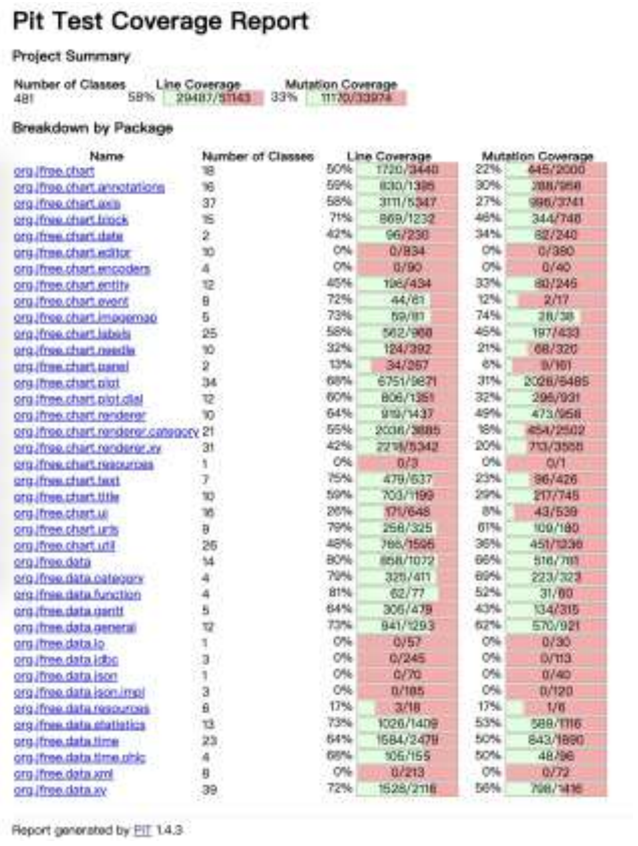
- Analyze correlation between metrics 1&2 and 3, 1&2 and 4 at class level
- Analyze correlation between metrics 1 and 6, metrics 2 and 6, metrics 5 and 6 at version level



```
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>1.4.3</version>

    <executions>
        <execution>
            <id>pit-report</id>
            <phase>test</phase>
            <goals>
                <goal>mutationCoverage</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Fig.3. Pitest configuration(JFreeChart)



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
481	58%	29487/51143	33%	11170/33974

Breakdown by Package

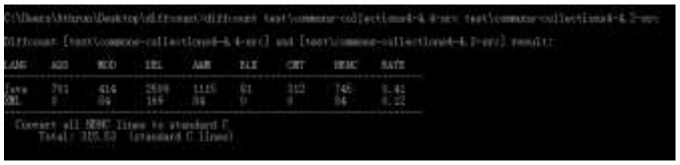
Name	Number of Classes	Line Coverage		Mutation Coverage	
org.jfree.chart	18	50%	1720/3440	22%	445/2000
org.jfree.chart.annotations	16	59%	830/1395	30%	288/958
org.jfree.chart.axis	37	58%	3111/5347	27%	996/3741
org.jfree.chart.block	15	71%	869/1232	46%	344/748
org.jfree.chart.date	2	42%	96/230	34%	82/240
org.jfree.chart.editor	10	0%	0/834	0%	0/380
org.jfree.chart.encoders	4	0%	0/90	0%	0/40
org.jfree.chart.entity	12	45%	196/434	33%	80/245
org.jfree.chart.event	8	72%	44/61	12%	2/17
org.jfree.chart.imagemap	5	73%	59/81	74%	28/38
org.jfree.chart.labels	25	58%	562/968	45%	197/433
org.jfree.chart.needle	10	32%	124/392	21%	68/320
org.jfree.chart.panel	2	13%	34/267	6%	9/161
org.jfree.chart.plot	34	68%	6751/9871	31%	2026/6485
org.jfree.chart.plot.dial	12	60%	806/1351	32%	296/931
org.jfree.chart.renderer	10	64%	919/1437	49%	473/958
org.jfree.chart.renderer.category	21	55%	2036/3685	18%	454/2502
org.jfree.chart.renderer.xy	31	42%	2218/5342	20%	713/3565
org.jfree.chart.resources	1	0%	0/3	0%	0/1
org.jfree.chart.text	7	75%	479/637	23%	96/426
org.jfree.chart.title	10	59%	703/1199	29%	217/745
org.jfree.chart.ui	16	26%	171/648	8%	43/539
org.jfree.chart.urls	8	79%	258/325	61%	109/180
org.jfree.chart.util	26	48%	766/1596	36%	451/1236
org.jfree.data	14	80%	858/1072	66%	516/781
org.jfree.data.category	4	79%	325/411	69%	223/323
org.jfree.data.function	4	81%	62/77	52%	31/60
org.jfree.data.gantt	5	64%	306/479	43%	134/315
org.jfree.data.general	12	73%	941/1293	62%	570/921
org.jfree.data.io	1	0%	0/57	0%	0/30
org.jfree.data.jdbc	3	0%	0/245	0%	0/113
org.jfree.data.json	1	0%	0/70	0%	0/40
org.jfree.data.json.impl	3	0%	0/185	0%	0/120
org.jfree.data.resources	6	17%	3/18	17%	1/6
org.jfree.data.statistics	13	73%	1026/1409	53%	588/1116
org.jfree.data.time	23	64%	1584/2479	50%	843/1690
org.jfree.data.time.ohlc	4	68%	105/155	50%	48/96
org.jfree.data.xml	8	0%	0/213	0%	0/72
org.jfree.data.xy	39	72%	1528/2116	56%	798/1416

Report generated by PIT 1.4.3

Fig.4. Pitest Report (JFreeChart)

Step 3: Calculating DLOC

We used a small program called diffcount to get the DLOC between versions. Before running the program, we will manually delete the non-Java files and the test directory to obtain accurate data. If the file directory structure is different, we will change the structure to make sure files are the same directory.



```
C:\Users\Athrun\Desktop\diffcount\diffcount test\commons-collections4-4.4-src test\commons-collections4-4.2-src
Diffcount [test\commons-collections4-4.4-src] and [test\commons-collections4-4.2-src] result:
LANG    ADD     MOD     DEL     A&M     BLK     CMT     NBNC    RATE
-----------------------------------------------------------------------
Java    751     414     2599    1115    61      312     745     1.41
XML     0       84      155     84      0       0       84      0.22
-----------------------------------------------------------------------
  Convert all NBNC lines to standard C
      Total: 315.53  (standard C lines)
```

Fig.5. DLOC of Apache Common Collections

Step 4: Calculating BMI

We review the issues of the project GitHub repository. We calculate BMI by tracking the number of issues arrived and how many of them closed in the specific version lifetime. Some projects have issue records on JIRA. We collect the number of closed issues per month by dividing the number of issues raised and get an average.

VI. Steps for Analyzing the Data

We use Pearson correlation coefficient to analyze correlation.

The Steps of data analysis are as follows:

- We collect data from jacoco reports and pitest reports manually, then combine them in the same excel according to same java class.
- Determining which two metrics are used for correlation comparative analysis and determining which level of data they are (e.g. class level). Extracting the metric data of specific project from the collected data.
- The whole data analysis process is implemented by using Python. First, we extract the collected metric data from excel form by using pandas library. Then, calculating the Pearson correlation coefficient between different metrics and using Matplotlib library generating the scatter diagrams of the data to illustrate the correlation of the date. Comparing the results of the specific metric correlation coefficients of the five projects and conclude the most general conclusion.

VII. Result

A. Correlation between Metric 1 & 2 and 3

TABLE I. *R(Pearson)* between metric 1 & 3

Project	**Sets of data (Class level)**	***R(Pearson)* of metric 1&3**
Apache commons collections	267	0.4629
JFreeChart	481	0.7996
Maven Docker Plugin	10	0.7144

TABLE II. *R(Pearson)* between metric 2 & 3

Project	**Sets of data (Class level)**	***R(Pearson)* of metric 2&3**
Apache commons collections	267	0.3714
JFreeChart	481	0.7548
Maven Docker Plugin	10	0.7230

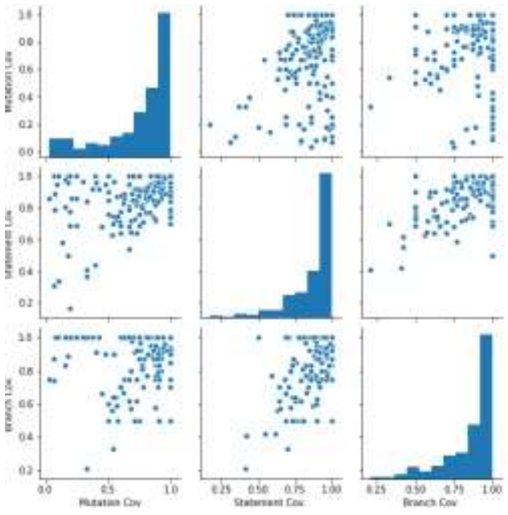


Fig.6. Data distribution diagram of class level between Metric 1&2 and 3 in Apache commons collections

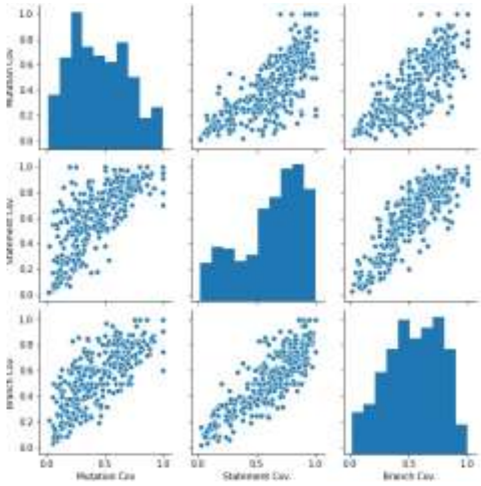


Fig.7. Data distribution diagram of class level between Metric 1&2 and 3 in JFreeChart

It can be seen from Figure 6 and 7, as well as table Ⅰ and table Ⅱ above that the R(Pearson) of the three groups is strong and the direction of the correlation is positive.

B. *Correlation between Metric 1&2 and Metric4*

TABLE III. *R(PEARSON)*BETWEEN METRIC 1 & 4

Project	**Sets of data (Class level)**	***R(Pearson)*of metric 1&4**
Apache commons collections	267	-0.0155
JFreeChart	481	-0.0613
Maven Docker Plugin	10	-0.0169

TABLE IV. *R(PEARSON)*BETWEEN METRIC 2 & 4

Project	**Sets of data (Class level)**	***R(Pearson)*of metric 2&4**
Apache commons collections	267	-0.1258
JFreeChart	481	-0.0981
Maven Docker Plugin	10	-0.1686

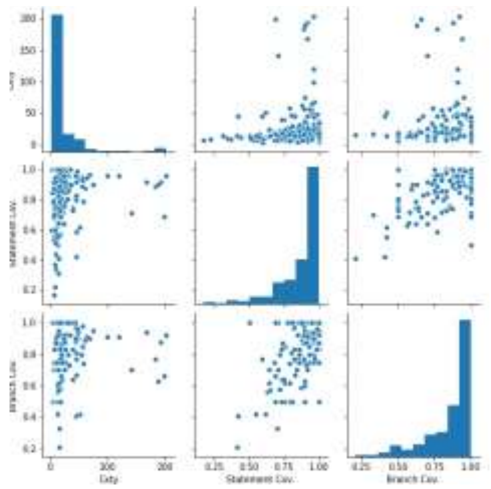


Fig.8. Data distribution diagram of class level between Metric 1&2 and 4 in Apache commons collections

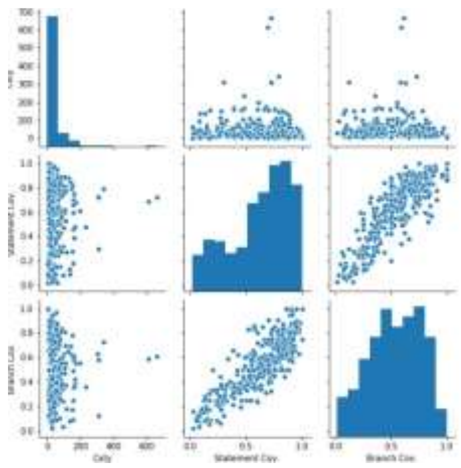


Fig.9. Data distribution diagram of class level between Metric 1&2 and 4 in JFreeChart

As can be seen from Figure 8, 9 and the Pearson correlation coefficients of metric 1&4, 2&4 of the three projects in Table Ⅲ and Table Ⅳ above, we can conclude from these two tables that the correlation between metric 1&2 and 4 is negative and the strength of the association is good but not very strong.

C. *Correlation between Metric 1&2 and Metric 6*

TABLE V. *R(PEARSON)*BETWEEN METRIC 1 AND METRIC 6

Project	**Sets of data (Version level)**	***R(Pearson)*of metric 1&6**
Apache commons collections	267	0.9999
JFreeChart	481	0.8722
Maven Docker Plugin	10	0.9713

TABLE VI. *R(PEARSON)*FOR METRIC 2 AND METRIC 6

Project	**Sets of data (Version level)**	***R(Pearson)*of metric 2&6**
Apache commons collections	267	0.9863
JFreeChart	481	0.9921
Maven Docker Plugin	10	0.9169

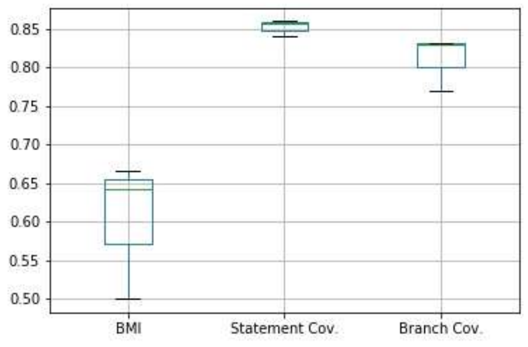


Fig.10. Boxplot of rs of metric1&6 and metric 2&6 in Apache commons collections

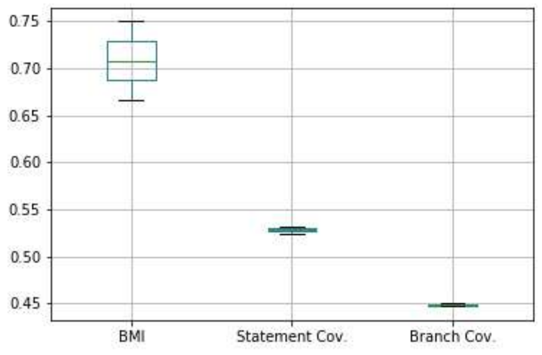


Fig.11. Boxplot of rs of metric1&6 and metric 2&6 in JFreeChart

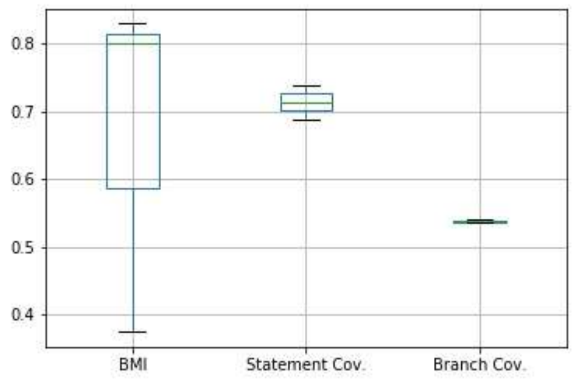


Fig.12. Boxplot of rs of metric1&6 and metric 2&6 in Maven Docker Plugin

The Pearson correlation coefficients *R(Pearson)* for metric 1&2 and metric 6 are shown in Table Ⅴ and Table Ⅵ. The absolute *R(Pearson)* of all three projects are extremely high that almost around 1, which shows a strongly positive correlation between metric 1&2 and metric 6. Consequently, we infer that the correlation between metric 6 and metric 1&2 is strong and positive.

D. Correlation between Metric 5 and Metric 6

The Pearson correlation coefficient R(Pearson)is calculated from the above 8 sets of data, and the value of R(Pearson) was -0.2344, so it shows that the negative correlation between metric 5 and metric 6 is medium.

TABLE VII. METRIC 5 AND METRIC 6 DATA FROM DIFFERENT VERSIONS OF 3 PROJECTS

Project (Version-Version)	**Metric5 DLOC**	**Metric 6 BMI**
Apache commons collections 3.2-4.0	58988	0.5
Apache commons collections 4.0-4.1	7218	0.642
Apache commons collections 4.1-4.3	3624	0.6667
JFreechart 0.19-1.5.0	65839	0.6667
JFreechart 0.19-master	14275	0.75
Docker 1.1.0-1.1.1	2	0.375
Docker 1.1.0-1.2.0	21	0.83
Docker 1.2.0-1.2.1	4	0.8

VIII. CONCLUSIONS

According to the graph shown above, we can conclude the relationship between different software measurement metrics.

For metric 1&2 and metric 3, it shows that the correlation is positive and very strong. We can conclude that suites with higher statement or branch coverage can show high mutation score. This conclusion is consistent with the rationale that test suites with higher coverage can show better test suite effectiveness.

For metric 1&2 and 4, it shows that the correlation is negative and the strength of the association is good but not very strong. We can conclude that classes with higher Cyclomatic Complexity show lower statement/branch coverage. This conclusion is consistent with the rationale that classes with higher complexity are less likely to have high coverage test suites.

For metric 1&2 and metric 6, it describes that the Pearson correlation coefficients for metric 1&2 and metric 6 were very small, even not greater than 0.1 in absolute value. Therefore, we consider that metric 1&2 and metric 6 are almost uncorrelated. We think that there is no correlation between statement/branch coverage and change proneness.

For metric 5 and metric 6, it shows that the Pearson correlation coefficients between them were negatively correlated and moderately strong. We conclude that on the project-level, project with lower LOC difference might show higher Backlog Management Index.

REFERENCES

[1] Code Coverage Tutorial: Branch, Statement, Decision, FSM [Online]. Available: https://www.guru99.com/code-coverage.html#6

[2] H. Felbinger, F. Wotawa and M. Nica, "Mutation Score, Coverage, Model Inference: Quality Assessment for T-Way Combinatorial Test-Suites," 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) , Tokyo, 2017, pp. 171-180.

[3] D. I. De Silva, N. Kodagoda and H. Perera, "Applicability of three complexity metrics," International Conference on Advances in ICT for Emerging Regions (ICTer2012) , Colombo, 2012, pp. 82-88.

[4] S. H. Kan, Ed., Metrics and Models in Software Quality Engineering. (2nd ed.) America: Addison-Wesley Professional., 2002.

[5] d. M. van and M. A. Revilla, "Correlations between internal software metrics and software dependability in a large population of small C/C++ programs," in 2007 18th IEEE International Symposium on Software Reliability Engineering, 2007, Available: http://dx.doi.org/10.1109/ISSRE.2007.12. DOI: 10.1109/ISSRE.2007.12.