# **Software Measurement – Team N**

# Dear Professor Jinqiu Yang,

Our team would like to submit the report based on the following content selected project links and descriptions, metric descriptions, steps for collecting the data, steps for analyzing the data, describing the results (e.g., descriptive summaries for collected metrics, results of correlation analysis etc.), related work.

In this paper, we did correlation analysis among followed six metrics: branch coverage, statement coverage, mutation score, McCabe complexity, backlog management index and Maintainability index.

The followed table is the information of our team members:

|  |  |  |
| --- | --- | --- |
| **Name** | **Student Number** | **E-mail** |
| Neha Sarang Dighe | 40105186 | [neha.dighe18@gmail.com](mailto:neha.dighe18@gmail.com) |
| Shashank Rao | 40104247 | shashankrao1994@gmail.com |
| Samir Anghan | 40040308 | samir.anghan@gmail.com |
| Aayush Lamichhane | 40091493 | ayus.lami7@gmail.com |
| Aamrean Singh Kaler | 40080820 | aamrean23@gmail.com |

Following is a link to the replication package in GitHub:

<https://github.com/samir-anghan/SoftwareMeasurement>

# Thank you very much for your attention.

Sincerely, Team N

***Abstract*- Correlation is a statistical measure that indicates the extent to which two or more variables fluctuate together. A positive correlation indicates the extent to which those variables increase or decrease in parallel; a negative correlation indicates the extent to which one variable increases as the other decreases. In this paper, we focus on several common internal metrics: statement coverage, branch coverage, mutation score, McCabe complexity, backlog management index and Maintainability index. We find relations between these software metrics by analyzing experimental data.**

**We select five open source projects to get data:**

**Apache commons Lang, Apache commons configuration, Apache commons Codec, Apache commons collections and JfreeChart. To calculate correlation, we used Spearman Correlation.**

**In conclusion, branch coverage, statement coverage and McCabe complexity is negative and the strength of the association is good but not very strong.**

**Branch coverage, statement coverage and metric mutation score is positive and very strong.**

**Branch coverage and statement coverage and metric 6 negative and small correlation.**

**Backlog management index and Maintainability Index were negative and small correlation.**

***Keywords—metric, correlation, software measurement***

1. INTRODUCTION

The increasing up-trend of software size brings about challenges related to release planning and maintainability[1]. Foreseeing the growth of software metrics can assist in taking proactive decisions regarding different areas where software metrics play vital roles. For example, source code metrics are used to automatically calculate technical debt related to code quality which may indicate how maintainable a software is. Thus, predicting such metrics can give us an indication of technical debt in the future releases of software. Objective: Estimation or prediction of software metrics can be performed more meaningfully if the relationships between different domains of metrics and relationships between the metrics and different domains are well understood.

In this paper, we tend to analyze the correlation among different metrics. According to development experience, we selected five open source projects and six software measurement metrics.

In next sections, projects description and metrics description were provided. Then we provide the steps of collecting and analyzing data. In last section, we show descriptive summaries for collected metrics and results of correlation analysis.

1. 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 [8].

We collected the information from various plugins which we embedded with [IntelliJ IDEA:[2]  It is developed by JetBrains, and is available as an Apache 2 Licensed community edition, and in a proprietary commercial edition. Both can be used for commercial development.](https://www.jetbrains.com/idea/)

1. Choosing projects
2. Build the projects
3. Configuration of JACOCO plugin to the project - Metrics 1 & Metrics 2 - Statement and Branch Coverage
4. Configuration of PI-test plugin to the project

Metrics 3 – Mutation Score( Mutation Score = (Killed Mutants / Total number of Mutants) \* 100 )

1. Configuration of Metrics Reloaded Plugin to the project - Metrics 4 – Cyclomatic Complexity (G = E-N+2P), Halstead Volume(V), WMC(Weighted Method Complexity)
2. Computed Metric 5 (Maintainability Index) using Metric 4(Cyclomatic Complexity) and Halstead Volume

Formula: (MI = 171 - 5.2 \* ln(V) - 0.23 \* (G) - 16.2 \* ln (LOC) )

1. Computed Metric 6 using Issue tracking system- Apache JIRA for apache projects, Github issue tracker for other projects.

Hence with the above metrics we have computed the complexity and the correlation between different matrices.

1. PROJECTS DESCRIPTION

To make our experiment more convincing, we're choosing 5 java open source projects, in which 3 of them are greater than 100K LOC. Moreover, for each project, we choosing 3-4 different versions to collect the data. So we are able to collect the difference during the version evolution period. In addition, all of the projects that we choose has an issue-tracking system, which we used for collecting the data for maintenance relevant metrics.

*Project 1: Apache commons IO*

<https://commons.apache.org/proper/commons-io/>

Commons IO is a library of utilities to assist with developing IO functionality.

There are six main areas included:[Utility classes](https://commons.apache.org/proper/commons-io/javadocs/api-release/index.html?org/apache/commons/io/package-summary.html) - with static methods to perform common tasks. [Input](https://commons.apache.org/proper/commons-io/javadocs/api-release/index.html?org/apache/commons/io/input/package-summary.html) - useful Input Stream and Reader implementations .[Output](https://commons.apache.org/proper/commons-io/javadocs/api-release/index.html?org/apache/commons/io/output/package-summary.html) - useful Output Stream and Writer implementations. [Filters](https://commons.apache.org/proper/commons-io/javadocs/api-release/index.html?org/apache/commons/io/filefilter/package-summary.html) - various implementations of file filters. [Comparators](https://commons.apache.org/proper/commons-io/javadocs/api-release/index.html?org/apache/commons/io/comparator/package-summary.html) - various implementations of java.util.Comparator for files. [File Monitor](https://commons.apache.org/proper/commons-io/javadocs/api-release/index.html?org/apache/commons/io/monitor/package-summary.html) - a component for monitoring file system *events*

*Project 2: Apache commons configuration*

<https://commons.apache.org/proper/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.

Apache commons configuration is a project that provides a library to use configuration/preferences of various sources and formats [5]. It also has an issue tracker (given on the project website), hence it opens up the possibility of using the project for defect related metrics (metrics 5). We’re collecting data using versions from 2.1.1 to 2.4.

The project is built by maven and consists of many documents to track the problems and help solve them during analysis. It also contains the test suites, which are used for metrics 1,2&4. And 1,2&3.

*Project 3: Apache commons Codec*

<https://commons.apache.org/proper/commons-codec/>

Apache Commons Codec (TM) software provides implementations of common encoders and decoders such as Base64, Hex, Phonetic and URLs.[4]

The size of collections is a 162.3K LOC which is the ideal size of our experiments

Commons Codec is a perfect project for us to collect the data from. The whole project is built by Maven, and it contains a lot of developer test cases. It is very convenience for us to collect the Jacoco and Pitest report since the only thing we need to do is the configuration. It also contains an issue tracking system and a lot of subversions. We are using versions from 1.8 to 1.12 for the experiment.

*Project 4: Apache commons collections*

<https://commons.apache.org/proper/commons-collections/>

The Java Collections Framework 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.

We are using from version 4.0 to 4.3 for our experiments in this project. The size of collections is a 132K LOC which is the ideal size of our experiments. Just like what other project does, it contains a continues issue-tracking system and build in Maven, which makes our data collecting work very convenient.

Apache Commons Collections is an open-source project that provides powerful data structures that assist in the development of most significant Java applications [3]. This project has around 132,000 lines of code and is almost completely written in Java. The Apache Commons’ website provides all the necessary details regarding all its previous and current versions, which will be useful while computing the evolution-related metrics. Since, the project is built with Maven, we can use the JaCoCo plugin to compute metrics 1 and 2 as well.

*Project 5: JfreeChart*

<http://www.jfree.org/jfreechart/>

JFreeChart is a free 100% Java chart library that makes it easy for developers to display professional quality charts in their applications.

JFreeChart is a maven project and in the size of 167K LOC, and a continuous issue-tracking system. However it doesn’t have too many versions we can collect the data from, we only analysis this project’s data from version 1.0.19 – 1.5.0.

1. METRICS DESCRIPTION

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

*Metric 1: Statement Coverage*

Statement coverage is a white box testing technique, which involves the execution of all the statements at least once in the source code. It is a metric, which is used to calculate and measure the number of statements in the source code which have been executed. Using this technique we can check what the source code is expected to do and what it should not. It can also be used to check the quality of the code and the flow of different paths in the program. Statement coverage count is how many statements are executed at least once during the test and thereby the more coverage percent it shows, the more opportunity to find the existing bug .

In white box testing, concentration of the tester is on the working of internal source code and flow chart or flow graph of the code.[7]

Generally, in the internal source code, there is a wide variety of elements like operators, methods, arrays, looping, control statements, exception handlers,. Based on the input given to the program, some code statements are executed and some may not be executed. The goal of statement coverage technique is to cover all the possible executing statements and path lines in the code.

*Statement coverage = No of statements Executed/Total no of statements in the source code \* 100*

*Metric 2: Branch Coverage*

Though statement coverage is essential, it also has some defects. For example, statement coverage only consider the executed statements and ignore the combinations of branches.

Branch coverage is a testing method, which aims to ensure that each one of the possible branch from each decision point is executed at least once and thereby ensuring that all reachable code is executed.

That is, every branch taken each way, true and false. It helps in validating all the branches in the code making sure that no branch leads to abnormal behavior of the application

*Branch Testing = (Number of decisions outcomes tested / Total Number of decision Outcomes) x 100%*

As a result, branch coverage was chosen. Branch coverage is how many branches from each decision point is executed at least once thereby the more coverage percent it shows, the more opportunity to find the existing bug .[8]

*Metric 3: Mutation Score*

The number of mutants depends on the definition of mutation operators and the syntax/structure of the software.

The number of mutants depends on the definition of mutation operators and the syntax/structure of the software 100% mutation score means killing all mutants (or a random sample).

*Score = (Killed Mutants / Total number of Mutants) \* 100*

Two assumptions: Competent programmer assumption: Developers write programs that are nearly correct with some small syntactic errors.

Coupling effect assumption: Test cases that distinguish all programs differing from a correct one by only simple errors is so sensitive that they also implicitly distinguish more complex errors.

*Metric 4: McCabe Metric (Cyclomatic Complexity)*

McCabe is a software metric used to calculate the complexity of a software. It is the measure of the number of independent executable paths within the code. It helps in determining the number of test cases which are required to get complete branch coverage [7]. McCabe makes use of Control Flow graph to calculate the Cyclomatic Complexity of the source code.

Cyclomatic Complexity is calculated as below,

**Cyclomatic Complexity = E – N + 2P**, where

E = the number of edges in CFG

N = the number of nodes in CFG

P = the number of connected components in CFG

D = is the number of control predicate (or decision) statements

For a single method or function, P is equal to 1

**Cyclomatic Complexity = E – N + 2**

Cyclomatic complexity can also be calculated by the below formula

**Cyclomatic Complexity = D + 1**, where

D = is the number of control predicate (or decision) statement

*Metric 5: Maintainability index*

The Maintainability Index(MI) is a combination of several metrics, including Cyclomatic Complexity and Average Lines of Code, as well the Halstead Volume[4][9].

The calculation method as follows: First, we need to measure the following metrics from the source code:

V = Halstead Volume

G = Cyclomatic Complexity

LOC = count of source Lines of Code (SLOC)

CM = percent of lines of Comment (optional)

From these measurements the MI can be calculated as:

*MI = 171 - 5.2 \* ln(V) - 0.23 \* (G) - 16.2 \* ln (LOC)*

The maintainability index is a value between 1 and 100. It represents the relative ease of maintaining the code. A high value means better maintainability [5].

*Metric 6: Fix Backlog and Backlog Management Index*

Backlog Management Index (BMI) uses Fix Backlog which is a workload statement for software maintenance. It is related to both the rate of defect arrivals and the rate at which fixes for reported problems become available. It is a simple count of reported problems that remain at the end of each month or each week. BMI is a metric to manage the backlog of open, unresolved, problems. As a ratio of the number of closed, or solved, problems to the number of problem arrivals during the month.[9]

*BMI = Number of problems closed during the month X 100%*

*Number of problem arrivals during the month*

1. STEPS TO COLLECTING THEDATA

We divide the data collecting process in to following steps:

* Step 1: Choosing the open source projects (three open-sourcesystems having at least 100K SLOC)
* Step 2: Building the project and running the test cases.
* Step 3: Configuring JaCoCo- free code coverage library for Java.
* Step 4: Configuring PIT for Mutation Testing.
* Step 5:Configuring MetricsReloaded plugin in IntelliJ Idea IDE for static code analysis.
* Step 6: Manual work for data cleaning and co-relations calculation for metrics using MS Excel.

*Step1: Choosing the open source projects*

We spent lot of time on finding and choosing the open source projects in order to satisfy the requirements for all of the metrics. We make sure to meet the following criteria:

* An open source project must have majority of Java codes and should have enough SLOCs.
* It must have active and well managed issue tracking system.
* It should have at-least 5 to 8 released versions or sub-versions.
* It must have Junit test suits in order to collect data for mutation testing.
* It should have at-least some documentations that can guide us to build the project.

Afterspending a lot time, we chose some projects that meets the above requirements. Majority of the chosen projects are Apache projects since they have lot of projects that meet our requirements.

*Step 2: Building the projects and running the test cases.*

After choosing the projects, we started building them. We realize that it isn’t the easy task to build them since the projects are using different build tools such as Maven, Ant, Gradle. It had some build errors due to various problems in each of the projects. Major problems includes missing dependencies due to versioning issues and using specific JDKs.

The next step was to run the unit test cases with some test suits because we wanted to make sure that it has enough test cases so that we can calculate the test coverages and perform the mutation testing. After applying the team work, our efforts lead us to successfully build the projects and ran the test cases as well.

*Step 3: Configuring JaCoCo -free code coverage library for Java.*

Metrics 1 & 2 are about the statement coverage and branch coverage. So, to get that data we chose JaCoColibrary since it is a free and popular code coverage library for Java. Since we were using the in IntelliJ Idea IDE, it was easy to configure the JaCoCo with IntelliJ Idea IDE. IntelliJ Idea has built in JaCoCo configured. We did not even need to modify the pon.xml file. Below are steps how we ran the test coverage using JaCoCo in IntelliJ Idea IDE.

*Step 1:*Make sure the project is building and test cases are running successfully.

*Step 2:*Create new Run/Debug configuration. Go to Run > Create/Edit configuration. As shown in Figure 1-1, choose “All in package” as test kind. Also make sure to choose the configuration for JaCoCo run as shown in Figure 1-2.

*Step 3:*Apply the newly created run profile and run JaCoCo with test coverage. Go to Run > “Run JaCoCo with Coverage”.

*Step 4:*Export the generated coverage results as shown in Figure 1-3.

A screenshot of a social media post

Description automatically generated

Figure 1‑‑: JaCoCo Configuration in IntelliJ Idea IDE

A screenshot of a social media post

Description automatically generated

Figure 1‑‑: JaCoCo Configuration in IntelliJ Idea IDE

A screenshot of a computer

Description automatically generated

Figure 1‑‑: JaCoCo Configuration in IntelliJ Idea IDE

While running the JaCoCo, sometimes it was producing the incorrect reports due to our mistakes in run configuration, we fixed the, resulting the correct statement and branch coverage reports (Figure 1-4).

A screenshot of a social media post

Description automatically generated

Figure 1‑‑ Sample JaCoCo report

*Step 4: Configuring PIT for Mutation Testing.*

Metric 3is about test suit effectiveness, and we chose PIT tool (<http://pitest.org>) to calculate mutation score.IntelliJ Idea has a plugin that simply provides results when we run mutation test on the selected test suits. The tool adds a 'Run configuration' that allows to execute PIT within IDE.

Usage: Run->Edit Configurations->Defaults->Pit Runner or just simply right click on project’s parent directory and choose to run PITest from the context menu as shown in Figure 2-1. Note here, it will apply the mutation on all the classes in the project since we are choosing the same option.

A screenshot of a computer

Description automatically generated

Figure 2‑1 PIT Configuration and running

After completion of running the mutation testing, it will generate the test reports like shown in Figure 2-2.

A screenshot of a cell phone

Description automatically generated

Figure 2‑2 PIT test coverage report

*Step 5: Configuring MetricsReloaded plugin in IntelliJ Idea IDE for static code analysis.*

MetricsReloaded plugin is used for static code analysis. We used this plugin to get Halstead Volume in order to calculate Metric 5.

The tool itself is very easy to use, we just need to install the plugin in IntelliJ Idea (https://plugins.jetbrains.com/plugin/93-metricsreloaded), specify the metrics and metrics calculation scope, and run it (Figure 3).

Usage: Choose Analyze > “Calculate Metric” from the menu.

A screenshot of a social media post

Description automatically generated

Figure 3 MetricsReloaded plugin configuration and usage

*Step 6:* Analyse the issue tracking systems and collect data for different versions of the software

For the Backlog Management Index, we collected the data for the projects using their issue tracking systems – Apache Issue Tracker (JIRA) for Apache projects and GitHub for JFreeChart. We decided to use the latest version of each project and collect the number of bugs reported and resolved every month over the period of one year using their respective issue tracking systems.

The data from JIRA can be readily collected using the *‘Created vs Resolved Issues Report’* and configuring the filters to generate the monthly data for the past 365 days. Meanwhile, using GitHub we can configure the filters in *‘Issues’* page by providing SQL like queries. For example, *‘is:issue created:2017-07-01..2017-07-31’* filter will give all the tickets that are issues and created in the month of July.

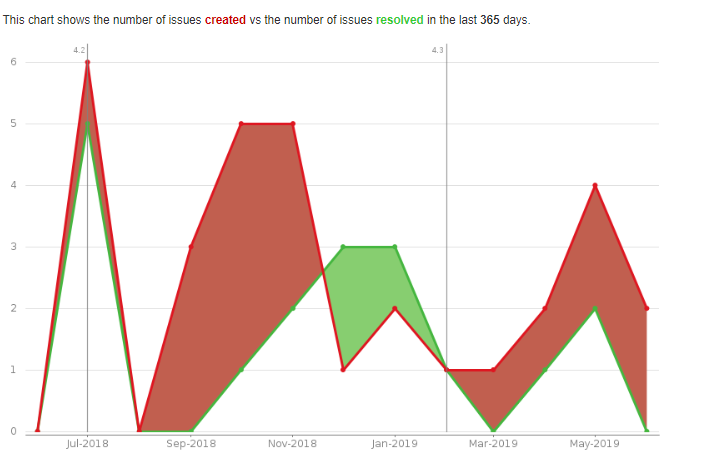


Fig.X. Created vs resolved issues – Apache Commons

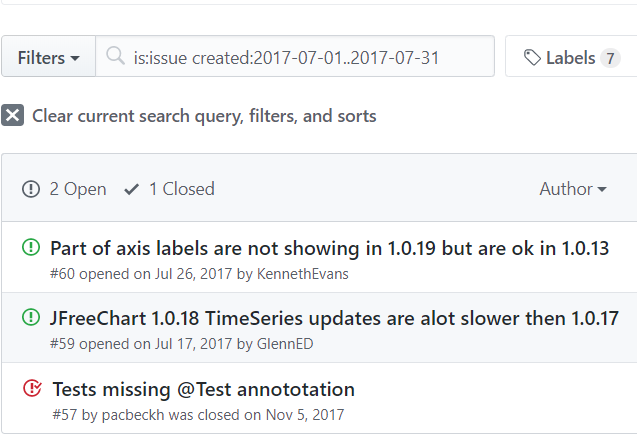


Fig.X. Issue Tracking using GitHub Collection

In the similar fashion, we can gather the data for each version using the version information from the projects’ homepage. After carefully plotting the tabulated data from the issue tracking systems in Excel, we can calculate the BMI for each project and take their average as shown in Fig. X1 and X2

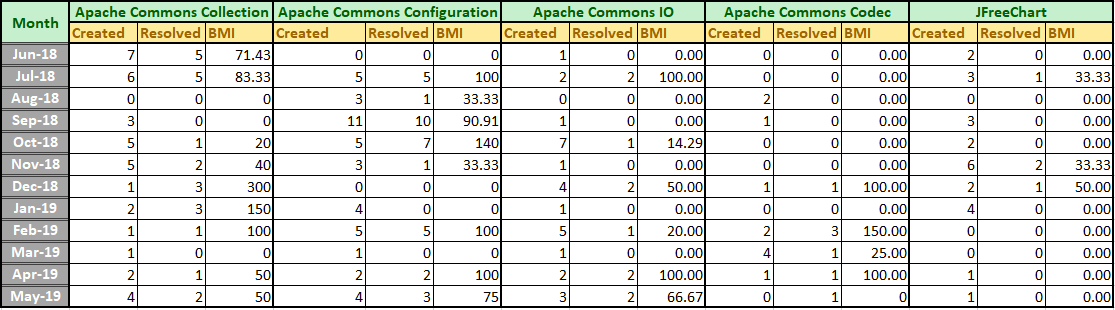


Fig. X1. BMI over the period of 1 year

As we can see some of the data in Fig. X1 shows BMI higher than 100. This indicates that the issue backlog of the project is reduced. Also, in some of the cases, the number of issues created(reported) is zero making the denominator zero which results in infinite BMI. Since, a project with infinite BMI does not make any sense, we take the BMI as zero if either the numerator or the denominator is zero.

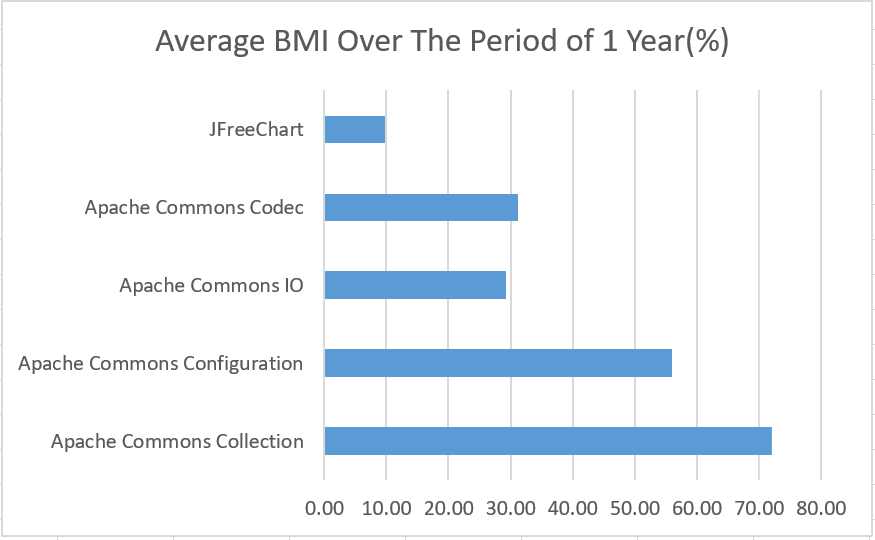


Fig. X2 – Average BMI over the period of 1 year

*Step 6: Manual work for data cleaning and* co-relations calculation *for metrics using MS Excel.*

After collecting all the data, we needed to do some manual work for calculating the co-relations among different metrics. We used MS Excel for that because it is easy to use yet good enough to calculate the co-relations and statistical work.

1. *Correlation Analysis*
2. *Correlation between Metric 1&2 and Metric 3*

The correlation analysis of metric 1&2 and metric 3 was carried out using spearman’s rank correlation coefficient rs.

* Correlation between 1 & 3

|  |  |  |
| --- | --- | --- |
| Project | Sets of data(class level) | rs of metric 1&3 |
| Apache commons codec | 119 | -0.2498 |
| Apache commons collections | 474 | -0.378 |
| Apache commons configuration | 306 | -0.3694 |
| Apache Common IO |  |  |
| JFreeChart |  |  |

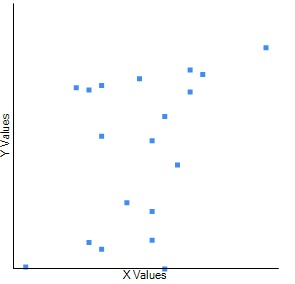
*Correlation coefficient between 1 & 3: 0.7882*

* Correlation between 2 & 3

|  |  |  |
| --- | --- | --- |
| Project | Sets of data(class level) | rs of metric 2&3 |
| Apache commons codec | 119 | -0.2498 |
| Apache commons collections | 474 | -0.378 |
| Apache commons configuration | 306 | -0.3694 |
| Apache Common IO |  |  |
| JFreeChart |  |  |

*Correlation coefficient between 2 & 3: 0.8270*

*Fig:* *Project: Apache Commons Collections  
X: Statement coverage  
Y: Mutation Coverage*

**

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

The correlation analysis of metric 1&2 and metric 4 was carried out using the Spearman's rank correlation coefficient *rs.*

* *Correlation between 1 & 4*

|  |  |  |
| --- | --- | --- |
| Project | Sets of data(class level) | rs of metric 1&4 |
| Apache commons codec | 119 | -0.2498 |
| Apache commons collections | 474 | -0.378 |
| Apache commons configuration | 306 | -0.3694 |
| Apache Common IO |  |  |
| JFreeChart |  |  |

*Calculate rs among the 5 projects(1663 sets of data) : = -0.3556*

* *Correlation between 2 and 4*

|  |  |  |  |
| --- | --- | --- | --- |
| Project | Project | Sets of data(class level) | rs of metric 2&4 |
| Apache commons codec | Apache commons codec | 119 | -0.2498 |
| Apache commons collections | Apache commons collections | 474 | -0.378 |
| Apache commons configuration | Apache commons configuration | 306 | -0.3694 |
| Apache Common IO | Apache Common IO |  |  |
| JFreeChart | JFreeChart |  |  |

*Calculate rs among the 5 projects(1663 sets of data) : = -0.2705*

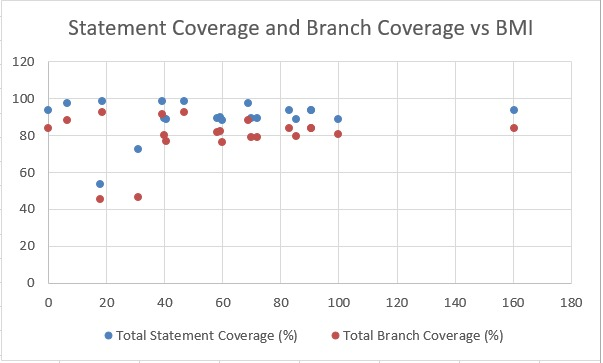
1. *Correlation between Metric 1&2 and Metric 6*

The Spearman correlation coefficients *Rs* for metric 1&2 and metric 6 are shown below.

*Correlation between metric 1 and 6 - Rs =* -0.03

*Correlation between metric 2 and 6 -**Rs =*  -0.06

Fig: **.** Data distribution diagram of different version between Metric 1,2 & 6 a) Apache commons Lang b) Apache commons codec c) Apache commons collections d) Apache commons configuration e) JFreeChart f) Apache Common IO.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Project** | **Vers** | **Total Statement Coverage in %** | **Total Branch Coverage in %** | **BMI** |
| Apache Common Configurations | 2.1.1 | 93.56 | 84.01 | 90.625 |
|  | 2.2 | 93.48 | 83.96 | 83.06 |
|  | 2.3 | 93.52 | 84 | 160.48 |
|  | 2.4 | 93.53 | 84.08 | 90.48 |
|  | 2.5 | 93.51 | 84.06 | 0 |
| Apache Common Collection | 4 | 88.44 | 76.21 | 59.93 |
|  | 4.1 | 88.6 | 76.86 | 40.56 |
|  | 4.2 | 89.31 | 78.82 | 72 |
|  | 4.3 | 89.51 | 78.91 | 70 |
| Apache Common Codec | 1.8 | 98.76 | 92.74 | 18.52 |
|  | 1.9 | 98.72 | 92.66 | 46.67 |
|  | 1.1 | 98.48 | 91.53 | 39.27 |
|  | 1.11 | 97.6 | 88.07 | 6.67 |
|  | 1.12 | 97.45 | 88.39 | 68.75 |
| Apache Commons IO | 2.6 | 89.28 | 80.25 | 40.03 |
|  | 2.5 | 88.97 | 80.85 | 99.72 |
|  | 2.4 | 89.71 | 82.11 | 59.33 |
|  | 2.3 | 89.26 | 81.52 | 58.33 |
|  | 2.1 | 88.82 | 79.28 | 85.28 |
| JFreeChart | 1.5.0 | 72.22 | 46.67 | 31.05 |
|  | 1.0.18 | 53.29 | 45.12 | 17.78 |
|  | 1.0.19 | 93.56 | 84.01 | 90.625 |

1. *Correlation between Metric 5 and Metric 6*

The Spearman correlation coefficient Rs is calculated from the above 14 sets of data, and the value of *R(Spearman)* was -0.21562, negative and small correlation If MI increases, then BMI will decreases.

|  |  |  |  |
| --- | --- | --- | --- |
| **Project** | **Project Versions** | **Maintainability Index** | **BMI** |
| Apache Common Configurations | 2.1.1 | 65.13 | 90.625 |
|  | 2.2 | 65.14 | 83.06 |
|  | 2.3 | 65.12 | 160.48 |
|  | 2.4 | 64.74 | 90.48 |
|  | 2.5 | 64.74 | 0 |
| Apache Common Collection | 4 | 69.16 | 59.93 |
|  | 4.1 | 69.12 | 40.56 |
|  | 4.2 | 68.84 | 72 |
|  | 4.3 | 68.83 | 70 |
| Apache Common Codec | 1.8 | 65.3 | 18.52 |
|  | 1.9 | 65.27 | 46.67 |
|  | 1.1 | 65.26 | 39.27 |
|  | 1.11 | 65.72 | 6.67 |
|  | 1.12 | 65.79 | 68.75 |
| Apache Commons IO | 2.6 | 64.23 | 40.03 |
|  | 2.5 |  | 99.72 |
|  | 2.4 |  | 59.33 |
|  | 2.3 |  | 58.33 |
|  | 2.1 |  | 85.28 |
| JFreeChart | 1.5.0 | 62.26 | 31.05 |
|  | 1.0.18 | 62.39 | 0 |
|  | 1.0.19 | 62.39 | 17.78 |
|  |  |  |  |

1. CONCLUSIONS

***Correlation between metric 1&2 and 3***

The correlation between metric 1&2 and metric 3 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.

***Correlation between metric 1&2 and 4***

The correlation between metric 1&2 and 4 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.

***Correlation between metric 1&2 and 6***

The correlation coefficients for metric 1&2 and metric 6 negative and small correlation. If the branch coverage and statement coverage increases, then BMI will decrease slightly.

The rationale is that if the program has high branch coverage and statement coverage (more of its source code executed during testing) then it has a lower chance of containing undetected software bugs. That means higher test coverage helps to increase software quality.

This conclusion is consistent with the rationale that classes with higher branch and statement coverage, then BMI will decrease.

***Correlation between metric 5 and 6***

The correlation coefficients of the metric 5 and metric 6 were negative and small correlation.We can conclude if MI increases, then BMI will decreases. This conclusion is consistent with the rationale that if MI increases then BMI decreases.

1. REFERENCES
2. <https://www.researchgate.net/publication/321091593_Correlations_of_software_code_metrics_an_empirical_study>
3. <https://www.jetbrains.com/idea/>.
4. <https://commons.apache.org/proper/commons-collections/download_collections.cgi>.
5. <https://commons.apache.org/proper/commons-codec/>
6. https://projects.apache.org/project.html?commons-configuration
7. <https://commons.apache.org/proper/commons-lang/>.
8. <https://www.javatpoint.com/statement-coverage-testing-in-white-box-testing>.
9. T. T. Chekam et al, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017, Available: http://dx.doi.org/10.1109/ICSE.2017.61. DOI: 10.1109/ICSE.2017.61.
10. http://www.informit.com/articles/article.aspx?p=30306&seqNum=3