Correlation Between Different Software Measurement metrics on Open Source Project

Haifeng Wu, Basant Gera, Sandeep Kaur Ranote, Sourabh Rajeev Badagandi , Meet Patel
Department of Computer Science & Software Engineering - Concordia University
1455 Boulevard de Maisonneuve O, Montréal, QC H3G 1M8, Canada
{w_haifen,b_gera, s_ranote, s_badaga,, me_el}@encs.concordia.ca
Github url : https://github.com/sourabh-rb/SOEN6611

Abstract

Software metrics are standards which measure software system properties. In this paper, we are exploring Statement Coverage, Branch Coverage, Mutation Score, McCabe Complexity, Package Instability, Post Release Defect Density, they have included coverage metrics, test suite effectiveness, complexity metric, coupling metric, and software quality metrics. We selected 4 open source projects: Apache Commons Collections, Apache Commons Lang, Alibaba fastjson, Apache Commons Configuration to analysis. We showed how we collect the data, the configuration of tools, and the data we get. Finally, we analyzed the correlation of metrics using Pearson coefficient. In conclusion, coverage metrics have a positive correlation with Mutation Score, have a negative correlation with McCabe Complexity, have a negative with post release defect density. Package Instability has a negative correlation with post release defect density.

Keyword Used : Mutation Testing , cyclomatic complexity,Test suite,coverage.

I. Introduction

With the development of software engineering, software measurement becomes more and more important. Software companies hope their products will be completed on time and on budget. Software developers and project managers must continuously evaluate software products and study relations and correlations of all properties and factors that may affect the process and budget. Software metrics help developers to have a good knowledge about their productions.

Academic area has proposed many software metrics, some of them are widely used in industry areas. Coverage metrics measure how many lines of code or branches are tested by test suites. Mutation score measures test suites by effectiveness. Complexity metric measures modules, packages, classes, methods in terms of complexity. We also propose two metrics in this paper, one of them quantifies one coupling of the program, the other one measures the quality attributes.

Regarding the correlation between metrics, there are some hypotheses for it. For example, test suites with higher coverage index tend to have better effectiveness. A class with higher complexity often has less code and branches to be tested. Programs with low test coverage contain more bugs. Finally, if a package has a higher instability, which means it has more dependencies, it shall have more bugs because of the dependency bugs. We will get the conclusion of these hypotheses.

The table which is shown below describes the open source projects we have taken for our analysis and their respective github and official website links:

Project Name	*Github Link*	*Official website Link*
1.Apache Commons Collections	https://github.com/apache/commons-collections	https://commons.apache.org/proper/commons-collections/
2.Apache Commons Lang	https://github.com/apache/commons-lang	https://commons.apache.org/proper/commons-lang/
3.Alibaba fastjson	https://github.com/alibaba/fastjson	https://github.com/alibaba/fastjson/wiki
4.Apache Commons Configuration	https://github.com/apache/commons-configuration	https://commons.apache.org/proper/commons-configuration/

TABLE : Project Name with their github &offical website links.

II. Related work

Coupling and Cohesion are two fundamental concepts that can be applied to design better modular object-oriented software. Coupling shows the dependency of one class on the other classes whereas cohesion defines the degree of connectivity among the elements of a module. Many researchers earlier have proposed different metrics to capture the OO features and many new metrics are still being proposed. These metrics capture the different aspects of the object-oriented system. J. Bansiya et al. [1] proposed a new metric to measure cohesion in a class named Cohesion Among Methods of Classes (CAMC). The CAMC metric is based on the type of input parameters that are taken by the method. It checks the relation of methods of a class based on their parameter list i.e. if all the methods of a class access similar input parameters, it can be concluded that they are closely related. The proposed metric was validated by comparing the results with the LCOM metric on a set of 17 classes from various commercial projects and was evaluated by highly trained software experts.

A lot of work has been done on the subject of code coverage and test suite effectiveness. Fault injection is one of the widely-used approaches to evaluate the effectiveness of test data so as to mitigate the risks posed by faulty OTS(Off-The-Shelf) components in a critical system. J. Bieman et al. [2], proposed using the same mechanism to improve test coverage in software systems. The paper developed a Visual C-Patrol system by extending the C-Patrol assertion insertion system to support fault injection. The VCP is based on an assertion violation scheme that makes use of pre and post-condition assertions i.e., by dynamically changing the state of a running program, we can create a fault so that the pre and post conditions are not met. The paper analyzed 4 projects using the VCP system and detected that they received 90% coverage or higher in all individual functions except one. They received improved coverage in almost 75% of the source files, along with greater than 80% branch coverage.

Package instability and abstractness metrics were defined by Martin and these metrics were enhanced by S. Almugrin et al. [3]. Martin's two coupling measures were efferent coupling (Ce) and afferent coupling (Ca), where (Ce) defined the number of dependencies of that package on other packages and (Ca) defined the number of classes outside a package which depend on the classes inside this package. The paper enhanced Martin's metrics based on responsibility factor, i.e. providing different weight to packages based on their level of responsibility. Afferent coupling (Ca) of a package was used as an indicator of the package's responsibility. The effectiveness of the new metrics was validated by comparing the results with Martin's instability and abstractness metrics and improved results were found.

III. Preliminaries

A. GitHub

GitHub is a Git repository hosting service which has a lot of features.It provides a web based graphical interface. It also provides access control and several collaborations such as wiki & basic task management tools for every project which can be used in agile methodology also.Some of the benefits of github are documentation, you can also showcase your work or markdown someone else work and you can track down changes in your code across the versions.

B. JIRA

JIRA is a tool which is designed to track a bug *(Fault)* and can be called a bug tracking system and project management software.The basic use of JIRA is to track faults and bugs of the software related to mobile apps and websites.It is also a agile project management tool which supports scrum, kanban with its unique flavour.We have "Ticket" to solve issue which is than have some status such as Solved, Unsolved.

C. *Maven*

Maven is a build automation tool which is generally used in java projects.It can also used to build and manage projects with C#, Ruby, Scala.The maven project is hosted by Apache Software foundation. It generally addressed two aspects of building software which are building the tool and its dependencies. It uses a XML file which describes the software project which is build order, directories and plugins.It generally downloads java libraries & plugins which can be used by the local projects.

Now there are some alternative tools in the market such as Gradle,Ant & Sbt which can be used to build the projects in java and many other projects.

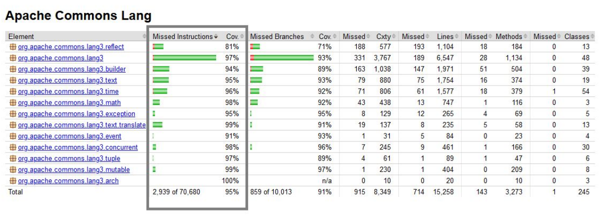
IV. Metrics description

Metric 1: Statement Coverage

Statement Coverage or line Coverage as it is called sometimes is a part of White box testing, it involves execution of source code statements at least once given the multiple inputs. The statement doesn't necessarily need to be executed on one run, if all the statements are executed on different runs with different inputs then it will have 100% coverage. To calculate the Statement Coverage following formula is used:

$$Statement\ Coverage\ = \frac{Number\ of\ Statements\ Executed}{Total\ Number\ of\ Statements} \times 100$$

Statement Coverage is Whitebox testing Because it involves testing source code unlike Blackbox where source code is not involved. It is calculated using Jacoco Tool, Jacoco uses test units as input and based shows Statement Coverage based on the test cases of the source code.



Apache Commons Lang

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.lang3.reflect		81%		71%	188	577	193	1,104	18	184	0	13
org.apache.commons.lang3		97%		93%	331	3,767	189	6,547	28	1,134	0	48
org.apache.commons.lang3.builder		94%		89%	163	1,038	147	1,971	51	504	0	39
org.apache.commons.lang3.text		95%		93%	79	880	75	1,754	16	374	0	19
org.apache.commons.lang3.time		96%		92%	71	806	61	1,577	18	379	1	54
org.apache.commons.lang3.math		98%		92%	43	438	13	747	1	116	0	3
org.apache.commons.lang3.exception		95%		95%	8	129	12	265	4	69	0	5
org.apache.commons.lang3.text.translate		99%		91%	19	137	8	235	5	58	0	13
org.apache.commons.lang3.event		91%		93%	1	31	5	84	0	23	0	4
org.apache.commons.lang3.concurrent		98%		96%	7	245	9	461	1	166	0	30
org.apache.commons.lang3.tuple		97%		89%	4	61	1	89	1	47	0	6
org.apache.commons.lang3.mutable		99%		97%	1	230	1	404	0	209	0	8
org.apache.commons.lang3.arch		100%		n/a	0	10	0	20	0	10	0	3
Total	2,939 of 70,680	95%	859 of 10,013	91%	915	8,349	714	15,258	143	3,273	1	245

In the Given example of Apache Commons Lang we can see that Jacoco gives us the Statements that were missed at the bottom, It also shows the coverage of each package and each class, The overall coverage is also shown at the bottom. It can also show what part of the code was missed inside a class if you click on it.

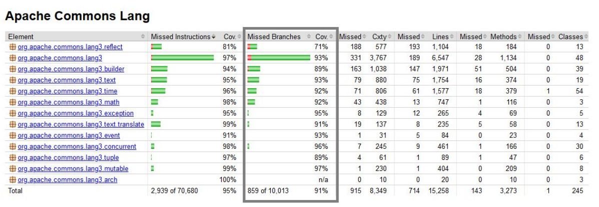
Statement coverage can help identify the dead code in the program. It helps find statements that are missed, It is also useful in identifying the flow of the code.

Metric 2: Branch Coverage

Branch Coverage is also a part of Whitebox testing, Here we test each branch in source code when represented as a class diagram, A branch occurs at every decision statement and only considers those that have its condition outcome as True. To get 100% coverage every branch needs to be executed but not necessarily in one run. To calculate branch coverage following formula is used:

$$Branch\ Coverage\ =\ \frac{Number\ of\ Executed\ Branches}{Total\ Number\ of\ Branches}\ \times\ 100$$

Branch Coverage can be using Jacoco as well. Jacoco gives branch coverage based on the test units in your program. It can also give coverage of every package and class.



Apache Commons Lang

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.lang3.reflect		81%		71%	188	577	193	1,104	18	184	0	13
org.apache.commons.lang3		97%		93%	331	3,767	189	6,547	28	1,134	0	48
org.apache.commons.lang3.builder		94%		89%	163	1,038	147	1,971	51	504	0	39
org.apache.commons.lang3.text		95%		93%	79	880	75	1,754	16	374	0	19
org.apache.commons.lang3.time		96%		92%	71	806	61	1,577	18	379	1	54
org.apache.commons.lang3.math		98%		92%	43	438	13	747	1	116	0	3
org.apache.commons.lang3.exception		95%		95%	8	129	12	265	4	69	0	5
org.apache.commons.lang3.text.translate		99%		91%	19	137	8	235	5	58	0	13
org.apache.commons.lang3.event		91%		93%	1	31	5	84	0	23	0	4
org.apache.commons.lang3.concurrent		98%		96%	7	245	9	461	1	166	0	30
org.apache.commons.lang3.tuple		97%		89%	4	61	1	89	1	47	0	6
org.apache.commons.lang3.mutable		99%		97%	1	230	1	404	0	209	0	8
org.apache.commons.lang3.arch		100%		n/a	0	10	0	20	0	10	0	3
Total	2,939 of 70,680	95%	859 of 10,013	91%	915	8,349	714	15,258	143	3,273	1	245

Branch Coverage helps find part of the code that is not executed, It can also give us independent code which does not have any branch. It can also help us ensure that branches do not lead to erroneous behaviour.

Metric 3: Mutation Testing

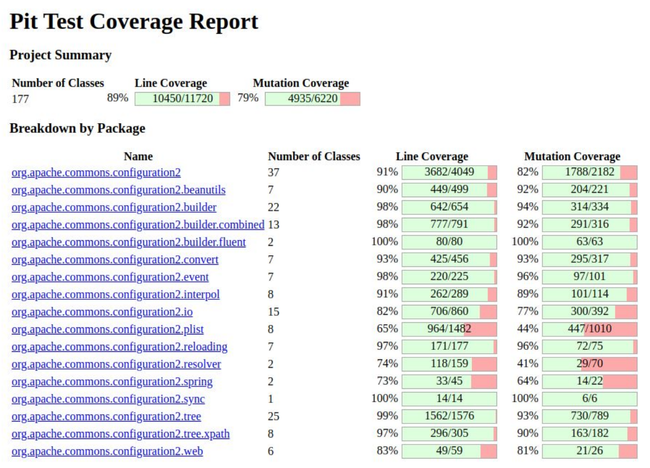
Mutation Testing is a fault-based testing technique which provides a testing criterion called the "mutation adequacy score". The mutation adequacy score can be used to measure the effectiveness of a test set in terms of its ability to detect faults.

The general principle involves creating small modifications to the original code, to create a new version called a mutant. Each mutant is a result of creating minor faults in the codebase. The test suite is then run against the original code base and all the mutants. The results are then compared. If the test results of the original code base and that of the mutant is different it implies that the test cases were able to catch the faults and the mutant is said to be killed. The Mutation score is the percentage of mutants killed. It is given by the following formula:

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

Mutation Testing can be used for testing software at the unit level, the integration level, and the specification level. It has been applied to many programming languages as a white box unit test technique. It is the most comprehensive technique to test a program. This is the method which checks for the effectiveness and accuracy of a testing program to detect the faults or errors in the system.

In our project, we have used the PIT tool to measure Mutation Scores. We choose PIT as it is fast, easy to use and actively supported. PIT can be run by adding a maven dependency and running the MutationCoverage goal or by using the PIT Eclipse plugin.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
177	89%	10450/11720	79%	4935/6220

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.apache.commons.configuration2	37	91%	3682/4049	82%	1788/2182
org.apache.commons.configuration2.beanutils	7	90%	449/499	92%	204/221
org.apache.commons.configuration2.builder	22	98%	642/654	94%	314/334
org.apache.commons.configuration2.builder.combined	13	98%	777/791	92%	291/316
org.apache.commons.configuration2.builder.fluent	2	100%	80/80	100%	63/63
org.apache.commons.configuration2.convert	7	93%	425/456	93%	295/317
org.apache.commons.configuration2.event	7	98%	220/225	96%	97/101
org.apache.commons.configuration2.interpol	8	91%	262/289	89%	101/114
org.apache.commons.configuration2.io	15	82%	706/860	77%	300/392
org.apache.commons.configuration2.plist	8	65%	964/1482	44%	447/1010
org.apache.commons.configuration2.reloading	7	97%	171/177	96%	72/75
org.apache.commons.configuration2.resolver	2	74%	118/159	41%	29/70
org.apache.commons.configuration2.spring	2	73%	33/45	64%	14/22
org.apache.commons.configuration2.sync	1	100%	14/14	100%	6/6
org.apache.commons.configuration2.tree	25	99%	1562/1576	93%	730/789
org.apache.commons.configuration2.tree.xpath	8	97%	296/305	90%	163/182
org.apache.commons.configuration2.web	6	83%	49/59	81%	21/26

Metric 4: Cyclomatic Complexity

Cyclomatic Complexity is a software metric used to measure the complexity of a program. It is a quantitative measure of independent paths in the source code of the program. An Independent path is defined as a path that has at least one edge which has not been traversed before in any other paths. Cyclomatic complexity can be calculated with respect to functions, modules, methods or classes within a program. If a piece of code has a high complexity number, then the probability of error is high with increased time for maintenance and troubleshooting.

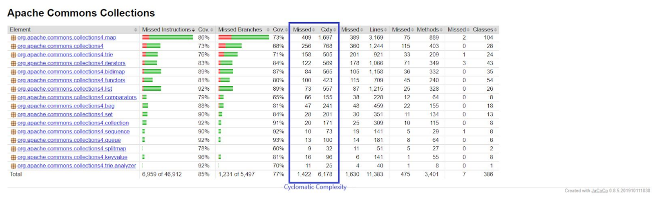
This metric was developed by Thomas J. McCabe in 1976 and it is based on a control flow representation of the program. Control flow depicts a program as a graph which consists of Nodes and Edges.

Mathematically, it is defined as:

$$V(G) = E - N + 2$$

Where, E= Number of edges and N= Number of nodes

The Jacoco tool has been used to capture complexity results in our project.



Apache Commons Collections

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.apache.commons.collections4.map		86%		73%	409	1,697	389	3,169	75	889	2	104
org.apache.commons.collections4		73%		68%	256	768	360	1,244	115	403	0	28
org.apache.commons.collections4.trie		76%		71%	158	505	201	921	33	209	1	24
org.apache.commons.collections4.iterators		83%		84%	122	569	178	1,066	71	349	3	43
org.apache.commons.collections4.bidimap		89%		87%	84	565	105	1,158	36	332	0	35
org.apache.commons.collections4.functors		81%		80%	100	423	115	709	45	240	0	54
org.apache.commons.collections4.list		92%		89%	73	557	87	1,215	25	328	0	26
org.apache.commons.collections4.comparators		79%		65%	66	155	38	228	12	64	0	8
org.apache.commons.collections4.bag		88%		81%	47	241	48	459	22	155	0	18
org.apache.commons.collections4.set		90%		84%	28	201	30	351	11	134	0	13
org.apache.commons.collections4.collection		92%		91%	20	171	25	309	10	115	0	8
org.apache.commons.collections4.sequence		90%		92%	10	73	19	141	5	29	1	8
org.apache.commons.collections4.queue		92%		93%	13	100	14	181	8	64	0	6
org.apache.commons.collections4.splitmap		78%		60%	9	32	11	51	5	27	0	2
org.apache.commons.collections4.keyvalue		96%		81%	16	96	6	141	1	55	0	8
org.apache.commons.collections4.trie.analyzer		92%		70%	11	25	4	40	1	8	0	1
Total	6,959 of 46,912	85%	1,231 of 5,497	77%	1,422	6,178	1,630	11,383	475	3,401	7	386

Metric 5: Package Instability

Package Instability metrics, unlike other object-oriented ones, don't represent the full set of attributes to assess individual object-oriented design, they only focus on the relationship between packages in the project.

Efferent Coupling (Ce): This metric is used to measure interrelationships between classes. As defined, it is a number of classes in a given package, which depends on the classes in other packages. It enables us to measure the vulnerability of the package to changes in packages on

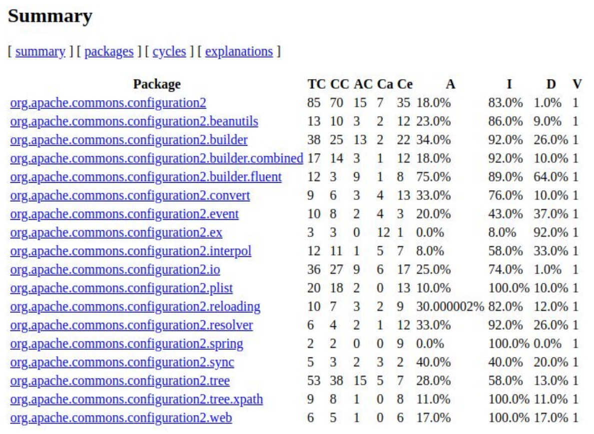
Afferent Coupling (Ca): This metric is an addition to metric Ce and is used to measure another type of dependencies between packages, i.e. incoming dependencies. It enables us to measure the sensitivity of remaining packages to changes in the analyzed package.

Instability (I): This metric is used to measure the relative susceptibility of class to changes. According to the definition, instability is the ratio of outgoing dependencies to all package dependencies and it accepts a value from 0 to 1.

The metric is defined according to the formula:

$$I = \frac{Ce}{Ce + Ca}$$

We have used JDepend tool to capture package instability matrices.



Summary

[summary] [packages] [cycles] [explanations]

Package	TC	CC	AC	Ca	Ce	A	I	D	V
org.apache.commons.configuration2	85	70	15	7	35	18.0%	83.0%	1.0%	1
org.apache.commons.configuration2.beanutils	13	10	3	2	12	23.0%	86.0%	9.0%	1
org.apache.commons.configuration2.builder	38	25	13	2	22	34.0%	92.0%	26.0%	1
org.apache.commons.configuration2.builder.combined	17	14	3	1	12	18.0%	92.0%	10.0%	1
org.apache.commons.configuration2.builder.fluent	12	3	9	1	8	75.0%	89.0%	64.0%	1
org.apache.commons.configuration2.convert	9	6	3	4	13	33.0%	76.0%	10.0%	1
org.apache.commons.configuration2.event	10	8	2	4	3	20.0%	43.0%	37.0%	1
org.apache.commons.configuration2.ex	3	3	0	12	1	0.0%	8.0%	92.0%	1
org.apache.commons.configuration2.interpol	12	11	1	5	7	8.0%	58.0%	33.0%	1
org.apache.commons.configuration2.io	36	27	9	6	17	25.0%	74.0%	1.0%	1
org.apache.commons.configuration2.plist	20	18	2	0	13	10.0%	100.0%	10.0%	1
org.apache.commons.configuration2.reloading	10	7	3	2	9	30.000002%	82.0%	12.0%	1
org.apache.commons.configuration2.resolver	6	4	2	1	12	33.0%	92.0%	26.0%	1
org.apache.commons.configuration2.spring	2	2	0	0	9	0.0%	100.0%	0.0%	1
org.apache.commons.configuration2.sync	5	3	2	3	2	40.0%	40.0%	20.0%	1
org.apache.commons.configuration2.tree	53	38	15	5	7	28.0%	58.0%	13.0%	1
org.apache.commons.configuration2.tree.xpath	9	8	1	0	8	11.0%	100.0%	11.0%	1
org.apache.commons.configuration2.web	6	5	1	0	6	17.0%	100.0%	17.0%	1

Metric 6: Defect Density

Defect Density is the number of confirmed defects detected in the software or a component during a defined period of development or operation, divided by the size of the software. It is one such process that enables one to decide if a piece of software is ready to be released.

Moreover, the importance of defect density is immense in Software Development Life Cycle (SDLC), as it is used to compare the relative number of defects in various software components, which further helps in identifying candidates for additional inspection or testing as well as for re-engineering or replacement.

Also, identifying defect prone components is made easy through defect density, which allows the testers to focus the limited resources into areas with the highest potential return on the investment. This further helps organisations and their businesses reach great heights of success, as they are able to deliver software and applications that are secure, safe, bug free and more.

It can be defined as:

$$Defect\ density\ = \frac{Total\ defects}{KLOC}$$

V. Assessment of the metrics

1. Subject software systems

For the Theoretical assessment of the selected metrics, we used 4 open-source software systems. The projects were selected as they were Java projects built in Maven and they met the requirements of LOC. For a better analysis we selected at least 4 versions of each project. The summary of the selected software systems' sizes are outlined in the table.

Project	Versions	LOC
Apache Commons Collections	4.0-4.4	67.k
Apache Commons Lang	3.5-3.9	80k
Alibaba fastjson	1.2.40 1.2.55 1.2.57 1.2.60 1.2.67	193k
Apache Commons Configuration	2.0-2.6	932k

2. Steps for collecting data

Once we selected the projects and metrics, the next step was to perform analysis on them. The analysis consists of two steps: collecting the data and performing correlation analysis on the data.

To collect the data for various metrics, the first step was to build the projects in eclipse. We faced several issues during the process like maven version conflicts with eclipse version. Secondly, we decided which metrics required version-level or project-level data for correlation analysis. We established that we needed version-level data for correlation between metrics 1,2,5 and 6 and project-level data for metrics 3 and 4.

As per the hypothesis which is given in the introduction we followed the below procedure to collect data :

1. Added dependency in pom.xml file for Jacoco,PIT eclipse and Jdepend.
2. Then after adding the dependency in pom.xml we did mvn clean & mvn install for generating reports in Jacoco.
3. For PIT mutation score we did (mvn -Drat.skip=true test) on cmd.
4. For Jdepend we added dependency and ran mvn site from cmd.
5. Got data in excel format from Jacoco, PIT eclipse and Jdepend (In webpage format .html).
6. Started making a query to look for bugs in JIRA website.
7. Started calculating defect density for JIRA with the help of CLOC tool and counting the no of defects to calculating post release defect density.
8. Started doing correlation between versions of the project in excel via Pearson Correlation & Spearman's Correlation.

VI. Correlation analysis

Few questions we solved in correlation analysis:

1. Jacoco csv files don't provide statement coverage and branch coverage, but "covered lines", "missed lines", "covered branch", "missed branch", "covered complexity", "missed complexity". We calculate statement coverage and branch coverage using statement coverage = covered/(covered + missed), branch coverage = covered/(covered + missed branches lines), complexity = covered + missed complexity.

2. Our metric 6 is version level, we compute version level of metric 1&2&5 in order to get the correlation of them with metric 6. Metric 1 & 2 we get version level by using total covered/total. For metric 5 we get version level by doing

$$Package\ Instability_{\ version} = \frac{Ce_{Total}}{Ce_{Total}+Ca_{Total}}$$

3. For branch coverage, some classes don't have a branch, so we don't count these classes.

4. For metrics 1&2, Jacoco computes all classes including inner classes, but Pitest report only provides the outer classes' mutation score. So we joined two tables accordingly to do correlation.

5. We compute Pearson correlation using formulas in spreadsheet, because it's easy to use. Some data may don't follow Normal Distribution, so we also calculated Spearman, but the result turned out to be similar to Pearson, so we decided not to add spearman in this report.

A. Correlation between Statement Coverage and Branch Coverage with Mutation Score

TABLE 2: Correlation of Statement Coverage and Branch Coverage with Mutation Score.

Project	Number of Classes	Pearson (Metric1 & Metric 3)	Pearson (Metric2 & Metric 3)
Apache Commons Collections	267	0.42865	0.14061
Apache Commons Lang	126	0.53826	-0.10525

Alibaba fastjson	132	0.4715	0.203
Apache Commons Configuration	176	0.822	0.7536

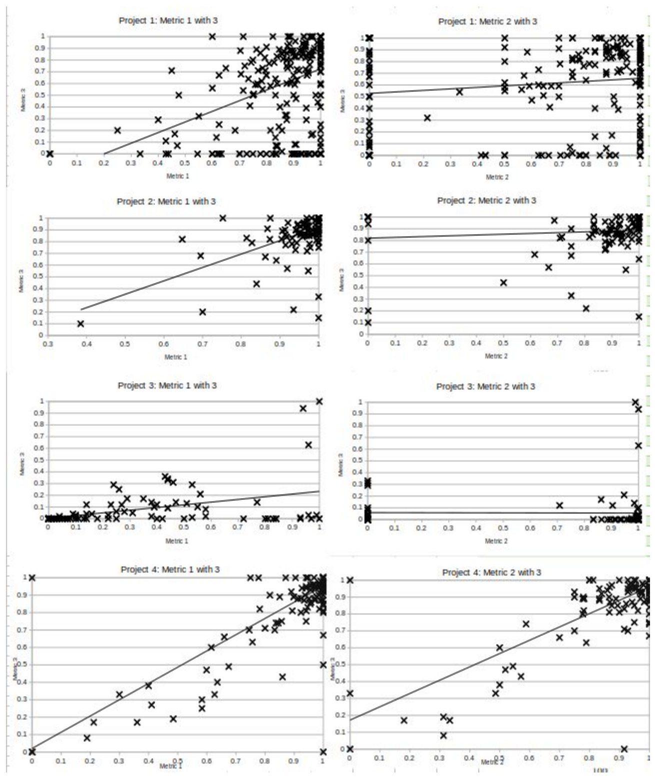


Fig 2: Scatter plot of Metric 1&2 over 3

From Table2 and Fig 2, we can see that most projects' statement coverage and branch coverage have a positive correlation with mutation score.

Apache Commons Lang's branch coverage shows that there is low negative correlation with Mutation Score. Its most branch coverages are around 90%, so the data is not good for correlation analysis. Therefore, -1.10525 is unreliable.

B. Correlation between Statement Coverage and Branch Coverage with McCabe Complexity

TABLE 3: Correlation of Statement Coverage and Branch Coverage with McCabe Complexity.

Project	Number of Classes	Pearson Correlation (Metric1 & Metric 4)	Pearson Correlation (Metric2 & Metric 4)
Apache Commons Collections	474	-0.02233	0.2638
Apache Commons Lang	121	0.00871	0.13122
Alibaba fastjson	182	0.00722	0.1374
Apache Commons Configuration	285	-0.0805	-0.2265

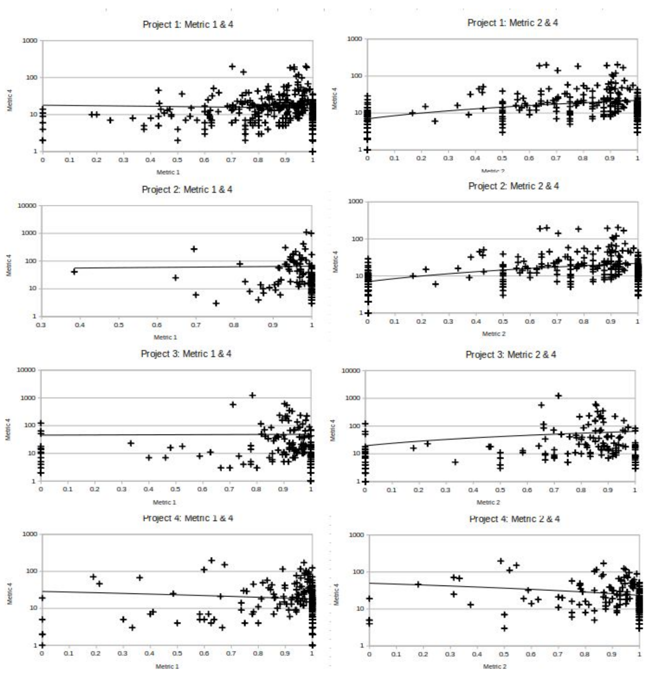


Fig 3: Scatter plot of Metric 1&2 over 4

As we can see from Table3 and Fig. 3, most correlation calculations get a negative number or around 0, that means the correlation between coverage metric and McCabe Complexity are negative or very weak.

On the other hand, Apache Common Collection branch coverage gets 0.2638 with its McCabe Complexity, which means there is a weak positive correlation.

C. Correlation between Statement Coverage and Branch Coverage with Post Release Defect Density

TABLE 4: Correlation of Statement Coverage and Branch Coverage with Post Release Defect Density.

Project	Versions	Metric 6	Pearson (Metric 1 Metric 6)	Pearson (Metric 2 Metric 6)

Apache Commons Collections	4.0 4.1 4.2 4.3 4.4	0.000658 0.000131 4.7E-05 1.5E-05 3.1E-05	-0.68254	0.2733
Apache Commons Lang	3.5 3.6 3.7 3.8 3.9	0.620501 0.225345 0.35546 0.117212 0.35694	-0.847195	-0.763877
Alibaba fastjson	1.2.40 1.2.55 1.2.57 1.2.60 1.2.67	3.47E-05 5.57E-05 5.5E-05 0 0	0.86334	0.37998
Apache Commons Configuration	V2.0 V2.1 V2.2 V2.3 V2.4 V2.5 V2.6	1.5E-4 8.85E-05 8.75E-05 0 0 1.44E-05 0	-0.8195	0.6

Post release defect density is a version level metric. In order to calculate the correlation, we also compute the version level of statement coverage and branch coverage.

The results in Table 4 show that three projects' statement coverage have a negative correlation with post release defect density. However, Alibaba fastjson gets 0.86334 coefficient, a strong positive correlation.

Apache Commons Collection, Commons Configuration and Alibaba fastjson's branch coverage has a middle level positive correlation with post release defect density. Apache Commons Lang and have a negative correlation.

D. **Correlation between Package Instability and Post Release Defect Density**

TABLE 5: Correlation of Package Instability and Post Release Defect Density.

Project	Versions	Metric 5	Metric 6	Pearson (Metric 5 & Metric 6)
Apache Commons Collections	4.0 4.1 4.2 4.3 4.4	0.5043 0.483 0.4817 0.4849 0.5032	0.000658 0.000131 4.7E-05 1.5E-05 3.1E-05	0.58144
Apache Commons Lang	3.5 3.6 3.7 3.8 3.9	0.7523 0.7545 0.7545 0.7545 0.7545	0.620501 0.225345 0.35546 0.117212 0.35694	-0.8472
Alibaba fastjson	1.2.40 1.2.55 1.2.57 1.2.60 1.2.67	0.7797 0.7725 0.7725 0.757 0.7587	3.47E-05 5.57E-05 5.5E-05 0 0	0.8168
Apache Commons Configuration	V2.0 V2.1 V2.2 V2.3 V2.4 V2.5 V2.6	0.7809 0.7839 0.7839 0.7807 0.7791 0.7765 0.7786	1.5E-4 8.85E-05 8.75E-05 0 0 1.44E-05 0	-0.9062

For correlation between metric 5 and 6, the result we get shows no consistency, two of them get positive correlation, the other two get negative correlation. Here we have to point out that Apache Common Lang and Common Configuration both have very small number of bugs, so that the number of bugs may not reflect the real defect of software.

VII. Threats To Validity

A. External Validity: External threats referred to generalized results.Since we have taken four open source projects from github which has a SLOC >= 60K.So we have tried to be consistent with the dataset we got and the project which we have taken various domains and sizes.

B. Internal Validity: Internal threats are those which are generally related to the environment they are carried out .Since if we take an example of maven projects which we have taken it is a possibility that the project doesn't follow maven 100%. Maybe it has some features of ANT or gradle.So there may be some threats regarding that that we can consider.

C. Construct Validity: Since we have applied all six metrics which are Statement & Branch Coverage . Mutation score, Cyclomatic Complexity, Package Instability & Post defect density. For e.g. these coverages such as code coverage via statement or branch coverage via jacoco give us a measure of effectiveness of the test suite.The same can be there in all other metrics which we have calculated which emphasis more on effectiveness of code.

VIII.Conclusion

According to the correlation we get for Metric 1 & 2 with 3, we can conclude that suites with higher statement and branch coverage, also have a higher mutation score. In other words, test suites with higher coverage show better effectiveness.

According to the correlation we get for Metric 1 & 2 with 4, we know that if a class has a higher complexity, prone to have a lower coverage. The conclusion is consistent with rational hypotheses.

According to the correlation we get for Metric 1 & 2 with 6, it is not clear for the relationship between coverage and post release defect density. The reason is that the number of bugs are very small, many defects may have not been found. In this case, the data we have is not good enough to analyze correlation.

For the correlation we get for Metric 5 & 6, we start correlation analysis with the hypothesis that the higher Instability, the higher defect density. However, we get the Pearson correlation of -0.9062 and -0.8168 for project 2 and 4, which is a contradictory correlation. Again, the reason may be the number of these two projects' bugs are very small, many defects may have not been found. In this case, the data we have is not good enough to analyze correlation.

We get Metric 5 Package Instability using Jdepend, itself is package level metrics. In order to do analysis with version level metric 6, we calculate version level of Metric 5 by the following formula:

$$Package\ Instability_{\ version} = \frac{Ce_{Total}}{Ce_{Total}+Ca_{Total}}$$

The formula is not proved theoretically.

In addition, each project may have unseen factors that the correlation analysis didn't consider. We shall improve our analysis by considering as many factors as possible.

References

[1] Bansiya, Jagdish, Letha H. Etzkorn, Carl G. Davis and Wei Li. "A Class Cohesion Metric For Object-Oriented Designs." JOOP 11 (1999): 47-52.

[2] J. M. Bieman, D. Dreilinger and Lijun Lin, "Using fault injection to increase software test coverage," Proceedings of ISSRE '96: 7th International Symposium on Software Reliability Engineering, White Plains, NY, USA, 1996, pp. 166-174.The above paper describes the best approach for test coverage for branch & statement coverage.

[3] S. Almugrin, W. Albattah, O. Alaql, M. Alzahrani and A. Melton, "Instability and Abstractness Metrics Based on Responsibility," 2014 IEEE 38th Annual Computer Software and Applications Conference, Vasteras, 2014, pp. 364-373.