Cover Letter

Dear Professor Yang,

Our team would like to submit the enclosed manuscript entitled “Correlation Metrics”. In this paper, we did correlation analysis among followed six metrics: branch coverage, statement coverage, mutation score, McCabe complexity, backlog management index and change proneness.

The followed table is the information of our team members:

Name	Student Number	E-mail
Runsen Tian	40083990	Rinotrs@gmail.com
Baiyu Huo	40076004	boyu.huo.china@gmail.com
Liangzhao Lin	40085480	283477489@qq.com
Hao Ma	40057767	mh2923166@gmail.com

Following is a link to the replication package in GitHub:
https://github.com/BoyuHuo/Metrics-Correlation-Analysis.git

Thank you very much for your attention.

Sincerely,
Team D

Exploring the Correlation among Different Metrics

Baiyu Huo
Concordia University
gina cody school of engineering and computer science
Montreal, Canada
boyu.huo.china@gmail.com

Liangzhao Lin
Concordia University
gina cody school of engineering and computer science
Montreal, Canada
283477489@qq.com

Runsen Tian
Concordia University
gina cody school of engineering and computer science
Montreal, Canada
Rinotrs@gmail.com

Hao Ma
Concordia University
gina cody school of engineering and computer science
Montreal, Canada
mh2923166@gmail.com

***Abstract*—Correlation metrics measure whether or not there is a relationship between two variables. In this paper, we focus on several common internal metrics: statement coverage, branch coverage, mutation score, McCabe complexity, fix backlog and backlog management index and change proneness. 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 Pearson Correlation and 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 were very small; backlog management index and change proneness were positively correlated and moderately strong.

Keywords—metric, correlation, software measurement

I. Introduction

With the development of science and technology, the software industry has been growing at an accelerated rate for the past 30 years [7]. Excellent software can improve work efficiency, while software with some bugs may have a bad impact on works. Therefore, software measurement becomes significant in recent years. In development, software developers and project managers need to continuously evaluate software products and study relations and correlations of all attributes and factors that can impact positively or negatively their developed software products. 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.

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 [8].

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 [8].

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

III. 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 Lang

https://commons.apache.org/proper/commons-lang/

Lang provides a host of helper utilities for the java.lang API, notably String manipulation methods, basic numerical methods, object reflection, concurrency, creation and serialization and System properties. Additionally, it contains basic enhancements to java.util.Date and a series of utilities dedicated to help with building methods, such as hashCode. toString and equals.

It is a large open source project which has 79.8K LOC and a continuous issue records in its issue tracking system. We are using versions from 3.0 to 3.8 to collecting the data for our experiment.

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.

The configuration is a large apache project, which contains serval active versions as well as a continuous bug-tracking system, which list out all the issues and its detail description, solving status and timestamp. It makes our data collection work for metrics 5 easier. We're collecting data using versions from 2.1 to 2.4.

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.

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. Meanwhile, it also contains an issue tracking system and a lot of subversions. We are using versions from 1.10 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 4.4 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.

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[5].

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.

IV. 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

Test coverage is an important metric of software quality, since it indicates thoroughness of testing. In industry, test coverage is often measured as statement coverage [1]. According to the actual development experience, statement coverage is suitable for software measurement in projects analysis. 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 [2].

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. However, branch test is available to detect cryptic errors in code. 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 [2].

Metric 3: Mutation Score

To find weakness of code, mutation score is an useful measurement metric. Mutation score could be obtained through mutation testing. Mutation testing is a means of creating more effective test cases. Mutation testing is primarily used as a program-based technique. It uses mutation operations to mutate the program and generate program mutants. The goal in mutation testing is killing the generated mutants by causing the mutant to have different behavior from the original program on the same input data [3]. The way to calculate mutation score as follow:

$$MutationScore = \frac{KilledMutants}{Totalnumberof Mutants} * 100$$

Metric 4: McCabe Metric (Cyclomatic Complexity)

Complexity is vital to software measurement. In the analysis, McCabe complexity (Cyclomatic Complexity) was selected. Cyclomatic Complexity is used as indicators for program modularization, revising specifications, and test coverage. In addition, it has been used in software quality prediction models, whose purposes include predicting fault numbers through multivariate regression analysis and identification of error-prone modules based on discriminant analysis [4]. For calculating McCabe complexity, followed elements should be counted:

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

- D = the number of control predicate (or decision) statements
- For a single method or function, P is equal to 1
- Cyclomatic Complexity = E – N + 2P

  (Or Cyclomatic Complexity = D + 1)

Metric 5: Fix Backlog and Backlog Management Index (BMI)

In software measurement, software maintenance effort should not be ignore. Backlog and backlog management index is related to software maintenance effort and it is a metric to manage the backlog of open, unresolved. If BMI is less than 100, then the backlog increased. With enough data points, the techniques of control charting can be used to calculate the backlog management capability of the maintenance process. More investigation and analysis should be triggered when the value of BMI exceeds the control limits. A BMI trend chart or control chart should be examined together with trend charts of defect arrivals, defects fixed (closed), and the number of problems in the backlog [5]. The formula as follow shows:

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

Metric 6: Change Proneness

Software change-proneness is a significant metric related to software quality, it predicts whether or not class files in a project will be changed in their next release, can help software developers allocate resources more effectively and reduce software maintenance costs. Previous studies found that change-proneness prediction cannot work well with limited training data, especially for new projects [6].

To calculate change-proneness of software, We assessed the change-proneness of a class C(i) in a release r(j) as the number of changes and the number of bug fixes C(j) was subject to in the time period t between the r(j) and the r(j+1) release dates. This implies that the length of t could play a role in the change-proneness of classes (i.e., the longer t the higher the class change-proneness). However, this holds for both smelly and non-smelly classes, thus reducing the bias of t as a confounding factor.

To mitigate such a threat we completely re-run our analyses by considering a normalized version of class change proneness. In particular, we computed the change-proneness of a class C(j)in a release r(j)as:

$$change\_proneness(C_i, r_j) = \frac{\#Changes(C_i)_{r_{j-1} \to r_j}}{\#Changes(r_{j-1} \to r_j)}$$

V. STEPS TO COLLECTING THE DATA

Our data collecting work can be totally divided into 6 steps:

- S1.selecting projects.
- S2.building projects.
- S3.configuring Jacoco plugin.
- S4.adding pit test plugin.
- S5. selecting the active period for issues tracking and collecting related data from the issue tracking system.
- S6.write shell script for change-report and collecting changes-data from different subversions.

Step1: Selecting projects

In order to boost our later process, we're carefully choosing the projects which meet the following standards:

- It is an open source project which is also programmed in Java Language.
- It is ether build by Maven or by Ant.
- It should be a single module project and the size of it shouldn't be too small.
- There is an issue-tracking system which contains continuous issue-solving records.
- There are several subversions for us to collect data.

After filtering many unqualified projects, we finally narrow down our searching scope to Apache project, since most of them are meet our standards in terms of size, programming language, issue tracking system as well as serval subversions.

Step 2: Building the projects

After selecting projects, we tried to build all of them, in order to see if there are some crucial problems or doesn't contain any unit test cases. For those contains some small problem, such as JDK version difference, we will fix it. However, for those projects which have crucial problems or doesn't exist any unit test cases, we will drop this project and then go back to step 1.

In conclusion, in step two, we're validating the selecting to see whether it is suitable for our experiment.

Step 3: Adding Jacoco plugin

In order to collect the data for statements coverage, branch coverage as well as complexity, we're configuring for each project including its subversions that we choose to generate the Jacoco reports. As you can see in Figure 1, We adding Jacoco plugin into build file for each project(pom for maven projects) and adding Jacoco reports task into the test phase.

During the process, some of the projects show some problems such as some test cases cannot be passed, so it will prevent Jacoco to generate the report. We are using two solutions to solving this problem. First, changing the expected value for that test case so it can be passed. Second, delete this test case. Since all of the projects that we chose contains thousands of test cases, so one or two test cases won't affect the final result. The example Jacoco report is shown in Figure 2.

Fig.1. Jacoco configuration (Commons Lang)

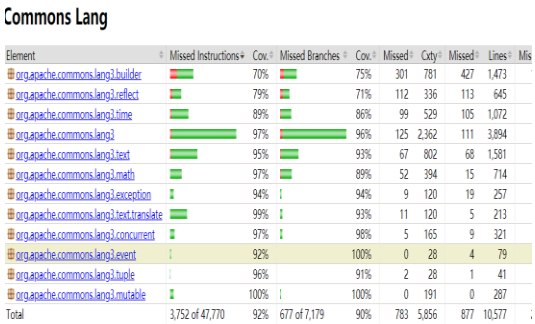

Fig.2. Jacoco report (Commons Lang)

Step 4: Adding pit plugin

For the mutation score, we are using pit plugin to generate the report. In the configuration, it allowed us to choose the mutator, target java classes, target test cases (configuration shown in Figure 3).

We used 7 default mutators(operators) to generate the mutation which contains: Conditionals Boundary, Increments, Invert Negatives, Math, Negate Conditionals, Return Values and Void Method Calls.

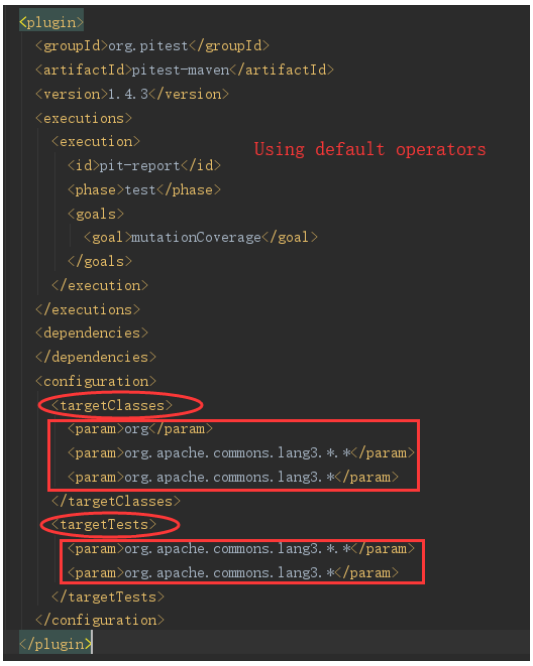

Fig.3. Pitest configuration(Commons Lang)

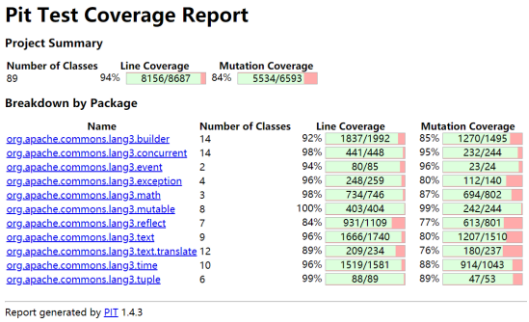

Fig.4. PItest Report (Commons Lang)

Step 5: Selecting active month and collecting data from the issue tracking system.

We decided to collect three active month data in each subversion and then get the average value among these three months for Fix Backlog and Backlog Management Index. Since for software, a new version comes to release doesn't mean the previous version is out of the stage. So you cannot using the total life of the projects to calculate BMI value. We carefully located the 3 active continuous months which has the largest number of issue arrivals for each subversion and then calculate the BMI based on the average value of these three months.

The issue-tracking system has basic statistics information of the issues and what we need to do is manually search the active month by SQL that it offered like Figure 5. Finally using excel to calculate all BMI values.

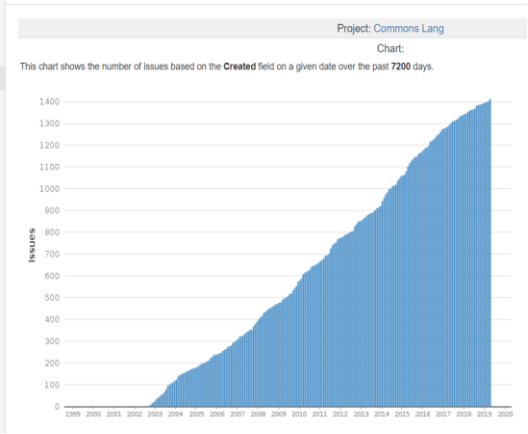

Fig.5. Active month located work (Lang)

Date	Number c	Number c	BMI		Max of BM	Min of BM	Average o	Version Coverage
Jul-11	24	3	12.5		20	0	10.83333	3.0~3.6
Aug-11	15	3	20					
Sep-11	8	0	0					
Jul-17	3	0	0		80	0	43.33333	3.6~3.7
Aug-17	5	4	80					
Sep-17	4	2	50					
Jan-18	5	2	40		40	28.57143	33.96825	3.7~3.8
Feb-18	3	1	33.33333					
Mar-18	7	2	28.57143					

project = LANG AND issuetype = Bug AND affectedVersion in (3.0,3.6) AND status = Closed AND createdDate > "2011/01/01" ORDER
https://issues.apache.org/jira/secure/ConfigureReport.jspa?projectOrFilterId=project-12310481&dateField=created&periodName=mont

Fig.6. BMI Report (Commons Lang)

Step 6: Writing script for collecting change-report and collecting changes-data from different subversions.

We're collecting data mainly for change proneness in this part. All of our collection is based on git log, which will compare two different submits or two different timestamps, and then give us the number of change lines for each file. To make the whole process more smooth, we write a script in shell, which will get the log file base on the input of 2 subversion's token, then do the calculation and convert it into the form what we want, and finally, convert it to a CSV file, so it will be easy for our later analysis work. The script shows in Figure 7.

After we got the report CSV file, we remove those file which is not Java, since we want only compare the changed line in each class and the total number of changes. What's more, we also delete all the test case file, because we also need to analyze the relation with code coverage, the Jacoco report won't generate the report for the test case. Our final report for Change proneness shown in Figure 8.

```
echo "*------------------------------------------------------------*"
echo "Welcome Soen 6611 Team D Change proness data collecting tool !"
echo "*------------------------------------------------------------*"
echo "Please input the version 1 token:"
read version1
echo "Please input the version 2 token:"
read version2
echo "Generating the change report..."
git diff $version1 $version2 --stat > test.txt
echo "Report, OK!"
echo "Converting txt into csv report..."
cat test.txt  | sed 's/\t/,/g;s/[[:space:]+-]//g;s/|/,/g;' >test.csv
echo "Converted, OK!"
for line in `cat test.csv`; do echo $line| awk -F "/" '{print $NF}'; done > diff.csv
echo "diff.cvs has generated!"
```

Fig.7 Change report script

Files	Change Number	Change proness value for each java file
ArchUtils.java	22	0.025641026
ArrayUtils.java	53	0.061771562
CharRange.java	4	0.004662005
CharSequenceUtils.java	12	0.013986014
CharSet.java	4	0.004662005
CharSetUtils.java	2	0.002331002
CharUtils.java	24	0.027972028
ClassUtils.java	158	0.184149184
JavaVersion.java	12	0.013986014
LocaleUtils.java	14	0.016317016
ObjectUtils.java	2	0.002331002
RandomStringUtils.java	12	0.013986014
SerializationUtils.java	2	0.002331002
StringUtils.java	71	0.082750583
SystemUtils.java	12	0.013986014
ThreadUtils.java	4	0.004662005
Validate.java	24	0.027972028
Processor.java	2	0.002331002
CompareToBuilder.java	2	0.002331002
EqualsBuilder.java	2	0.002331002
EqualsExclude.java	2	0.002331002
HashCodeExclude.java	2	0.002331002
ReflectionToStringBuilder.java	4	0.004662005
ToStringBuilder.java	2	0.002331002
ToStringExclude.java	2	0.002331002

Fig.8. final report for Change proneness (Commons Lang)

VI. Steps for Analyzing the Data

We use Pearson correlation coefficient and Spearman correlation coefficient to analyze correlation.

The Steps of data analysis are as follows:

- Determining which two metrics are used for correlation comparative analysis and determining which level of data they are (e.g. package level, class level). Extracting the metric data of specific project from the collected data.
- Importing the collected metric data into a MATLAB program in order to calculate their Pearson correlation coefficient and Spearman's rank correlation coefficient. Collecting the correlation coefficient and generating the distribution diagrams of the data.
- 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**
Total 5 project	1063	0.7476
Apache commons Lang	89	-0.0564
Apache commons codec	52	0.8027
Apache commons collections	264	0.4510

Apache commons configuration	177	0.8266
JFreeChart	481	0.7996
Apache commons Lang	11 sets Package-Level data	0.3152

TABLE II. *R(PEARSON)*BETWEEN METRIC 2 & 3

Project	**Sets of data (Class level)**	***R(Pearson)*of metric 2&3**
Total 5 project	899	0.7707
Apache commons Lang	75	-0.0847
Apache commons codec	47	0.8674
Apache commons collections	206	0.3714
Apache commons configuration	143	0.753
JFreeChart	428	0.7996
Apache commons Lang	11 sets Package-Level data	0.8627

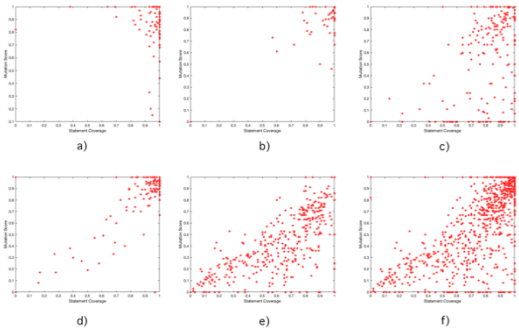

Fig.9. Data distribution diagram of class level between Metric 1 & 3 a) Apache commons Lang b) Apache commons codec c) Apache commons collections d) Apache commons configuration e) JFreeChart f） Total five project class level data.

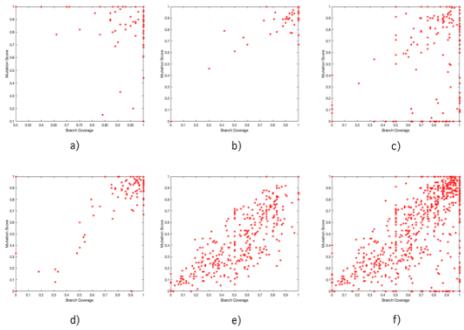

Fig.10. Data distribution diagram of class level between Metric 2 & 3 a) Apache commons Lang b) Apache commons codec c) Apache commons collections d) Apache commons configuration e) JFreeChart f) Total five project class level data

It can be seen from Figure 9 and 10, as well as table Ⅰ and table Ⅱ above that the *R(Pearson)* of the four groups is strong and the direction of the correlation is positive except the 'Apache Commons Lang' project. The correlation coefficient of 'Apache Commons Lang' project is obviously different from the other four projects.

Hence, we summarized and analyzed the class level data of the five projects and calculated that the *R(Pearson)* of metric 1&3 of total 5 projects is 0.7476, and *R(Pearson)* of metric 2&3 of total 5 projects is 0.7707.

As for why the correlation coefficient of the project Apache Commons Lang is very small. We found that the data of the Apache Commons Lang metric 1 and metric 2 is concentrated on more than 90%, as shown in Figure 11. Therefore, the data distribution is too centralized to form a good correlation comparison, and it is easy to cause the deviation of the correlation coefficient on Apache Commons Lang. We specifically list 11 sets of data for the Apache commons Lang package level in Table Ⅰ and Table Ⅱ, we can see *R(Pearson)* is strong and positive for metric 1&2 and metric 3. Combining with the similar *R(Pearson)* of the four groups of projects and the universality of the five groups of data, it can be seen considered that the correlation coefficient on the class level of Apache Commons Lang is an abnormal result, which is not universal and can be ignored.

Therefore, we can conclude from the *R(Pearson)* of the five groups of projects that the correlation between metric 1&2 and metric 3 is very strong and positive.

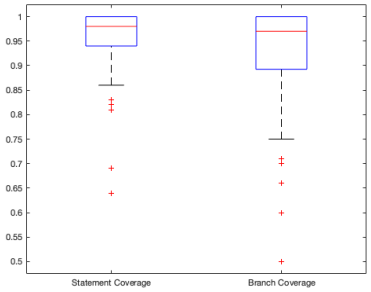

Fig.11. Boxplot of Apache commons Lang class level data of metric 1 and metric 2

B. 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 r_s.

TABLE III. r_S BETWEEN METRIC 1 AND METRIC 4

Project	**Sets of data (Class level)**	**r_s of metric 1&4**
Total 5 project	1663	-0.3556
Apache commons Lang	246	-0.2116

Apache commons codec	89	-0.2605
Apache commons collections	474	-0.3780
Apache commons configuration	306	-0.3694
JFreeChart	548	-0.0655

TABLE IV. r_s BETWEEN METRIC 2 AND METRIC 4

Project	Sets of data (Class level)	r_s of metric 2&4
Total 5 project	1174	-0.2705
Apache commons Lang	162	-0.2985
Apache commons codec	59	-0.3509
Apache commons collections	319	-0.2614
Apache commons configuration	197	-0.3177
JFreeChart	437	0.0958

As can be seen from Figure 12 and the Spearman correlation coefficients r_s of metric 1&4, 2&4 of the five projects in Table Ⅲ and Table Ⅳ above, r_s of most projects are around -0.3, so 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.

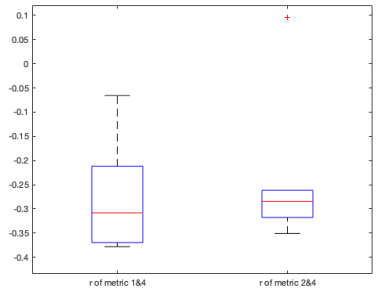

Fig.12. Boxplot of r_s of metric1&4 and metric 2&4

C. Correlation between Metric 1&2 and Metric 6

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

Project	Sets of data (Class level)	R (Pearson)of metric 1&6
Apache commons Lang	126	-0.0544
Apache commons codec	60	-0.0761
Apache commons collections	270	-0.0237
Apache commons configuration	186	0.0404
JFreeChart	524	0.0328

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

Project	Sets of data (Class level)	R (Pearson)of metric 1&6
Apache commons Lang	126	-0.0544
Apache commons codec	60	-0.0761
Apache commons collections	270	-0.0237
Apache commons configuration	186	0.0404
JFreeChart	524	0.0328

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 five projects are less than 0.01. Consequently, we infer that there is almost no correlation between metric 6 and metric 1&2.

D. Correlation between Metric 5 and Metric 6

The Pearson correlation coefficient *R(Pearson)*is calculated from the above 14 sets of data, and the value of *R(Pearson)* was 0.2732, so it shows that the positive correlation between metric 5 and metric 6 is medium.

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

Project (Version-Version)	Metric5 BMI	Metric 6 Change proneness
Apache commons Lang 3.0-3.6	10.833	0.00591716
Apache commons Lang 3.6-3.7	43.333	0.020833333
Apache commons Lang 3.7-3.8	33.9683	0.017241379
Apache commons codec 1.10-1.11	30.5556	0.03125
Apache commons codec 1.11-1.12	44.4444	0.041666667
Apache commons codec 1.9-1.10	100	0.025641026
Apache commons collections 3.2-4.0	40.3175	0.00177305
Apache commons collections 4.0-4.1	38.611	0.005076142
Apache commons collections 4.1-4.3	41.6667	0.003030303

Apache commons configuration 2.1-2.2	66.667	0.00990099
Apache commons configuration 2.2-2.3	15.7576	0.071428571
Apache commons configuration 2.3-2.4	3.0303	0.005235602
Jfreechart 1.0.18-1.0.19	250	0.045454545
Jfreechart 0.19-1.5.0	66.667	0.000770416

VIII. CONCLUSIONS

According to result *VII. A.*, it shows that 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.

According to result *VII. B.*, it shows that 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.

According to result *VII. C.*, 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.

According to result *VII. D.*, it shows that the Pearson correlation coefficients of the metric 5 and metric 6 were positively correlated and moderately strong. We conclude that on the project-level, project with higher Backlog Management Index might show higher change proneness.

REFERENCES

[1] R. M. Hierons, M. Harman and C. J. Fox, "Branch-coverage testability transformation for unstructured programs," Comput. J., vol. 48, (4), pp. 421-36, 2005. Available: http://dx.doi.org/10.1093/comjnl/bxh093. DOI: 10.1093/comjnl/bxh093.

[2] 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.

[3] M. H. Moghadam and S. M. Babamir, "Mutation score evaluation in terms of object-oriented metrics," in 2014 4th International Conference on Computer and Knowledge Engineering (ICCKE), 2014, Available: http://dx.doi.org/10.1109/ICCKE.2014.6993419. DOI: 10.1109/ICCKE.2014.6993419.

[4] R. Takahashi, "Software quality classification model based on McCabe's complexity measure," J. Syst. Software, vol. 38, (1), pp. 61-69, 1997. Available: http://dx.doi.org/10.1016/S0164-1212(97)00060-5. DOI: 10.1016/S0164-1212(97)00060-5.

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

[6] Y. Ge et al, "Deep metric learning for software change-proneness prediction," in 8th International Conference on Intelligence Science and Big Data Engineering, IScIDE 2018, August 18, 2018 - August 19, 2018, Available: http://dx.doi.org/10.1007/978-3-030-02698-1_25. DOI: 10.1007/978-3-030-02698-1_25.

[7] R. Hofman, "Behavioral economics in software quality engineering," Empirical Software Engineering, vol. 16, (2), pp. 278- 293, 2011. Available: http://dx.doi.org/10.1007/s10664 -010-9140-x. DOI: 10.1007/s10664-010- 9140-x.

[8] 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.